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

Tarek El-Ghazawi, GWU

Brad Chamberlain, Cray

Vijay Saraswat, IBM

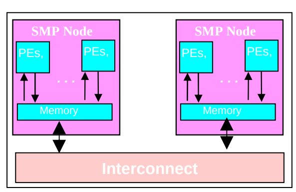
Overview

- **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

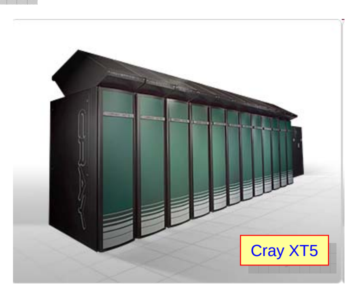
The common architectural landscape

SMP Clusters



Massively Parallel Processor Systems

IBM Blue Gene



Cray XT5

4

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

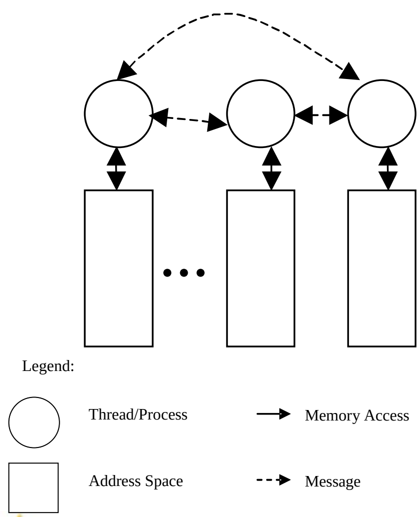
5

Examples of Parallel Programming Models

- Message Passing
- Shared Memory (Global Address Space)
- Partitioned Global Address Space (PGAS)

6

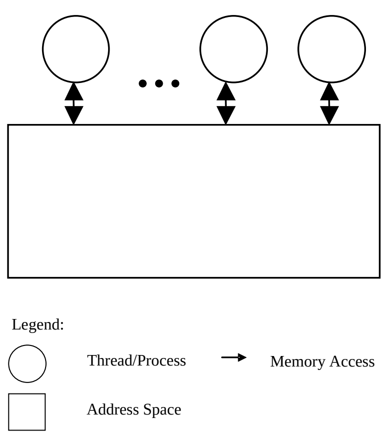
The Message Passing Model



- 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

7

The Shared Memory Model

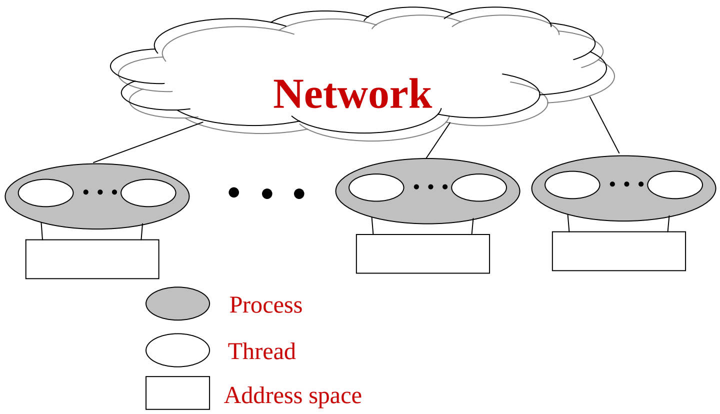


- Concurrent threads with shared space
- Positive:
  - Simple statements
  - Read remote memory via an expression
  - Write remote memory through assignment
- Negative:
  - Manipulating shared data leads to synchronization requirements
  - Does not allow locality exploitation
- Example: OpenMP, Java

8

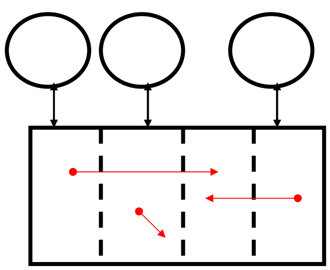
Hybrid Model(s)

Example: Shared + Message Passing



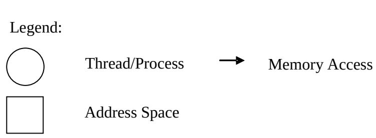
- Example: OpenMP at the node (SMP), and MPI in between

The PGAS Model



PGAS

UPC, CAF, X10



- 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

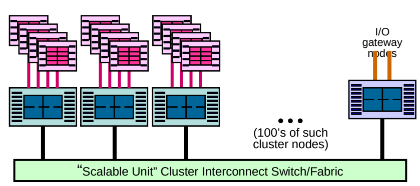
PGAS vs. other programming models/languages

	UPC, X10, Chapel	MPI	OpenMP	HPF
Memory model	PGAS (Partitioned Global Address Space)	Distributed Memory	Shared Memory	Distributed Shared Memory
Notation	Language	Library	Annotations	Language
Global arrays?	Yes	No	No	Yes
Global pointers/references?	Yes	No	No	No
Locality Exploitation	Yes	Yes, necessarily	No	Yes

11

The heterogeneous/accelerated architectural landscape

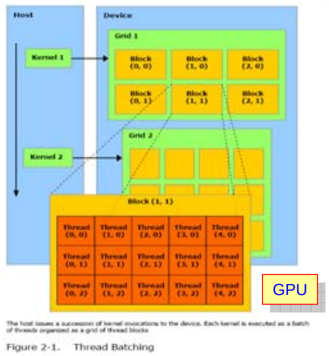
Cray XT5h: FPGA/Vector-accelerated Opteron



Road Runner: Cell-accelerated Opteron

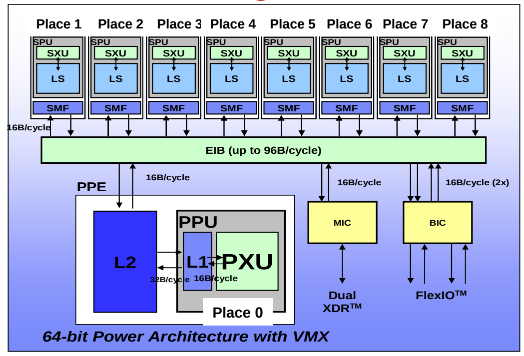
12

Example accelerator technologies

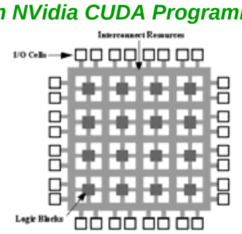


GPU

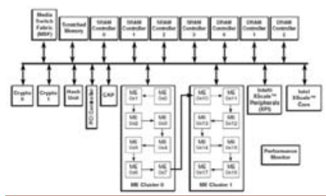
From NVidia CUDA Programming Guide



Cell Processor



FPGA



Multi-core w/ accelerators *(Intel IXP 2850)*

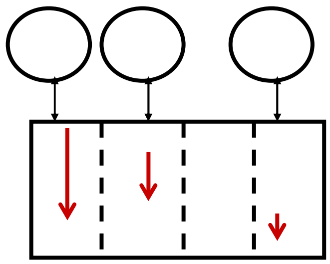
13

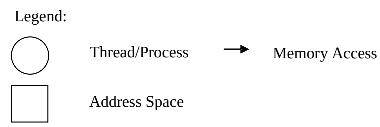
The current architectural landscape

- 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.

14

Asynchronous PGAS





PGAS

UPC, CAF, X10

- Explicit concurrency
- SPMD is a special case
- Asynchronous activities can be started and stopped in a given space partition
- Asynchronous activities can be used for active messaging
  - DMAs,
  - fork/join concurrency, do-all/do-across parallelism

Concurrency is made explicit and programmable.

15

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

16

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)

1

SC08 PGAS Tutorial

Unified Parallel C - UPC

Tarek El-Ghazawi
tarek@gwu.edu

The George Washington University

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

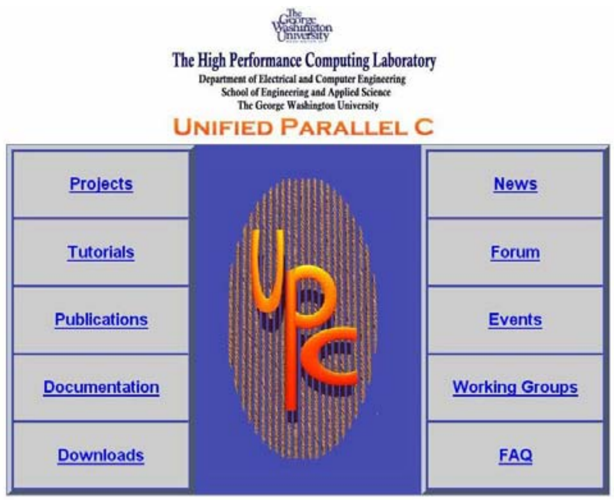
5

UPC Systems

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

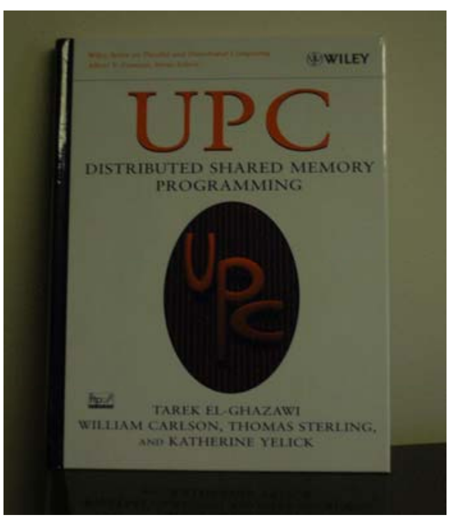
6

UPC Home Page
http://www.upc.gwu.edu



7

UPC textbook now available



- ***UPC: Distributed Shared Memory Programming***
  Tarek El-Ghazawi
  William Carlson
  Thomas Sterling
  Katherine Yelick
- **Wiley, May, 2005**
- **ISBN: 0-471-22048-5**

8

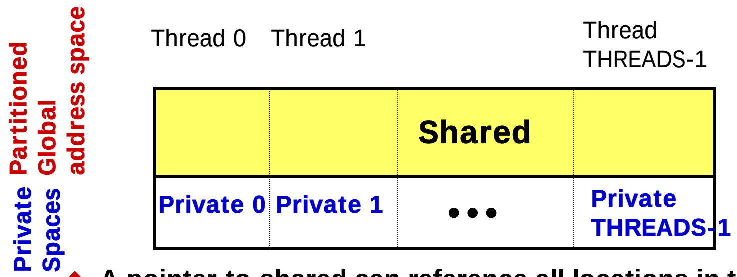
What is UPC?

- 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

UPC Memory Model



- 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];
}
```

Iteration #:	Thread 0	Thread 1
	0	1
	2	3
	v1[0]	v1[1]
	v1[2]	v1[3]
	…	
	v2[0]	v2[1]
	v2[2]	v2[3]
	…	
	v1plusv2[0]	v1plusv2[1]
	v1plusv2[2]	v1plusv2[3]
	…	

Shared Space

13

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];
}
```

Iteration #:	Thread 0	Thread 1
	0	1
	2	3
	v1[0]	v1[1]
	v1[2]	v1[3]
	…	
	v2[0]	v2[1]
	v2[2]	v2[3]
	…	
	v1plusv2[0]	v1plusv2[1]
	v1plusv2[2]	v1plusv2[3]
	…	

Shared Space

14

3^{rd} 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];
}
```

	Thread 0	Thread 1
Iteration #:	0	1
	2	3
	v1[0]	v1[1]
	v1[2]	v1[3]
	…	
	v2[0]	v2[1]
	v2[2]	v2[3]
	…	
	v1plusv2[0]	v1plusv2[1]
	v1plusv2[2]	v1plusv2[3]
	…	

Shared Space

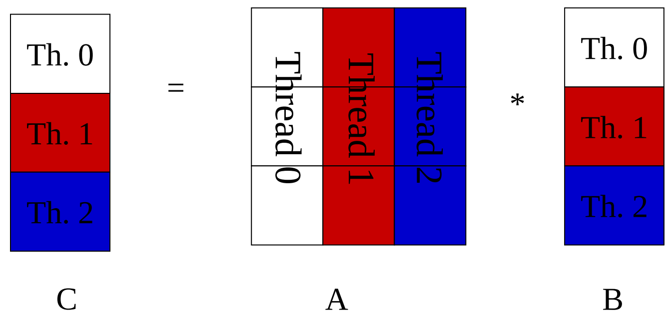
15

Example: UPC Matrix-Vector Multiplication- Default Distribution

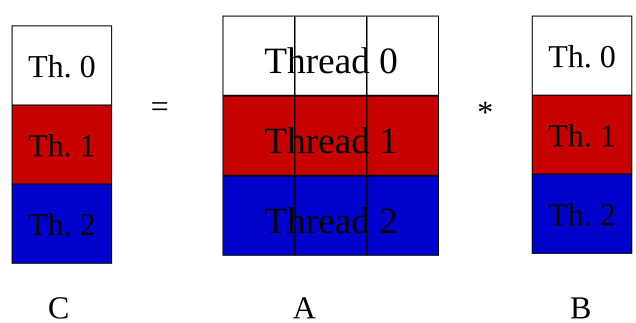
```
// vect_mat_mult.c
#include <upc_relaxed.h>

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];
      }
}
```

16

Data Distribution



A Better Data Distribution



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];
        }
}
```

19

Shared and Private Data

<u>Examples of Shared and Private Data Layout:</u>

Assume THREADS = 3

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

will result in the layout:

Thread 0	Thread 1	Thread 2
x		
y[0]	y[1]	y[2]
z	z	z

20

Shared and Private Data

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

will result in the following data layout:

Thread 0	Thread 1	Thread 2
A[0][0]	A[0][1]	A[0][2]
A[1][0]	A[1][1]	A[1][2]
A[2][0]	A[2][1]	A[2][2]
A[3][0]	A[3][1]	A[3][2]

21

Shared and Private Data

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

will result in the following data layout:

Thread 0	Thread 1	•••	Thread (THREADS-1)
A[0][0]	A[0][1]		A[0][THREADS-1]
A[0][THREADS]	A[0][THREADS+1]	•••	A[0][2*THREADS-1]
A[1][0]	A[1][1]		A[1][THREADS-1]
A[1][THREADS]	A[1][THREADS+1]		A[1][2*THREADS-1]

22

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:

```
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:

$$\left\lfloor \frac{i}{blocksize} \right\rfloor \bmod THREADS$$

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

Shared and Private Data

Assume THREADS = 4

```
shared [3] int A[4][THREADS];
```

will result in the following data layout:

Thread 0	Thread 1	Thread 2	Thread 3
A[0][0]	A[0][3]	A[1][2]	A[2][1]
A[0][1]	A[1][0]	A[1][3]	A[2][2]
A[0][2]	A[1][1]	A[2][0]	A[2][3]
A[3][0]	A[3][3]		
A[3][1]			
A[3][2]			

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

27

Usage Example of Special Operators

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

- upc_localsizeof(sharray) → 10*sizeof(int)
- upc_localsizeof(a) →10 *sizeof(int)
- upc_localsizeof(i) →1
- upc_blocksizeof(a) →1
- upc_elementsizeof(a) →sizeof(int)

28

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

29

String functions in UPC

- 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

30

UPC Pointers

Where does it point to?

Where does it reside?		Private	Shared
	Private	PP	PS
	Shared	SP	SS

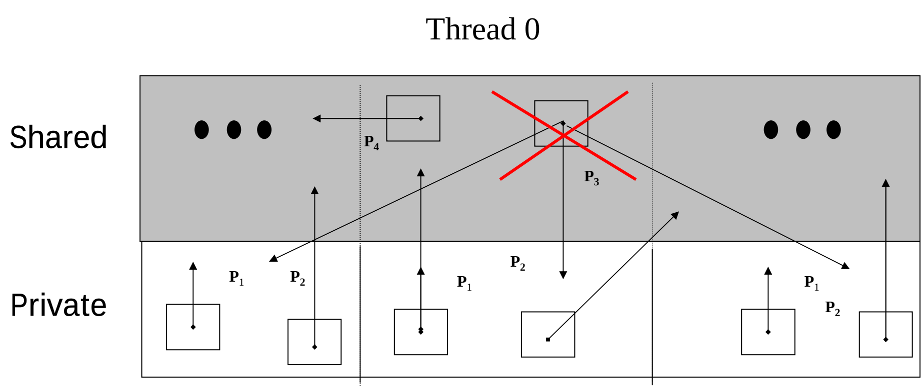
UPC Pointers

- **How to declare them?**

```
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.**

UPC Pointers



UPC Pointers

- 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*/

UPC Pointers

- In UPC pointers to shared objects have three fields:
  - thread number
  - local address of block
  - phase (specifies position in the block)

Thread #	Block Address	Phase

- Example: Cray T3E implementation

Phase	Thread	Virtual Address
63 49	48 38	37 0

35

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

36

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 upc_addrfield(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

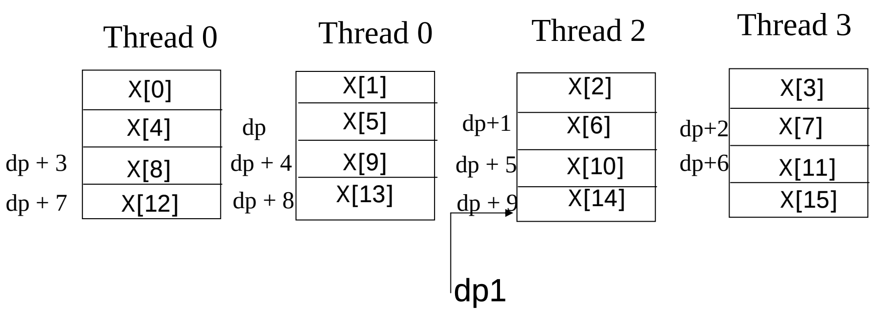
UPC Pointers

pointer to shared Arithmetic Examples:

Assume THREADS = 4

```
#define N 16
shared int x[N];
shared int *dp=&x[5], *dp1;
dp1 =  dp + 9;
```

UPC Pointers



39

UPC Pointers

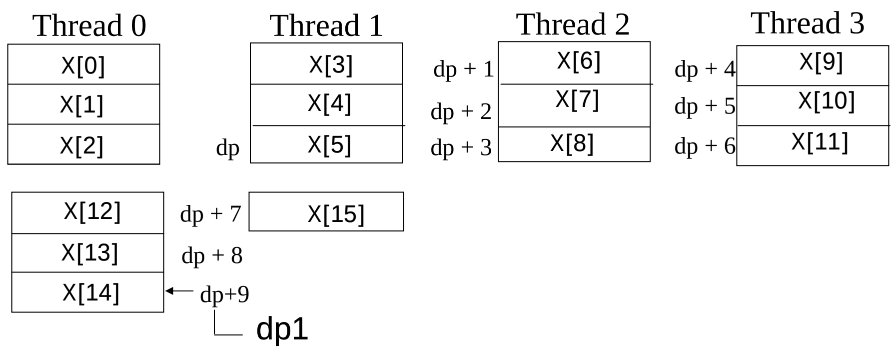
Assume THREADS = 4

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

dp1 = dp + 9;

40

UPC Pointers



41

UPC Pointers

<u>Example Pointer Castings and Mismatched Assignments:</u>

- 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**

42

UPC Pointers

- **Mismatched Assignments**

Assume THREADS = 4

shared int x[N];

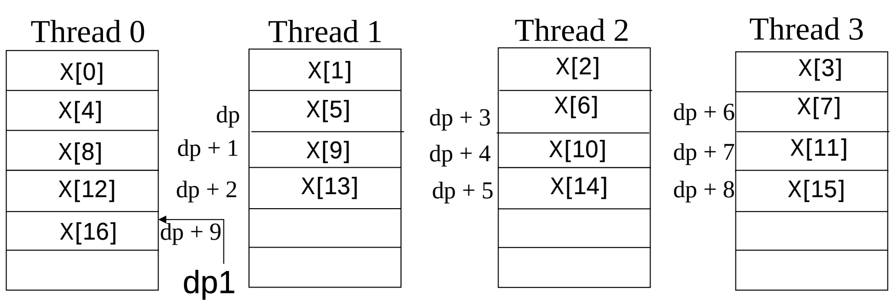
shared[3] int *dp=&x[5], *dp1;

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**

43

UPC Pointers



44

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

45

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

46

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

47

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

48

Work Sharing: upc_forall()

- **Example 3: Implicitly with distribution by chunks**

```
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

i	i*THREADS	i*THREADS/100
0..24	0..96	0
25..49	100..196	1
50..74	200..296	2
75..99	300..396	3

49

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**

50

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”

51

Collective Global Memory Allocation

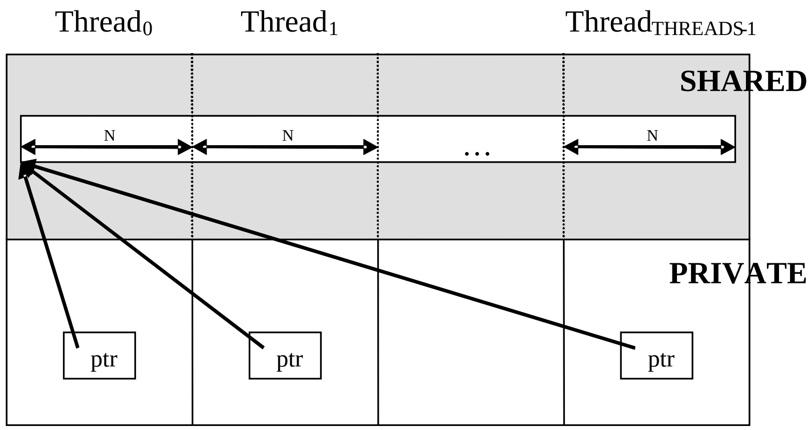
```
shared void *upc_all_alloc
         (size_t nblocks, size_t nbytes);
```

nblocks: 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 :**
  shared [nbytes] char[nblocks * nbytes]

52

Collective Global Memory Allocation




```
shared [N] int *ptr;
ptr = (shared [N] int *)
        upc_all_alloc( THREADS, N*sizeof( int ) );
```


Global Memory Allocation

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

nblocks : number of blocks
nbytes : block size

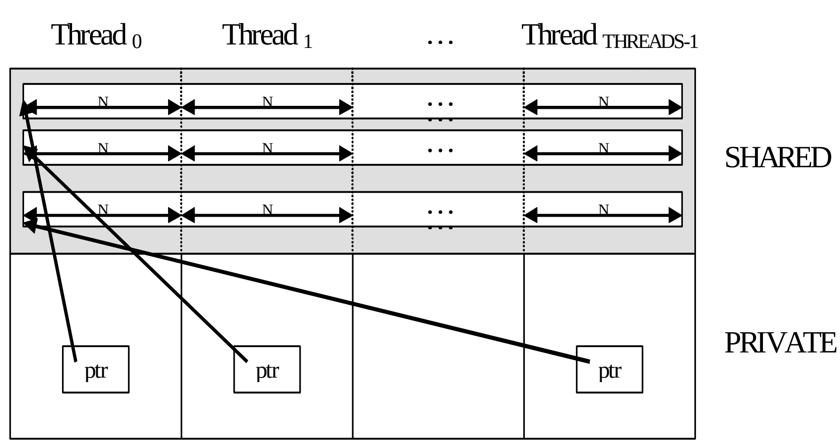
- **Non collective, expected to be called by one thread**
- **The calling thread allocates a contiguous memory region in the shared space**
- **Space allocated per calling thread is equivalent to :** shared [nbytes] char[nblocks * nbytes]
- **If called by more than one thread, multiple regions are allocated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

Global Memory Allocation

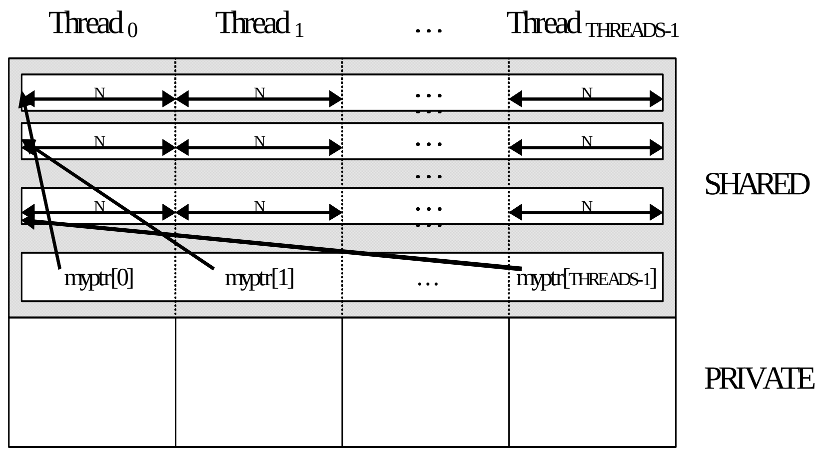
```
shared [N] int *ptr;

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 ));
```

55



56



57

Local-Shared Memory Allocation

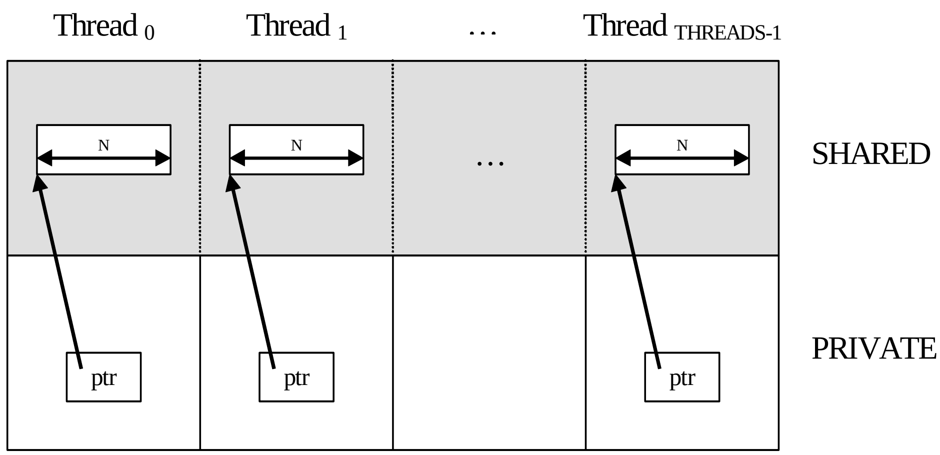
```
shared void *upc_alloc (size_t nbytes);
```

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**

58

Local-Shared Memory Allocation




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

59

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

60

Example: Matrix Multiplication in UPC

- 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:

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

61

Doing it in C

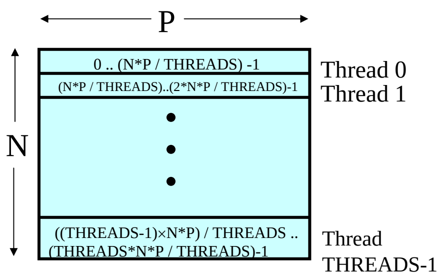
```
#include <stdlib.h>

#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];
        }
     }
}
```

62

Domain Decomposition for UPC

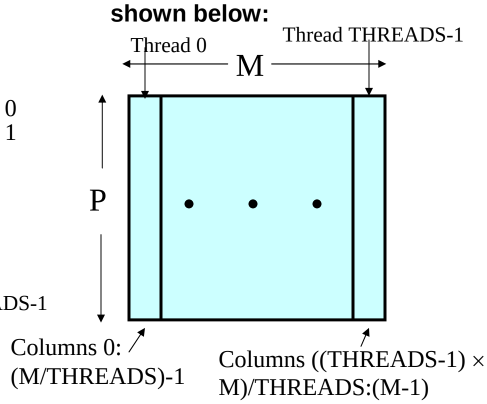
Exploiting locality in matrix multiplication

- **A (N × P) is decomposed row-wise into blocks of size (N × P) / THREADS as shown below:**



•**Note:** N and M are assumed to be multiples of THREADS

- **B(P × M) is decomposed column-wise into M/ THREADS blocks as shown below:**



63

UPC Matrix Multiplication Code

```
#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];
       }
   }
}
```

64

UPC Matrix Multiplication Code with Privatization

```
#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];
}  }    }
```

65

UPC Matrix Multiplication Code with block copy

```
#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++; &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_local[l][j];
        }
   }
}
```

66

UPC Matrix Multiplication Code with Privatization and Block Copy

```
#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));
   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_local[l][j];
} }   }
```

67

Matrix Multiplication with dynamic memory

```
#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_elemsizeof(*a));
  c=upc_all_alloc(THREADS,(N*M/THREADS)*
       upc_elemsizeof(*c));
  b=upc_all_alloc(P*THREADS,(M/THREADS)*
       upc_elemsizeof(*b));

  upc_forall(i = 0 ; i<N ; i++; &c[i*M]) {
     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];
     }
  }
}
```

68

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 GF(2), $X_{63}+1$.

For each a_i :

$$T[LSB_{n-1\ldots0}(a_i)] = T[LSB_{n-1\ldots0}(a_i)] \text{ XOR } S[MSB_{m-1\ldots0}(a_i)]$$

2 Sets of typical problem sizes:

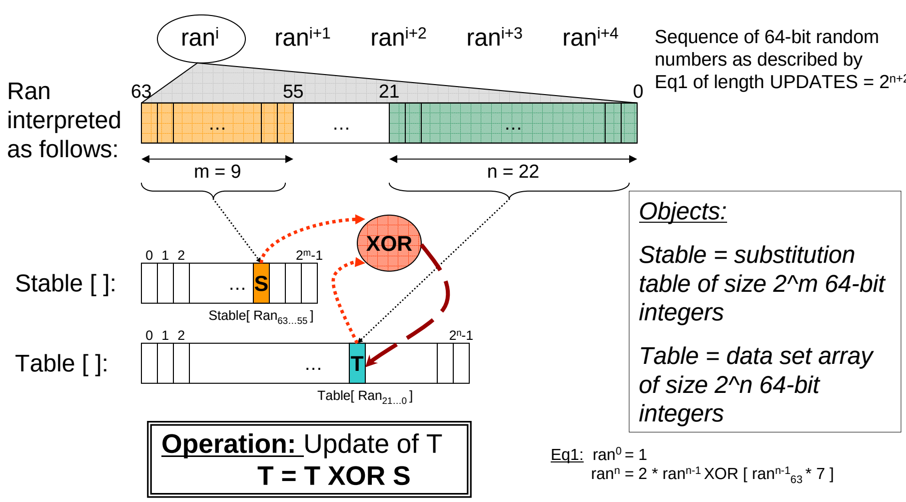
(a) m=9, n=8, 9, max integer size possible

(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

69

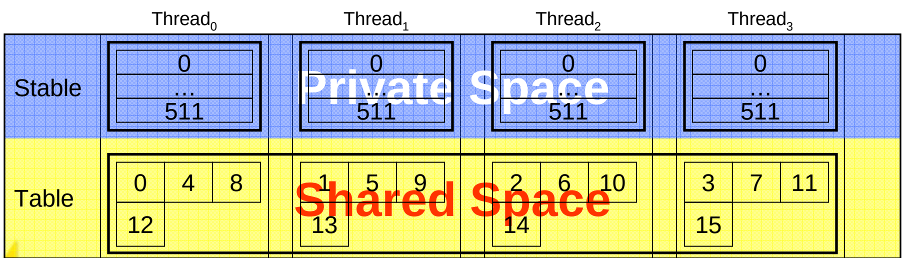
Example: RandomAccess



70

RandomAccess – UPC

```
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

```
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)];
     }
  }
}
```

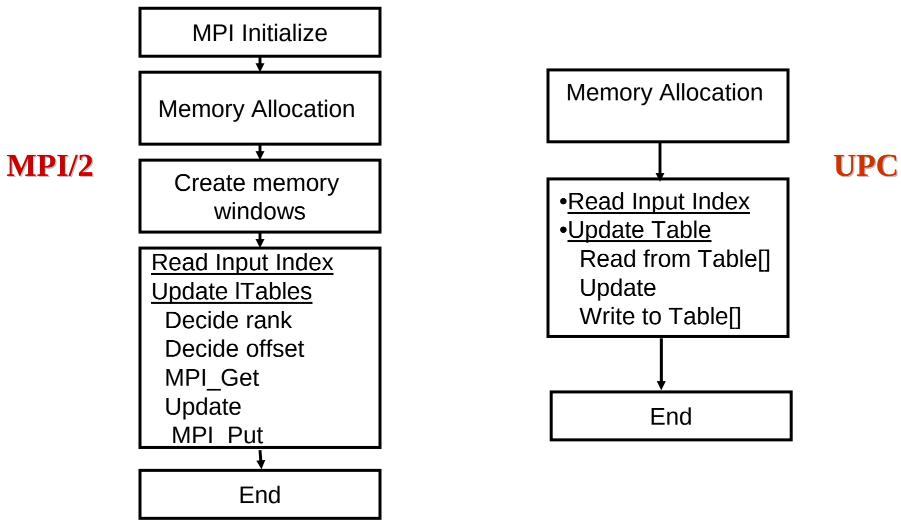
Initialize the **local** part of Table[]

Synchronization

Workload distribution

As-it-is in SEQ code

Expressing Memory Operations Efficiently: Large Histogram (HPCC RA)



73

Compact Code

	Random Access (#lines)	%Increase
C	144	-
UPC	158	9.72%
MPI/2	210	45.83%
MPI/1	344	138.89%

***Study done with HPCC 0.5alpha compliant code**

74

Less Conceptual Complexity

		Work Distr.	Data Distr.	Comm.	Synch. & Consist.	Misc. Ops	Sum	Overall Score
RandomAccess MPI/2	# Parameters	26	9	35	5	6	81	151
	# Function Calls	0	2	8	4	4	18	
	# Keywords with rank and np	15	6	8	0	2	31	
	# MPI Types	0	5	10	4	2	21	
	Notes	11 If 5 For	1 memalloc 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		
RandomAccess UPC	# Parameters	19	2	0	0	2	23	42
	# Function Calls	0	1	0	5	2	8	
	# Keywords	5	1	0	0	0	6	
	# UPC Constructs & UPC Types	3	2	0	0	0	5	
	Notes	3 forall 4 if 1 for	2 shared 1 upc_all_alloc		5 barriers	2 global_exit		

75

Synchronization

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

76

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

77

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

78

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)

79

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

80

Memory Consistency Example

```
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.

81

UPC Libraries

- UPC Collectives
- UPC-IO

82

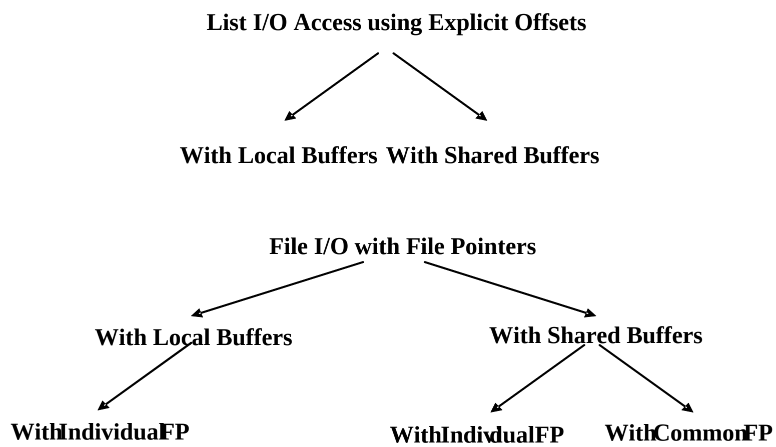
Overview UPC Collectives

- **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



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

85

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

86

Reduced Coding Effort is Not Limited to Random Access– NPB Examples

		SEQ1	UPC	SEQ2	MPI	UPC Effort (%)	MPI Effort (%)
NPB-CG	#lines	665	710	506	1046	6.77	106.72
	#chars	16145	17200	16485	37501	6.53	127.49
NPB-EP	#lines	127	183	130	181	44.09	36.23
	#chars	2868	4117	4741 F	6567	43.55	38.52
NPB-FT	#lines	575	1018	665	1278	77.04	92.18
	#chars	13090	21672	22188	44348	65.56	99.87
NPB-IS	#lines	353	528	353	627	49.58	77.62
	#chars	7273	13114	7273	13324	80.31	83.20
NPB-MG	#lines	610	866	885	1613	41.97	82.26
	#chars	14830	21990	27129	50497	48.28	86.14

$$UPC_{effort} = \frac{\#UPC - \#SEQ1}{\#SEQ1} \qquad MPI_{effort} = \frac{\#MPI - \#SEQ2}{\#SEQ2}$$

SEQ1 is C

SEQ2 is from NAS, all FORTRAN except for IS

87

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)

1

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 uptodate 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 Tardieu, Krishna Venkat, Jose Castanos, Ankur Narang
- X10 Tools
  - Philippe Charles, Robert Fuhrer, Matt Kaplan
- Emeritus
  - Kemal Ebcioglu, 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 Tipparaju, Jarek Nieplocha (PNNL)
  - Kathy Yelick, Dan Bonachea (Berkeley)
  - Several others at IBM

Selected Recent Publications

1. "Solving large, irregular graph problems using adaptive work-stealing", to appear in ICPP 2008.
2. "Constrained types for OO languages", to appear in OOPSLA 2008.
3. "Type Inference for Locality Analysis of Distributed Data Structures", PPoPP 2008.
4. "Deadlock-free scheduling of X10 Computations with bounded resources", SPAA 2007
5. "A Theory of Memory Models", PPoPP 2007.
6. "May-Happen-in-Parallel Analysis of X10 Programs", PPoPP 2007.
7. "An annotation and compiler plug-in system for X10", IBM Technical Report, Feb 2007.
8. "X10: An Object-Oriented Approach to Non-Uniform Cluster Computing", OOPSLA conference, October 2005.
9. "Concurrent Clustered Programming", CONCUR conference, August 2005.

Tutorials

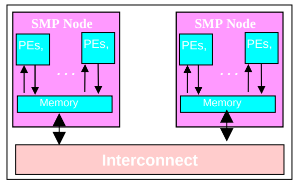
- TiC 2006, PACT 2006, OOPSLA 2006, PPoPP 2007, SC 2007
- Graduate course on X10 at U Pisa (07/07)
- Graduate course at Waseda U (Tokyo, 04/08)

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

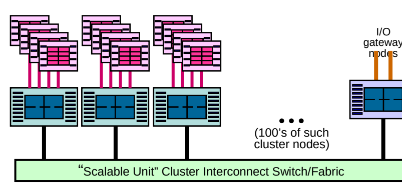
IBM Blue Gene

Cray XT5

5

The heterogeneous/accelerated architectural landscape

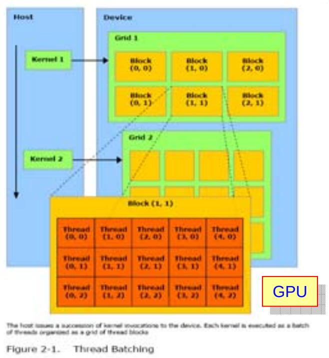
Cray XT5h: FPGA/Vector-accelerated Opteron



Road Runner: Cell-accelerated Opteron

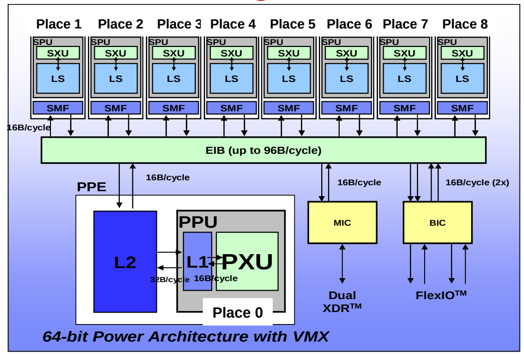
6

Example accelerator technologies

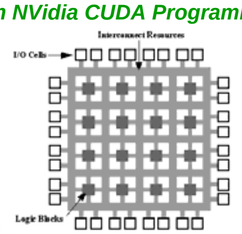


GPU

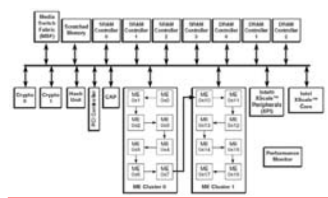
From NVidia CUDA Programming Guide



Cell Processor

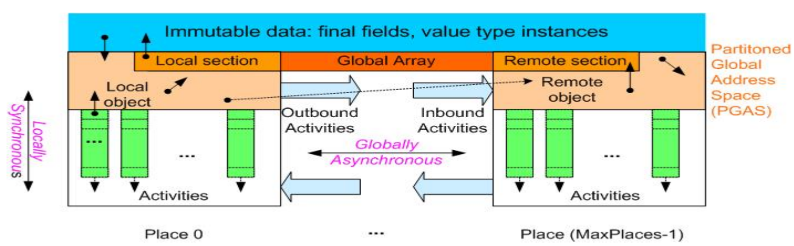


FPGA



Multi-core w/ accelerators *(Intel IXP 2850)*

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**
- **Asyncs used for active messaging**
  - **(remote asyncs),**
  - **DMAs,**
  - **fork/join concurrency, do-all/do-across parallelism**
  - **SPMD is a special case**

Concurrency is made explicit and programmable.

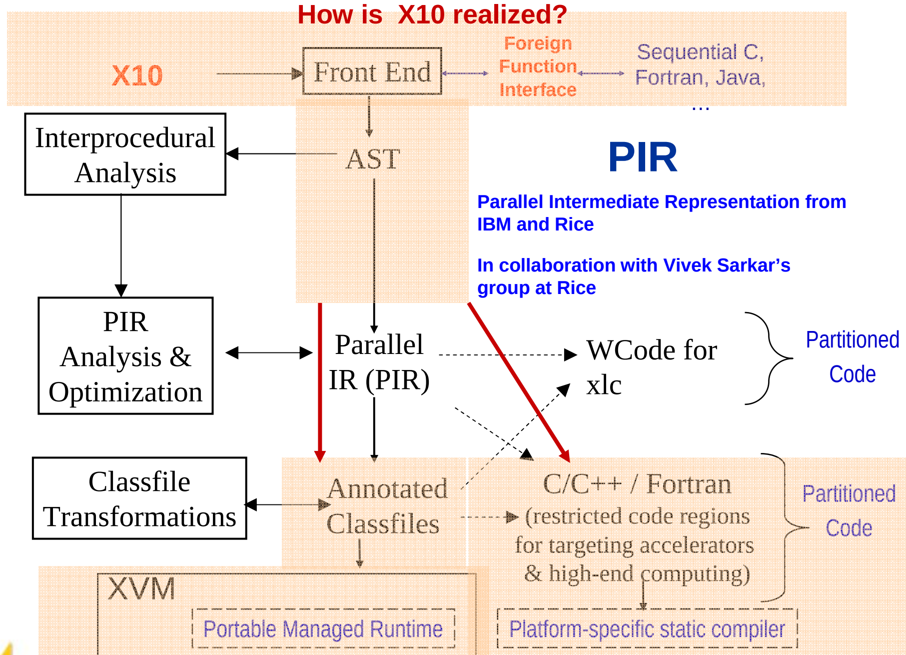
What is X10?

- 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

9

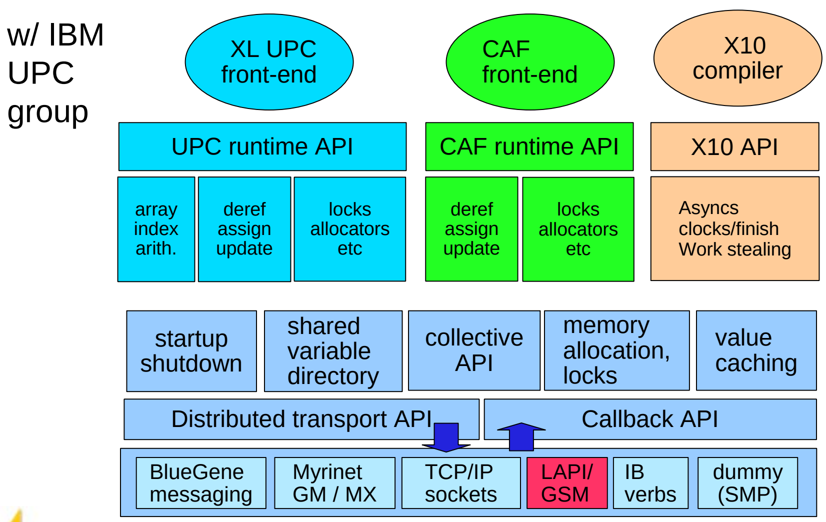
How is X10 realized?



Planned X10 Toolchain / Current X10 Toolchain

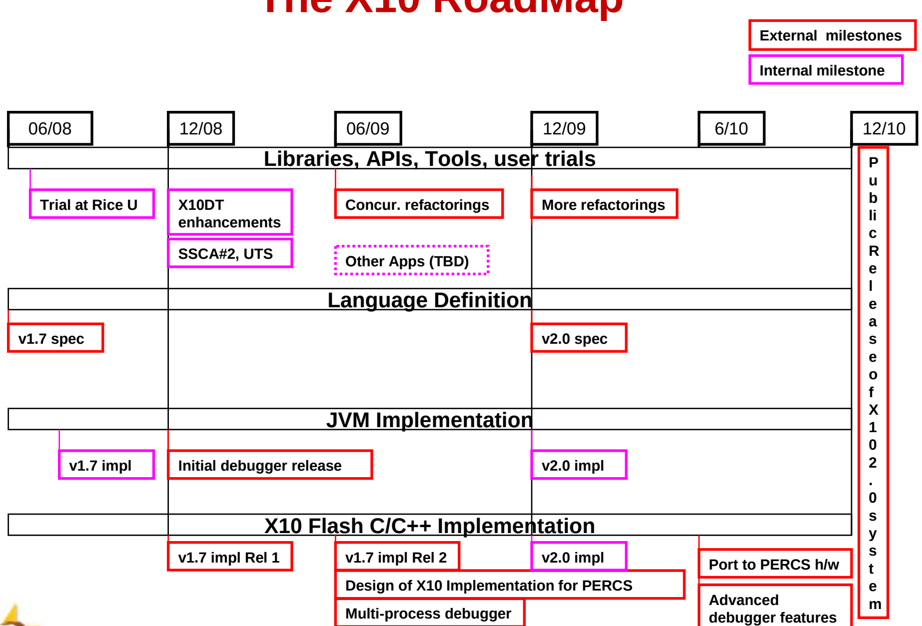
10

PGAS runtime structure



11

The X10 RoadMap



12

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.

14

Overview of Features

- A lot of sequential features of Java inherited unchanged
  - Classes (w/ single inheritance)
  - Intefaces, (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

15

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 (Remote Refs)

16

Points and Regions

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

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

Operations
- p1.rank
  - returns rank of point p1
- p1(i)
  - returns element (i mod p1.rank) if i < 0 or i >= p1.rank
- p1 < p2, p1 <= p2, p1 > p2, p1 >= p2
  - returns true iff p1 is lexicographically <, <=, >, or >= p2
  - only defined when p1.rank and p1.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

17

Distributions and Arrays

- Distrbutions 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 initializaers
  - 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

18

Generic classes

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

21

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)

22

async

Stmt ::= **async(p,l)** *Stmt*

- Creates a new child activity that executes statement S
- Returns immediately
- S may reference **final** variables in enclosing blocks
- Activities cannot be named
- Activity cannot be aborted or cancelled

cf Cilk's spawn

```
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

L:finish S

- Execute S, but wait until all (transitively) spawned asyncs have terminated.

Stmt ::= **finish** *Stmt*

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.

```
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;
  }
```

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**

```
// target defined in lexically
// enclosing scope.
atomic boolean CAS(Object old,
                   Object new) {
   if (target.equals(old)) {
     target = new;
     return true;
   }
   return false;
}
```

```
// push data onto concurrent
// list-stack
Node node = new Node(data);
atomic {
     node.next = head;
     head = node;
}
```


when

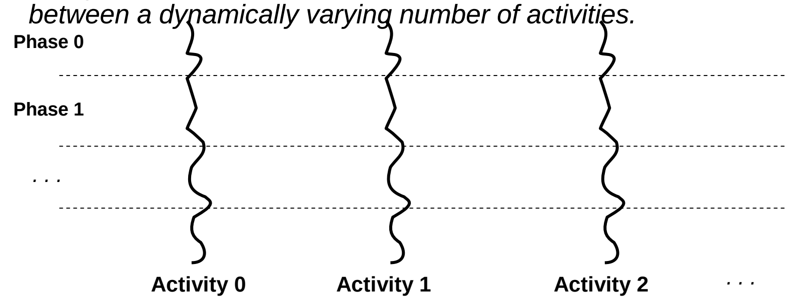
Stmt ::= WhenStmt
***WhenStmt ::=* when (*Expr*) *Stmt* |**
WhenStmt* or *(Expr) Stmt

- **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;
       }
   }
}
```

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.*



27

Clocks – Main operations

var c = Clock.make();

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

async(…) clocked (c1,c2,…) S

ateach(…) clocked (c1,c2,…) S

foreach(…) clocked (c1,c2,…) S

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

c.resume();

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

next;

- 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

28

Fundamental X10 Property

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

X10 Programming Idioms

Random Access (LOC=79)

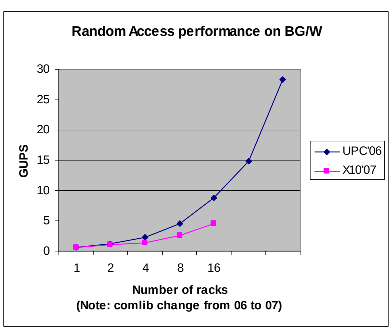
Core algorithm:

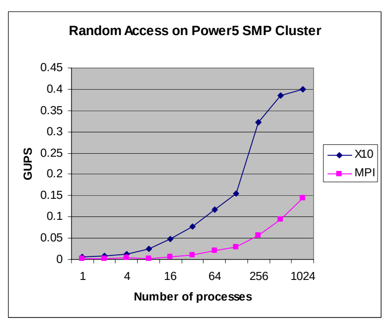
```
static RAUpdate(logLocalTableSize:Int, Table: Array[Long]{rail}){
 finish ateach((p) in UNIQUE) {
   val localTableSize=1<<logLocalTableSize,
        TableSize=localTableSize*NUM_PLACES,
        mask=TableSize-1,
        NumUpdates=4*localTableSize;
   var ran:Long =HPCC_starts(p*NumUpdates);
  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

SPMD + Remote atomic operations (X10Flash Implementation)

31

Performance





	1	2	4	8	16	32	64
UPC	0.58	1.15	2.28	4.49	8.83	14.8	28.3
X10	0.58	1.04	1.31	2.51	4.51		

	1	2	4	8	16	32	64	128	256	512	1024
X10	0.005613	0.007755	0.013426	0.025345	0.048783	0.078405	0.11704	0.155228	0.323228	0.384118	0.398779
MPI	0.001342	0.002124	0.004144	0.002796	0.005622	0.01084	0.020375	0.029293	0.055678	0.094022	0.143919

32

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();}
  …
}
```

Single-Node Work-stealing (Java implementation) – ICPP 08 paper

33

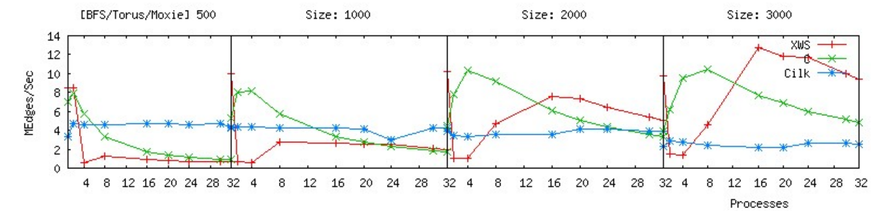
Adaptive stealing -- Code

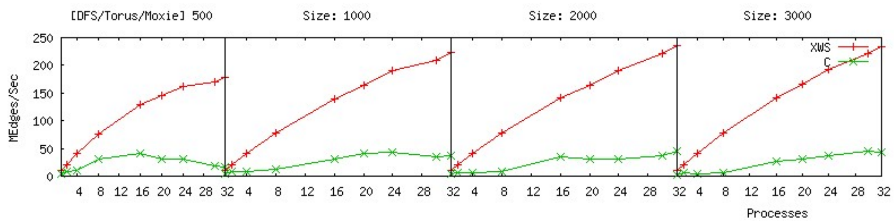
```
def compute(w:Worker):Void throws StealAbort {
  w.popAndReturnFrame();
  val newList:List = null, newLength:Int = 0;
  var oldList:V = this, par:V = parent, batchSize:Int=0;
  do {
    val v = oldList, edges = v.neighbors;
    oldList = v.next;
    for (var k:Int = 0; k < edges.length; ++k) {
      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);
      <push onto batch, and push batch onto deque if full>}
  } while (oldList != null);
}
```

Implemented in application space.

34

Torus on moxie: DFS vs BFS

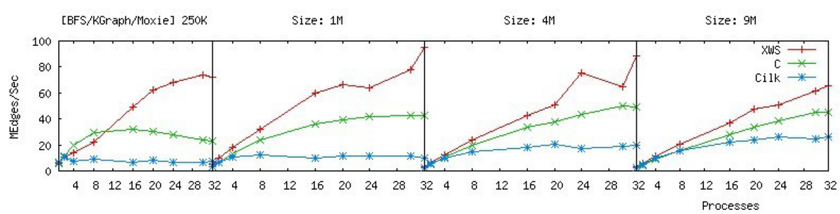


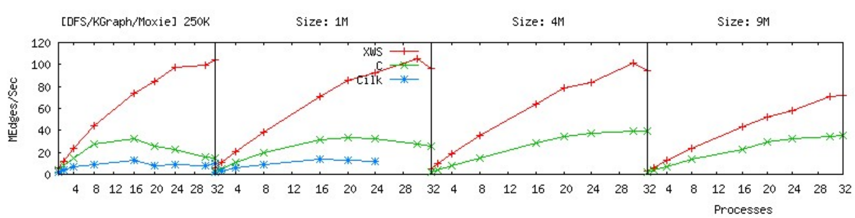


Moxie= 32-way Sun Fire T200 server, 1.2GHz, 32GB memory, 8KB data ache/processor 16KB instrn cache/core, 2MB integrated L2 cache

35

KGraph on Moxie: BFS vs DFS





36

FT Code (LOC=137)

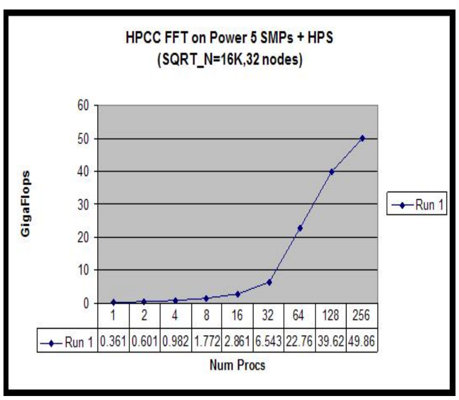
Key routine: global transpose

```
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);
      }}}
```

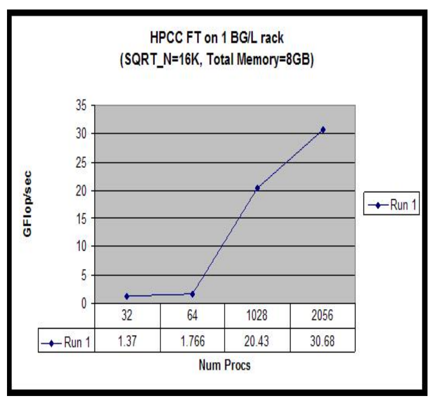
Communication/computation overlap across multiple nodes (X10Flash implementation)

37

FT Performance



Num Procs	1	2	4	8	16	32	64	128	256
Run 1	0.361	0.601	0.982	1.772	2.861	6.543	22.76	39.62	49.86



Num Procs	32	64	1028	2056
Run 1	1.37	1.766	20.43	30.68

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.)

38

LU Code (LOC=291)

Core algorithm:

```
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++;}
    }}}
```

Single-place program

2d-block distribution of workers

Step checks dependencies and executes appropriate basic operation (LU, bSolve, lower).

(Code doesn't compute y.)

Communication/computation overlap across multiple nodes (X10Flash implementation)

Example of steps

```
def step(val startY:Int, startX:Rail[Int]):Boolean {
    visitCount++;
    if (count==maxCount) {
      return I<J ? stepIltJ() : (I==J ? stepIeqJ() : stepIgtJ());
    } 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;
    }
  }
```

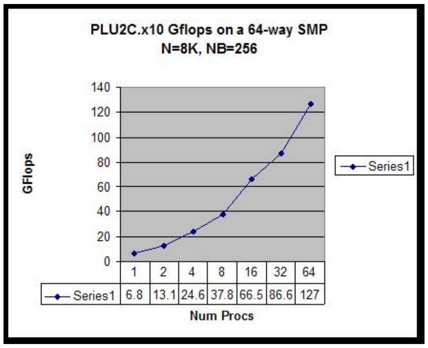
stepIltJ → wait; backsolve

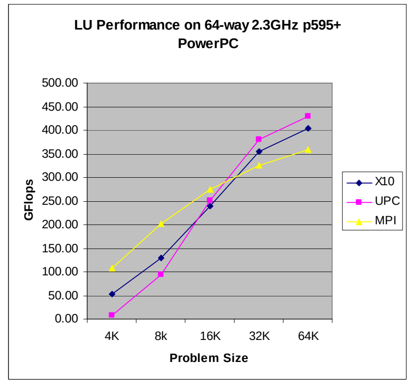
stepIeqJ → wait; control panel LU factorization

stepIgtJ → wait; compute lower, participate in LU factorization

Call BLAS for DGEMM.

LU Performance (SMP only)





IBM J9 JVM (64-bit)

ProbSz	BlkSz	PX	PY	X10	UPC (Gflops)	MPI
4096	256	8	8	53.24	7.10	108.00
8192	256	8	8	129.30	93.55	201.90
16384	256	8	8	238.65	250.14	274.30
32768	256	8	8	354.36	380.16	325.60
65536	256	8	8	404.43	428.60	358.40

41

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)

1

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× 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

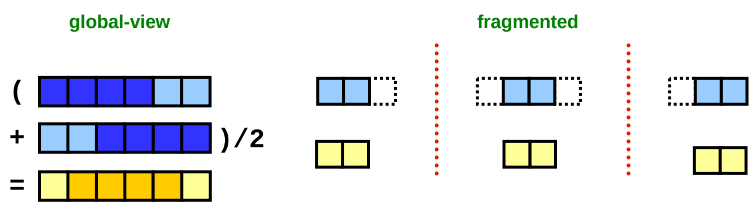
programming model: the mental model a programmer uses when coding using a language, library, or other notation

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

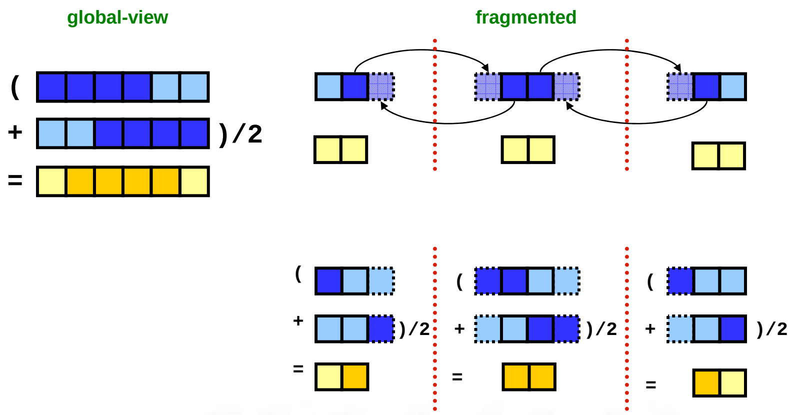
Global-view vs. Fragmented

Problem: “Apply 3-pt stencil to vector”



Global-view vs. Fragmented

Problem: "Apply 3-pt stencil to vector"



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

```
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

```
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));
  }
  if (iHaveLeftNeighbor) {
    send(left, a(1));
    recv(left, a(0));
  }
  forall i in 1..locN {
    b(i) = (a(i-1) + a(i+1))/2;
  }
}
```


Global-view vs. SPMD Code

Problem: "Apply 3-pt stencil to vector"

Assumes *numProcs* divides *n*; a more general version would require additional effort

global-view

```
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

```
def main() {
  var n: int = 1000;
  var locN: int = n/numProcs;
  var a, b: [0..locN+1] real;
  var innerLo: int = 1;
  var innerHi: int = locN;

  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;
  }
}
```

MPI SPMD pseudo-code

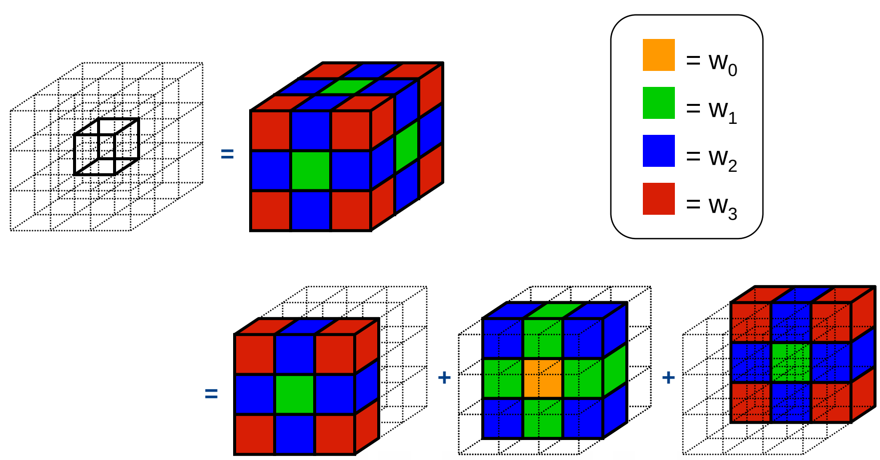
Problem: "Apply 3-pt stencil to vector"

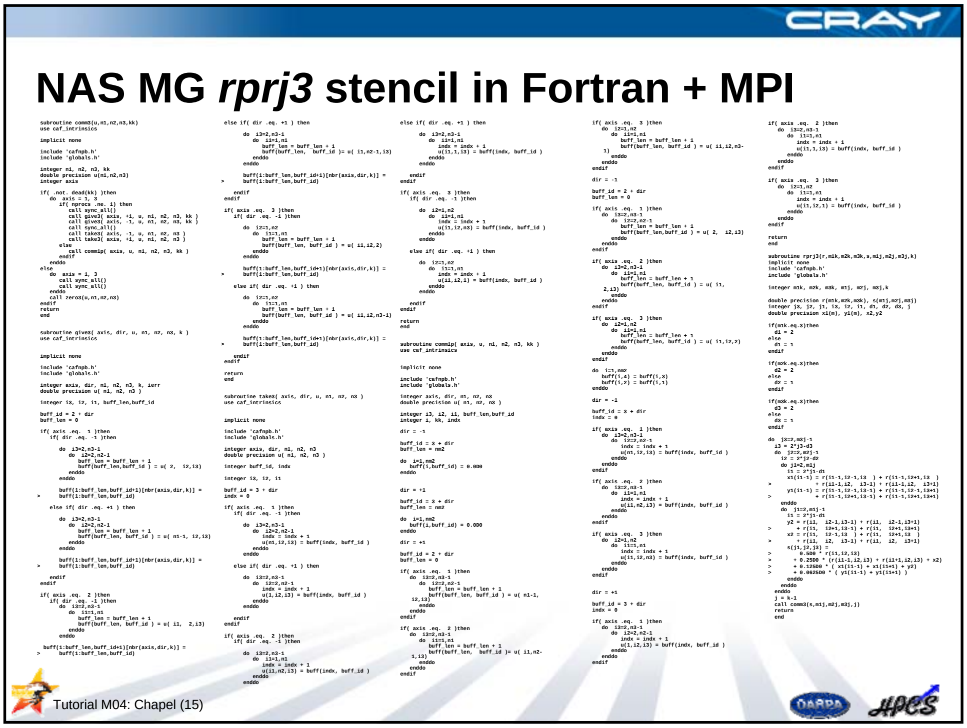
SPMD (pseudocode + MPI)

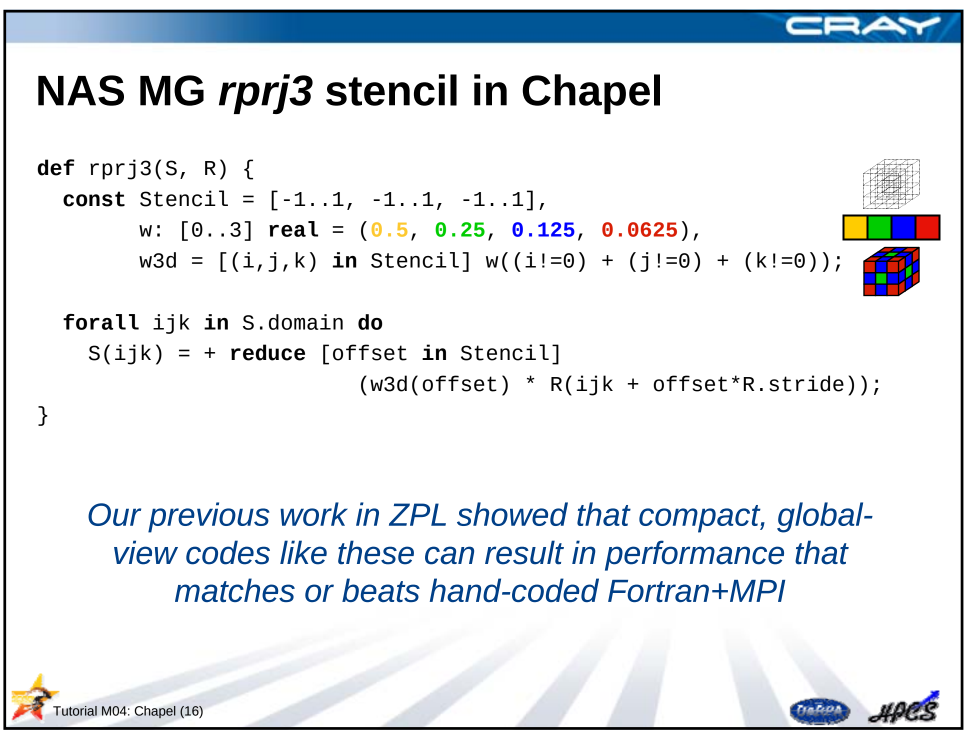
```
var n: int = 1000, locN: int = n/numProcs;
var a, b: [0..locN+1] real;
var innerLo: int = 1, innerHi: int = locN;
var numProcs, myPE: int;
var retval: int;
var status: MPI_Status;

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




CRAY
NAS MG *rprj3* stencil in Fortran + MPI
Tutorial M04: Chapel (15)
DARPA
HPCS


CRAY
NAS MG *rprj3* stencil in Chapel
def rprj3(S, R) {
const Stencil = [-1..1, -1..1, -1..1],
w: [0..3] real = (0.5, 0.25, 0.125, 0.0625),
w3d = [(i,j,k) in Stencil] w((i!=0) + (j!=0) + (k!=0));
forall ijk in S.domain do
S(ijk) = + reduce [offset in Stencil]
(w3d(offset) * R(ijk + offset*R.stride));
}
Our previous work in ZPL showed that compact, global-view codes like these can result in performance that matches or beats hand-coded Fortran+MPI
Tutorial M04: Chapel (16)
DARPA
HPCS

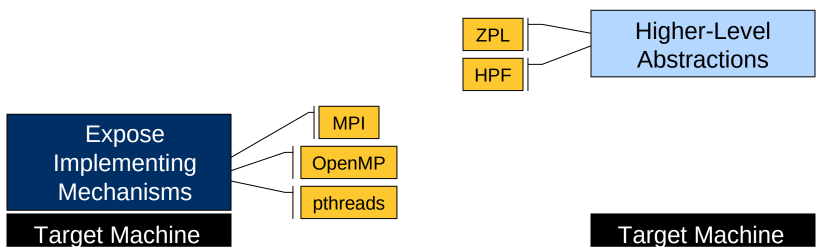
Summarizing Fragmented/SPMD Models

- 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

	data / control
communication libraries:	
MPI, MPI-2	fragmented / fragmented/SPMD
SHMEM, ARMCI, GASNet	fragmented / SPMD
shared memory models:	
OpenMP, pthreads	global-view / global-view (trivially)
PGAS languages:	
Co-Array Fortran	fragmented / SPMD
UPC	global-view / SPMD
Titanium	fragmented / SPMD
HPCS languages:	
Chapel	global-view / global-view
X10 (IBM)	global-view / global-view
Fortress (Sun)	global-view / global-view

Parallel Programming Models: Two Camps



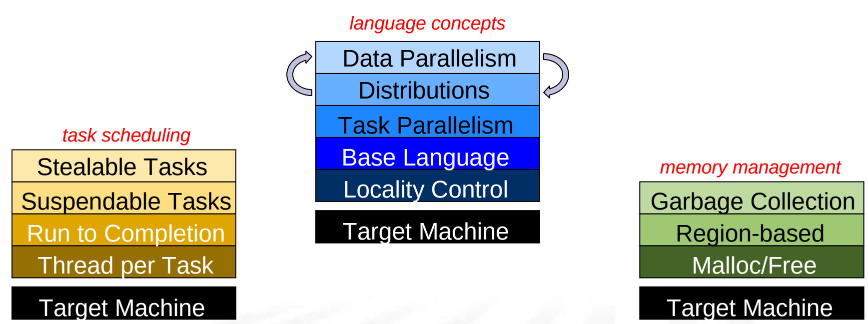
"Why is everything so painful?"

"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



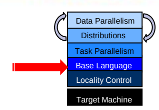
CRAY

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

CRAY

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#)

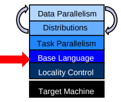
  Java, C#: OOP, type safety

  C++: generic programming/templates (without adopting its syntax)

  C, Modula, Ada: syntax

CRAY

Base Language: Standard Stuff



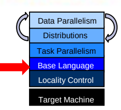
- Lexical structure and syntax based largely on C/C++

```
{ 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

CRAY

Base Language: Departures



- **Syntax:** declaration syntax differs from C/C++

```
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++

```
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)

```
config const n = 100;  // override with --n=1000000
```

- **Tuples**
- **Iterators…**

Base Language: Motivation for Iterators

Given a program with a bunch of similar loops…	Consider the effort to convert them from RMO to CMO…	Or to tile the loops…

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

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…

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

Or to change the distribution of the work/arrays in that MPI program…

Or to label them as parallel for OpenMP or a vectorizing compiler…

Or to do *anything* that we do with loops all the time as a community…

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);
}       }
```

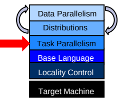
- iterators are used to drive loops:

```
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



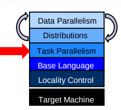
begin: creates a task for future evaluation

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

sync: waits on all begins created within a dynamic scope

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


Task Parallelism: Task Coordination



sync variables: store full/empty state along with value

```
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
```

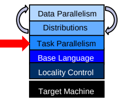
single-assignment variables: writable once only

```
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

```
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:

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

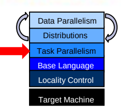
```
cobegin {
  computeTaskA(…);
  computeTaskB(…);
  computeTaskC(…);
} // implicit join
```

coforall: creates a task per loop iteration

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

Tutorial M04: Chapel (31)

Producer/Consumer example



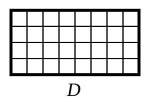

```
var buff$: [0..buffersize-1] sync int;

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)…;
  }
}
```

Tutorial M04: Chapel (32)

Domains

domain: a first-class index set

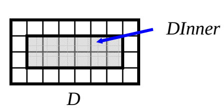
```
var m = 4, n = 8;

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



Domains

domain: a first-class index set

```
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:

```
var A, B: [D] real;
```

- Iteration (sequential or parallel):

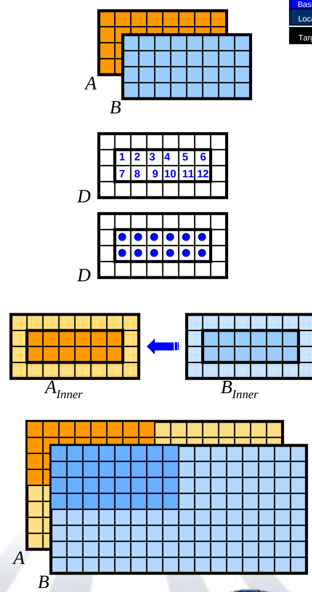
```
    for    ij in Inner { … }
or: forall ij in Inner { … }
or: …
```

- Array Slicing:

```
A[Inner] = B[Inner];
```

- Array reallocation:

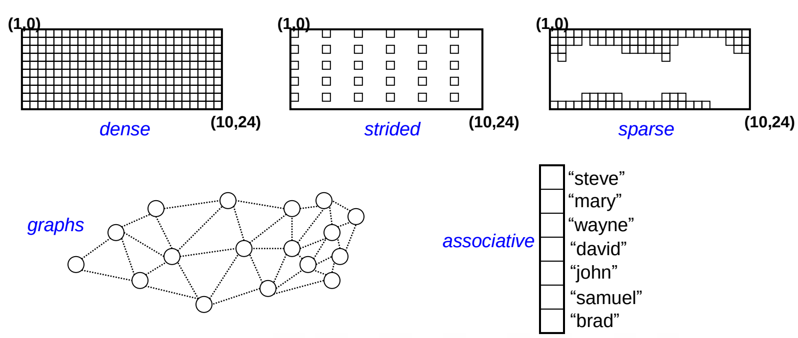
```
D = [1..2*m, 1..2*n];
```



Data Parallelism: Domains

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

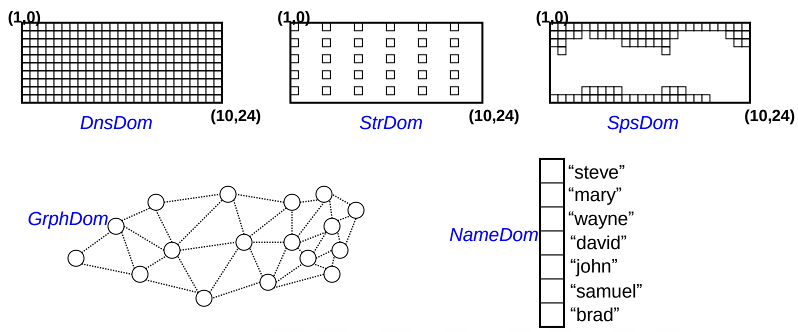
…integer tuples…



…anonymous… …or arbitrary values.

Data Parallelism: Domain Declarations

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

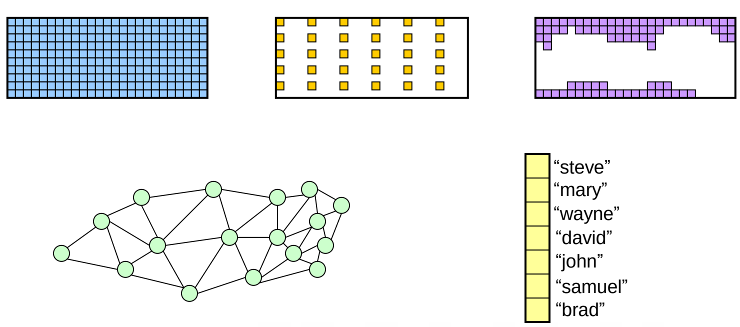



```
var GrphDom: domain(opaque),
    NameDom: domain(string) = readNames();
```

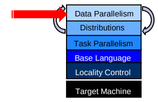

Data Parallelism: Domains and Arrays

Domains are used to declare arrays…

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

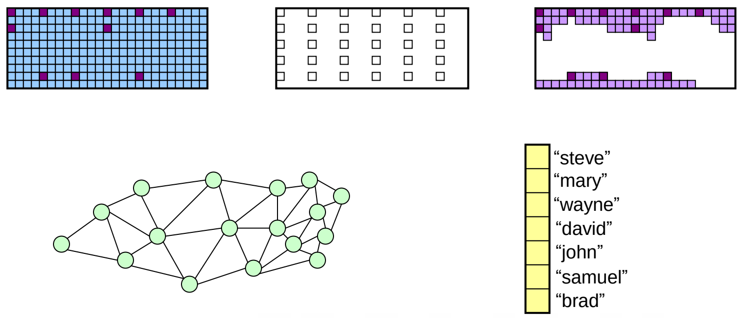


Data Parallelism: Domain Iteration



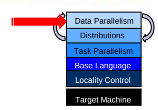
…to iterate over index spaces…

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



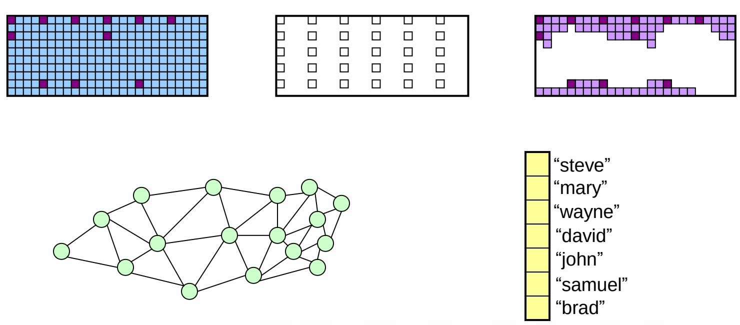
Tutorial M04: Chapel (39)

Data Parallelism: Array Slicing



…to slice arrays…

```
DnsArr[StrDom] += SpsArr[StrDom];
```

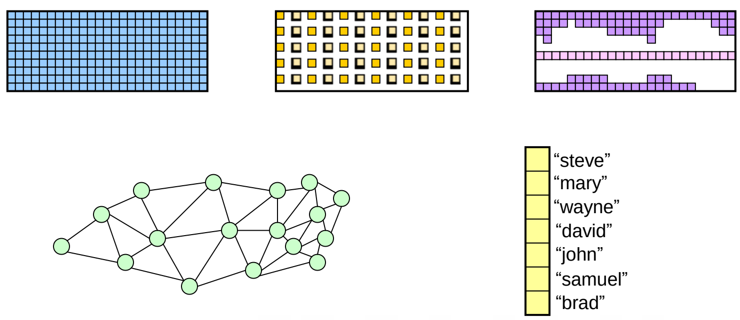


Tutorial M04: Chapel (40)

Data Parallelism: Array Reallocation

…and to reallocate arrays

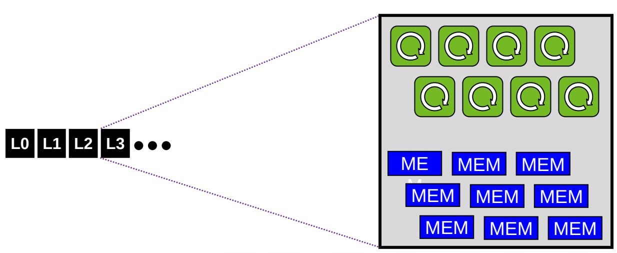
```
StrDom = DnsDom by (2,2);
SpsDom += genEquator();
```



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]);
```

0 1 2 3
4 5 6 7

```
var TaskALocs = Locales[..numTaskALocs];
var TaskBLocs = Locales[numTaskALocs+1..];
```

0 1

2 3 4 5 6 7

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

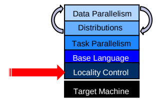
```
cobegin {
  on TaskALocs do computeTaskA(…);
  on TaskBLocs do computeTaskB(…);
  on Locales(0) do computeTaskC(…);
}
```

0 1 computeTaskA()

2 3 4 5 6 7 computeTaskB()

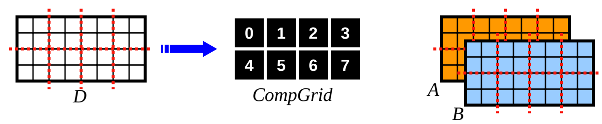
0 computeTaskC()

Locality: Domain Distribution



Domains may be distributed across locales

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



A distribution implies…

…ownership of the domain's indices (and its arrays' elements)

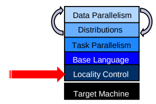
…the default work ownership for operations on the domains/arrays

Chapel provides…

…a standard library of distributions (Block, Recursive Bisection, …)

…the means for advanced users to author their own distributions

Locality: Domain Distributions

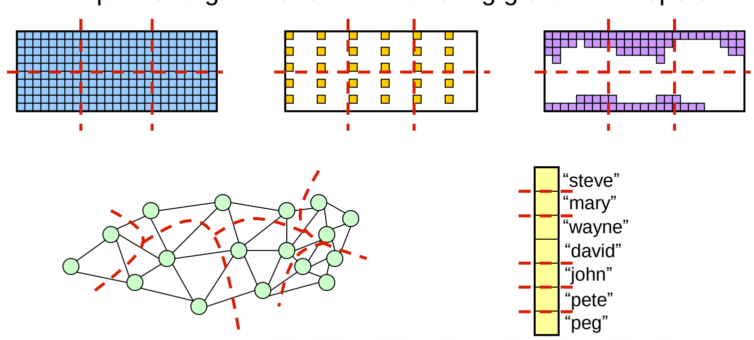


A distribution must implement…

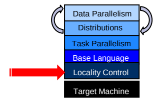
…the mapping from indices to locales

…the per-locale representation of domain indices and array elements

…the compiler's target interface for lowering global-view operations

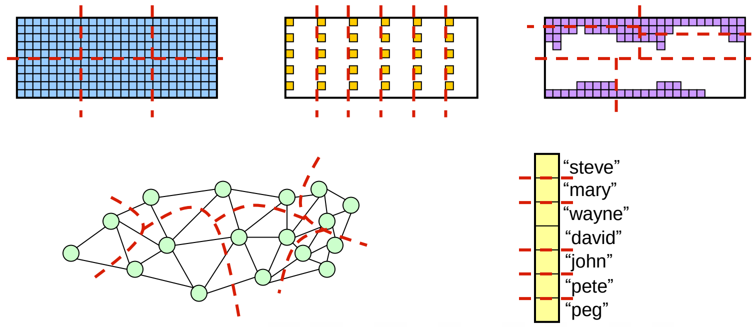


Locality: Domain Distributions

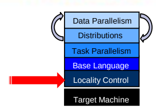


A distribution must implement…

…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



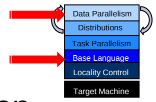
Locality: Distributions Overview



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



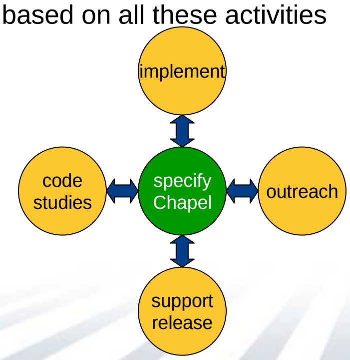
- zippered and tensor flavors of iteration and promotion
- *subdomains* and *index types* to help reason about indices
- reductions and scans (standard or user-defined operators)

Outline

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

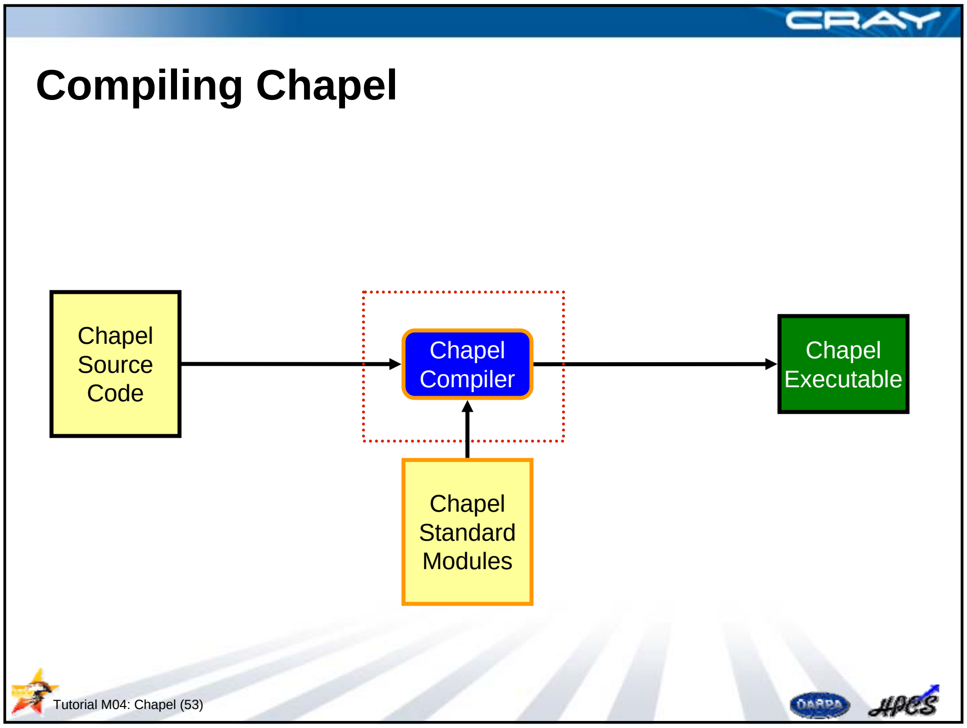
Chapel Work

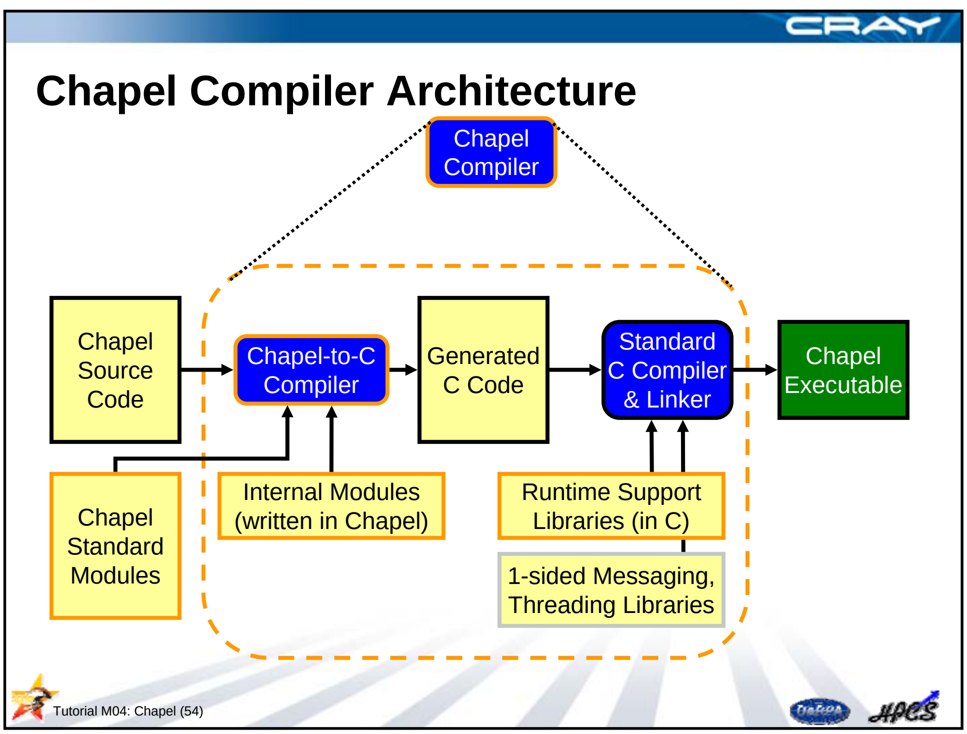
- Chapel Team's Focus:
  - specify Chapel syntax and semantics
  - implement open-source prototype compiler for Chapel
  - perform code studies of benchmarks, apps, and libraries in Chapel
  - do community outreach to inform and learn from users/researchers
  - support users of code releases
  - refine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


CRAY
Compiling Chapel
Chapel Source Code
Chapel Compiler
Chapel Executable
Chapel Standard Modules
Tutorial M04: Chapel (53)
HPCS


CRAY
Chapel Compiler Architecture
Chapel Compiler
Chapel Source Code
Chapel-to-C Compiler
Generated C Code
Standard C Compiler & Linker
Chapel Executable
Chapel Standard Modules
Internal Modules (written in Chapel)
Runtime Support Libraries (in C)
1-sided Messaging, Threading Libraries
Tutorial M04: Chapel (54)
HPCS

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

(Your name here?)

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…)

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

Tutorial M04: Chapel (61)

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

Tutorial M04: Chapel (62)

Chapel Team

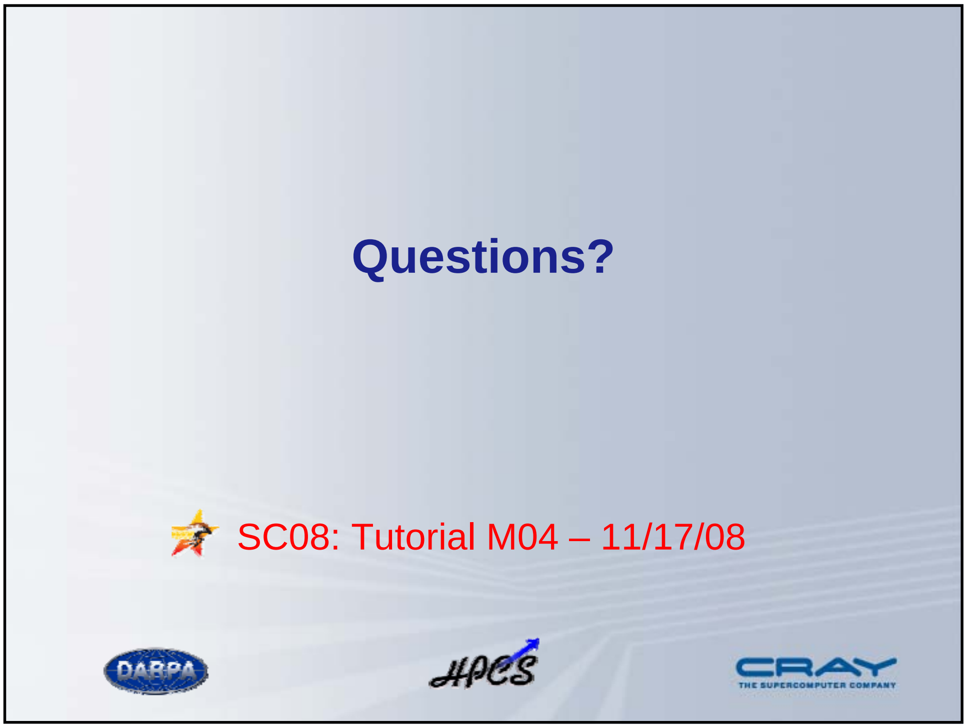
- Current Team
  - Brad Chamberlain 
  - Samuel Figueroa 
  - Steve Deitz 
  - David Iten 
- Interns
  - Robert Bocchino (`06 – UIUC)
  - James Dinan (`07 – Ohio State)
  - Mackale Joyner (`05 – Rice)
  - Andy Stone (`08 – Colorado St)
- Alumni
  - David Callahan
  - Roxana Diaconescu
  - Shannon Hoffswell
  - Mary Beth Hribar
  - Mark James
  - John Plevyak
  - Wayne Wong
  - Hans Zima

For More Information

chapel_info@cray.com

http://chapel.cs.washington.edu

Parallel Programmability and the Chapel Language; Chamberlain, Callahan, Zima; International Journal of High Performance Computing Applications, August 2007, 21(3):291-312.


Questions?
SC08: Tutorial M04 – 11/17/08
DARPA
HPCS
CRAY
THE SUPERCOMPUTER COMPANY

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)

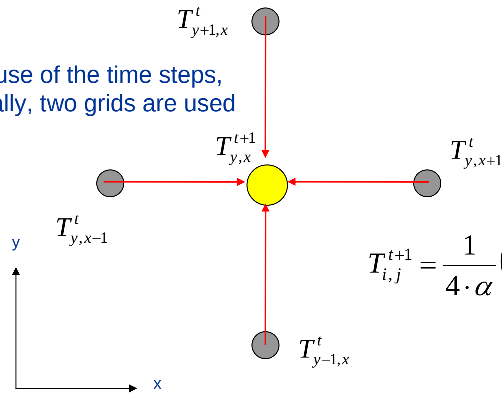
Comparison of Languages

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):**

Because of the time steps, Typically, two grids are used

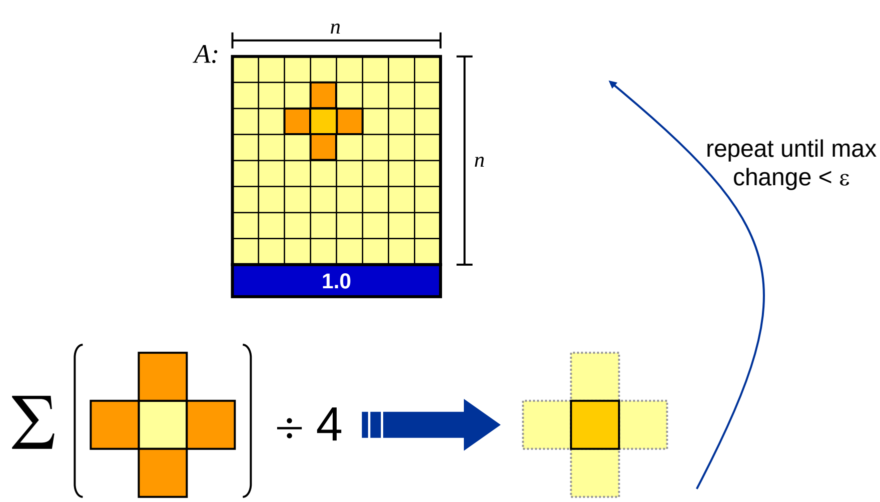


$$\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} = \frac{1}{\alpha}\frac{\partial T}{\partial t} \quad (1)$$

$$T^{t+1}_{i,j} = \frac{1}{4\cdot\alpha}\left(T^{t}_{i-1,j} + T^{t}_{i+1,j} + T^{t}_{i,j-1} + T^{t}_{i,j+1}\right) \quad (2)$$

3

Heat Transfer in Pictures



4

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);
     }
   }
```

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

4-point Stencil

5

2D Heat Conduction Problem

```
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

```
 if( dTmax < epsilon )
  finished = 1;
 else{
/*swapping the source &
  destination “pointers”*/
  dg = sg;
  sg = 1-sg;
 }
  nr_iter++;
} while( !finished );
upc_barrier;
```

6

Comparison of Languages

X10

Heat transfer in X10

- X10 permits smooth variation between multiple concurrency styles
  - "High-level" ZPL-style (operations on global arrays)
    - Chapel "global view" style
    - Expressible, but relies on "compiler magic" for performance
  - OpenMP style
    - Chunking within a single place
  - MPI-style
    - SPMD computation with explicit all-to-all reduction
    - Uses clocks
  - "OpenMP within MPI" style
    - For hierarchical parallelism
    - Fairly easy to derive from ZPL-style program.

8

Heat Transfer in X10 – ZPL style

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

  const BigD = Dist.makeBlock([0..n+1, 0..n+1]),
        D = BigD | [1..n, 1..n],
        LastRow = [0..0, 1..n] to Region;
  val A=Array.make[Real](BigD), 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);
  }}
```

Type declaration

Block distribution

Instance initializer

Operation on global arrays

9

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

10

Heat Transfer in X10 -- II

```
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);
  }}
```

- **Flat parallelism: Assume one activity per place is desired.**
- **D.places() returns ValRail of places in D.**
- **D(z) returns sub-region of D at place z.**

Explicit Loop Chunking

Heat Transfer in X10 -- III

```
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);
  }}
```

- **Hierarchical parallelism: P activities at place z.**
  - **Easy to change above code so P can vary with z.**
- **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 with Hierarchical Parallelism

Heat Transfer in X10 -- IV

```
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);
      ateach (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);
     }}
```

One activity per place == MPI task

Akin to UPC barrier

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

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

13

Heat Transfer in X10 -- V

```
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);
      ateach (z in D.places()) clocked(c)
        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"

14

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);
```

17

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[Las

do {
  [(i

  con
  A[D
} whi

writeln(A);
```

Declare program parameters

const ⇒ can't change values after initialization

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

note that no types are given; inferred from initializer
n ⇒ **integer** (current default, 32 bits)
epsilon ⇒ **floating-point** (current default, 64 bits)

18

Heat Transfer in Chapel

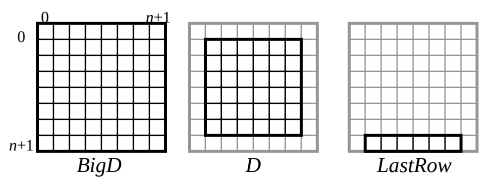
```
config const n = 6,
             epsilon = 1.0e-5;

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

Declare domains (first class index sets)

domain(2) ⇒ 2D arithmetic domain, indices are integer 2-tuples

subdomain(*P*) ⇒ a domain of the same type as *P* whose indices are guaranteed to be a subset of *P*'s

4;



exterior ⇒ one of several built-in domain generators

19

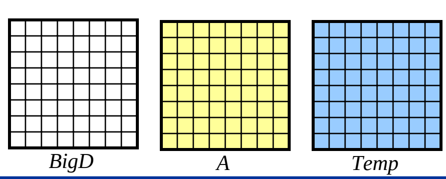
Heat Transfer in Chapel

```
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;
```

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)

4;



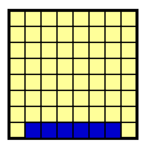
20

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;
```

Set Explicit Boundary Condition

indexing by domain ⇒ slicing mechanism
array expressions ⇒ parallel evaluation

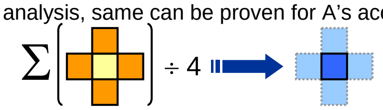


Heat Transfer in Chapel

Compute 5-point stencil

[(*i,j*) in *D*] ⇒ parallel forall expression over *D*'s indices, binding them to new variables *i* and *j*

Note: since (*i,j*) ∈ *D* and *D* ⊆ *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) 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],
```

Compute maximum change

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

Promotion: *abs()* and – are scalar operators, automatically promoted to work with array operands

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

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

writeln(A);
```

23

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[La
```

Copy data back & Repeat until done

uses slicing and whole array assignment
standard *do…while* loop construct

```
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);
```

24

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 {
  [(                                          j)
                                              1)) / 4;

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

writeln(A);
```

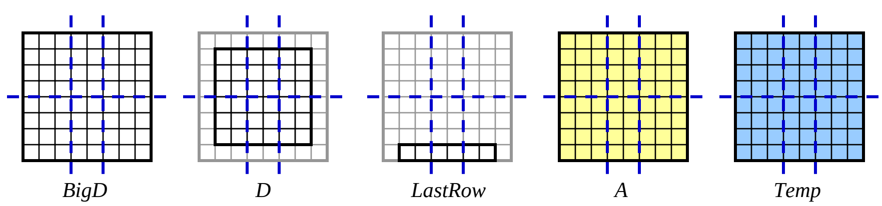
Write array to console

If written to a file, parallel I/O would be used

Heat Transfer in Chapel

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

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

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

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



Heat Transfer in Chapel

```
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);
```

27

Comparison of Languages

Comparative Feature Matrix

Features Matrix

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

29

Features Matrix

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

30

Features Matrix

	UPC	X10	Chapel
Dynamic Memory Allocation	Private or shared with or without blocking	Supports objects and arrays.	No pointers -- all dynamic allocations are through allocating new objects & resizing arrays
Synchronization	Barriers, split phase barrier, locks, and memory consistency control	Conditional atomic blocks, dynamic barriers (clocks)	Synchronization and single variables; transactional memory-style atomic blocks
Type Conversion	C rules Casting of shared pointers to private pointers	Coercions, conversions supported as in OO languages	C#-style rules
Pointers To Shared Space	Yes	Yes	Yes
global-view distributed arrays	Yes, but 1D only	Yes	Yes

31

Partial Construct Comparison

Constructs	UPC	X10	Chapel
Concurrency spawn	upc_forall	async,future, foreach, ateach	begin, cobegin, forall, coforall
Termination detection	finish	sync	N/A
Distribution construct	affinity in upc_forall, blocksize in work distribution	places, regions, distributions	locales, domains, distributions
Atomicity control	N/A	Basic atomic blocks	TM-based atomic blocks
Data-flow synchronization	N/A	Conditional atomic blocks	single variables
Barriers	upc_barrier	clocks	sync variable

32

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

33

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

34

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

35

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)**

37

D. Hands-On

Backup

Heat Transfer in Chapel
(Backup Variations)

40

Heat Transfer in Chapel (double buffered 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);

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);
```

41

Heat Transfer in Chapel (named direction 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);

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);
```

42

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);
```

43

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);
```

44

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 {
  [(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);
```

45