CSEP 524: Parallel Computation
(week 8)

Brad Chamberlain

Tuesdays 6:30 – 9:20

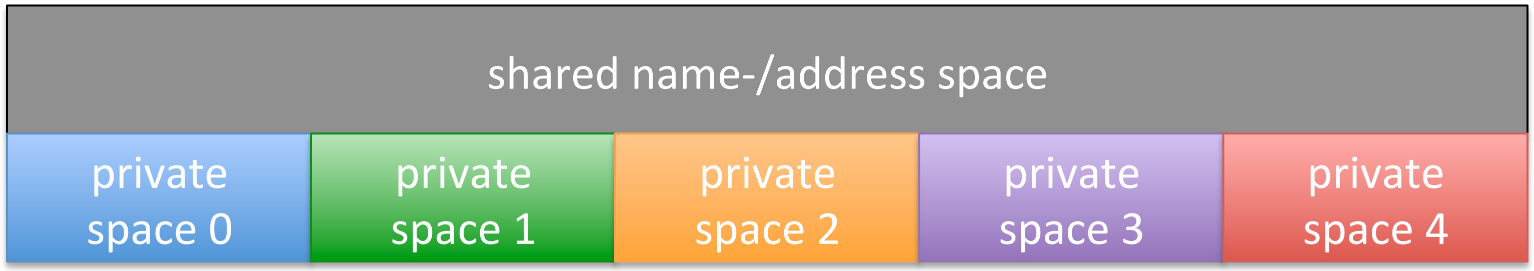
MGH 231

Partitioned Global Address Space (PGAS) Programming Models

Partitioned Global Address Space Languages

(Or perhaps: partitioned global namespace languages)

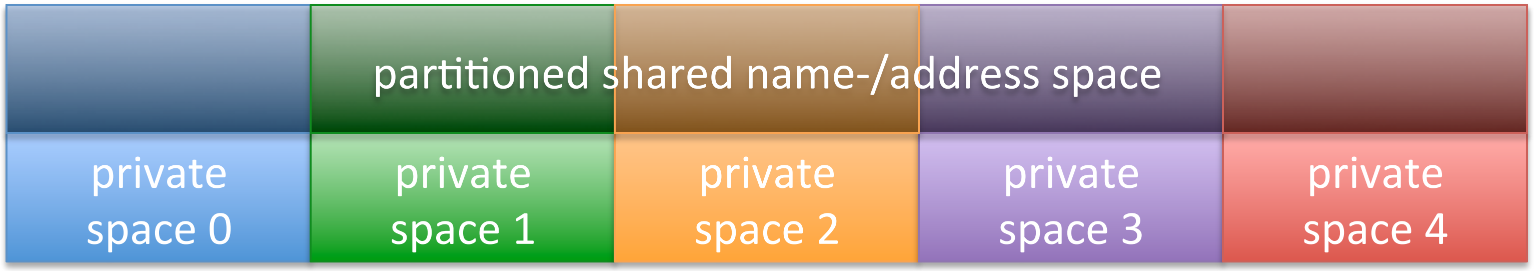
- abstract concept:
  - support a shared namespace on distributed memory
    - permit any parallel task to access any lexically visible variable
    - doesn't matter if it's local or remote
  - establish a strong sense of ownership
    - every variable has a well-defined location
    - local variables are cheaper to access than remote ones



Partitioned Global Address Space Languages

(Or perhaps: partitioned global namespace languages)

- abstract concept:
  - support a shared namespace on distributed memory
    - permit any parallel task to access any lexically visible variable
    - doesn't matter if it's local or remote
  - establish a strong sense of ownership
    - every variable has a well-defined location
    - local variables are cheaper to access than remote ones



Co-Array Fortran (CAF)

CAF: The first of our "traditional" PGAS languages

- developed ~1994
- adopted into the 2008 Fortran standard

Motivating Philosophy: "What is the smallest change required to convert Fortran 95 into a robust parallel language?"

- originally referred to as F-- to emphasize "smallest change"

Quick Fortran Review/Intro

- Traditional variables in Fortran:

```
integer i            ! declares an integer, i
real x               ! declares a float, x
real a(20)           ! declares a 20-element array
real b(N,N)          ! declares an N x N array
```

- Array accesses are written with parenthesis:

```
a(1) = x             ! Fortran uses 1-based indexing by default
b(1,1) = 2*x
b(2,:) = 3*x         ! assign 3*x to the second row of b
                     ! (':' is like '..' in Chapel)
```

CAF is SPMD

- SPMD programming/execution model
  - similar to MPI* in this regard
  - program copies are referred to as 'images'

- Use intrinsic functions to query the basics:

```
integer :: p, me
p = num_images()   ! returns number of processes
me = this_image()  ! returns value in 1..num_images()
```

- Barrier sync:

```
sync_all()          ! wait for all processes/images
```

*= typical uses of it, anyway

Main CAF Concept: Co-Dimensions

Co-Dimension: an array dimension that refers to the space of CAF *images* (processes)

- defined using square brackets
  - (distinguishes it syntactically from a traditional dimension)

Main CAF Concept: Co-Dimensions

- Co-array variables in Fortran:

```
integer i[*]        ! declares an integer, i, per image
real x[*]           ! declares a float, x, per image
real a(20)[*]       ! declares a 20-element array per image
real b(N,N)[*]      ! declares an N x N array per image
```

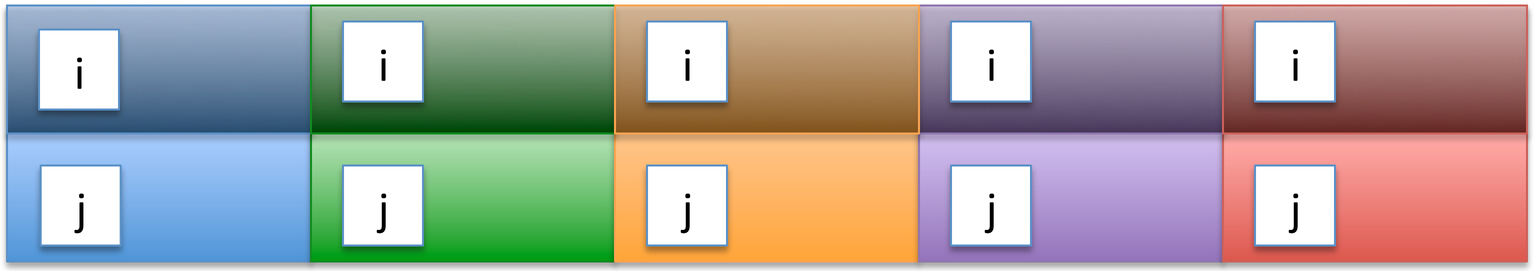
Main CAF Concept: Co-Dimensions

- Co-array variables in Fortran:

```
integer i[*]        ! declares an integer, i, per image
```

- Of course, traditional variables also result in a copy per image (it's SPMD after all), but *private* to that image

```
integer j           ! declares a private integer, j, per image
```



Using Co-Arrays

```
integer i[*]
real x[*]
```

- Refer to other images’ values via co-array indexing:

```
if (me == 2)  then
   nextX = x[me+1]     ! read neighbor’s value of x
   i[1] = i            ! copy my value of ‘i’ into image 1’s
endif
```

- Co-array indexing/square brackets ⇒ communication

Stylized Collective Communications in CAF

Given declarations:

```
real x[*]    real y    real a(num_images())
```

Broadcast:

```
x[:] = y
```

Reduction:

```
y = MINVAL(x[:])
```

Gather:

```
a(:) = x[:]
```

Scatter:

```
x[:] = a(:)
```

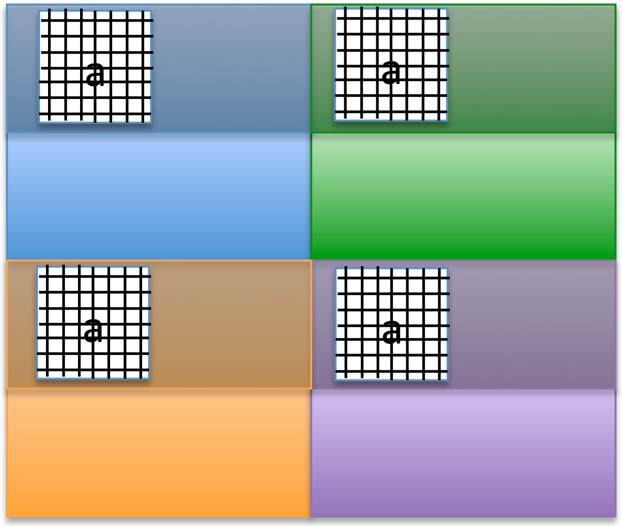
Distributed Arrays in CAF

- When things divide evenly, you're pretty happy:
  - e.g., 1000 x 1000 array on a 2 x 2 processor grid:

    ```
    real a(500,500)[2,2]
    ```

  - or, adding in additional space for stencil ghost cells:

    ```
    real a(0:501, 0:501)[2,2]
    ```



Distributed Arrays in CAF

- When things divide evenly, you're pretty happy:
  - e.g., 1000 x 1000 array on a 2 x 2 processor grid:

    ```
    real a(500,500)[2,2]
    ```

  - or, adding in additional space for stencil ghost cells:

    ```
    real a(0:501, 0:501)[2,2]
    ```

- Stencil-style boundary value communication idioms:

```
! compute myrow, mycol, numrows, numcols
if (myrow .ne. 1) then
  a(0,:) = a(500,:)[myrow-1,mycol]
endif
if (myrow .ne. numrows) then
  a(501,:) = a(1,:)[myrow+1, mycol]
endif
! etc.
```

Distributed Arrays in CAF

- When they don't, more work is required…
  - e.g., 1000 x 1000 array on a 2 x 3 processor grid:

    ```
    real a(500,334)[2,3]    ! allocate ceil(n/p) everywhere
    ```

    …and then the images have to do bookkeeping to keep track of which image(s) own 334 items and which own 333

  - details start to resemble the 9-point MPI code from HW
    - e.g., global-to-local and local-to-global index transformations
    - also, due to PGAS model, need to know more about neighbors
      - **MPI:** "I'll send you my high column which has index 333!"; "I'll recv it!"
      - **CAF:** "I'm going to access your high column" ⇒ "I must know its index"
    - (of course, some of this applies when things divide evenly as well…)

CAF Summary

- Program in SPMD style
- Communicate via variables with co-dimensions
  - a copy per program image
  - refer to other images' copies via square bracket subscripts
  - take advantage of good multidimensional array support
    - multidimensional views of process grid
    - multidimensional views of local data
    - syntactic support for slicing (:)
- Other stuff too, but this gives you the main idea
- Adopted into Fortran 2008 standard
  - see also http://www.co-array.org

CAF 2.0 (Rice University)

Motivation: Respond to a lack of richness in CAF

- difficult to have sets of images doing distinct things (teams)
- no support for pointer-based data structures
- poor support for collectives

For more information:

http://caf.rice.edu

UPC: Unified Parallel C

UPC: Our second "traditional" PGAS language

- developed ~1999
- "unified" in the sense that it combined 3 distinct parallel Cs:
  - AC, Split-C, Parallel C Preprocessor
- though a sibling to CAF, philosophically quite different

Motivating Philosophy:

- extend C concepts logically to support SPMD execution
  - 1D arrays
  - for loops
  - pointers (and pointer/array equivalence)

UPC is also SPMD

- SPMD programming/execution model
  - program copies are referred to as ‘threads’

- Built-in constants provide the basics:

```
int p, me;
p = THREADS;     // returns number of processes
me = MYTHREAD;   // returns a value in 0..THREADS-1
```

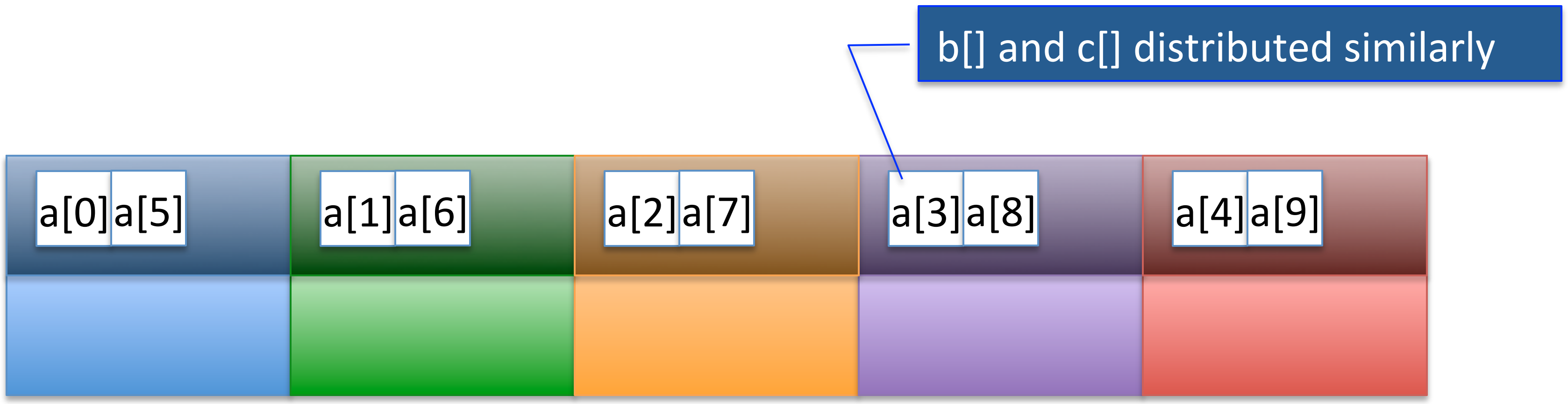
- Barrier synch statement:

```
upc_barrier;     // wait for all processes/threads
```

Distributed Arrays in UPC

- Arrays declared with the ‘shared’ keyword are distributed within the shared space
  - uses a cyclic distribution by default

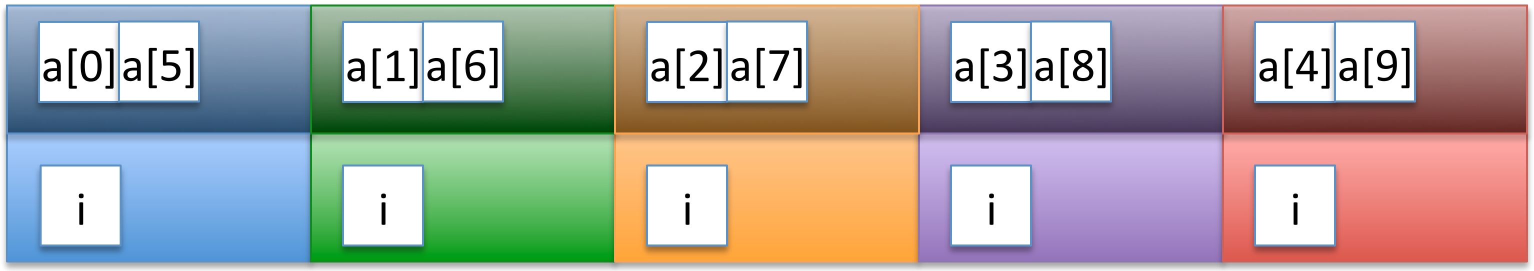
```
#define N 10
shared float a[N], b[N], c[N];
```



Distributed Arrays in UPC

- Arrays declared with the ‘shared’ keyword are distributed within the shared space
  - uses a cyclic distribution by default

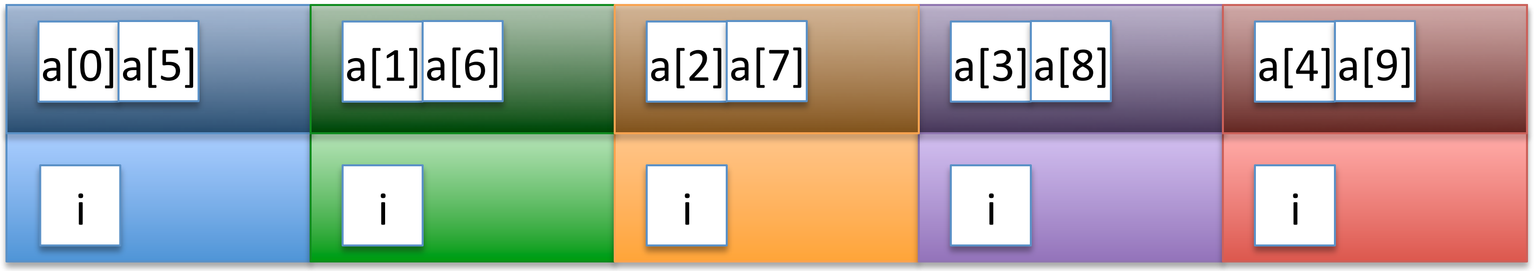
```
#define N 10
shared float a[N], b[N], c[N];
for (int i=0; i<N; i++) { // dumb loop: O(N)
  if (i%THREADS == MYTHREAD) {
    c[i] = a[i] + alpha * b[i];
} }
```



Distributed Arrays in UPC

- Arrays declared with the ‘shared’ keyword are distributed within the shared space
  - uses a cyclic distribution by default

```
#define N 10
shared float a[N], b[N], c[N];
// smarter loop: O(N/THREADS)
for (int i=MYTHREAD; i<N; i+=THREADS) {
  c[i] = a[i] + alpha * b[i];
}
```

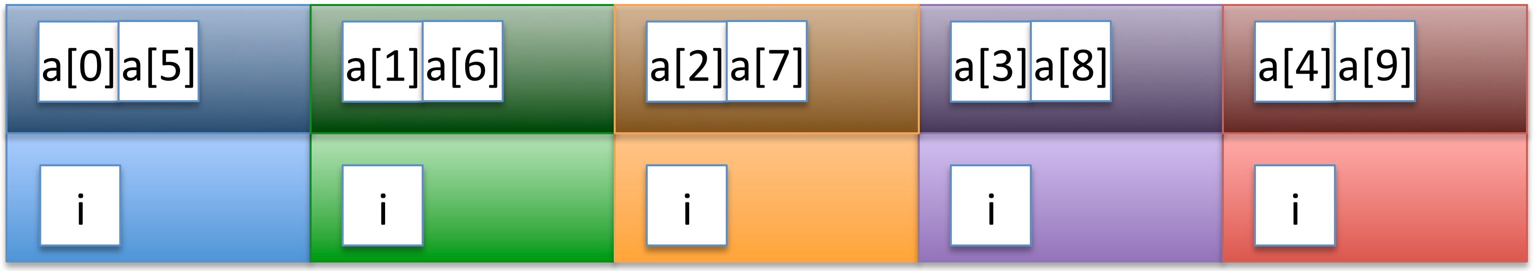


Distributed Arrays in UPC

- Arrays declared with the ‘shared’ keyword are distributed within the shared space
  - uses a cyclic distribution by default

```
#define N 10
shared float a[N], b[N], c[N];
// “global-view”equivalent to the previous
upc_forall (int i=0; i<N; i++; i) {
  c[i] = a[i] + alpha * b[i];
}
```

Affinity field: Which thread should execute this iteration? (if int, %THREADS to get ID)

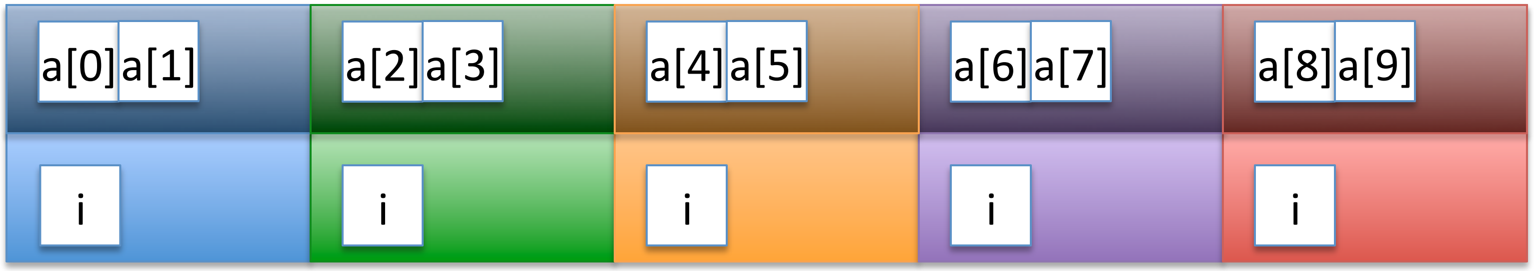


Distributed Arrays in UPC

- Arrays declared with the 'shared' keyword are distributed within the shared space
  - can specify a block-cyclic distribution as well

```
#define N 10
shared [2] float a[N], b[N], c[N];
upc_forall (int i=0; i<N; i++; &c[i]) {
  c[i] = a[i] + alpha * b[i];
}
```

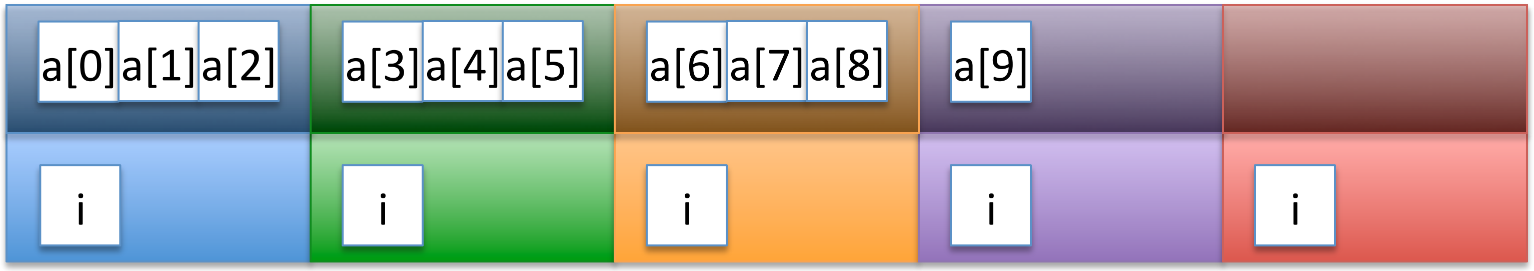
Affinity field: Which thread should execute this iteration? (if ptr-to-shared, owner does)



Distributed Arrays in UPC

- Arrays declared with the ‘shared’ keyword are distributed within the shared space
  - can specify a block-cyclic distribution as well

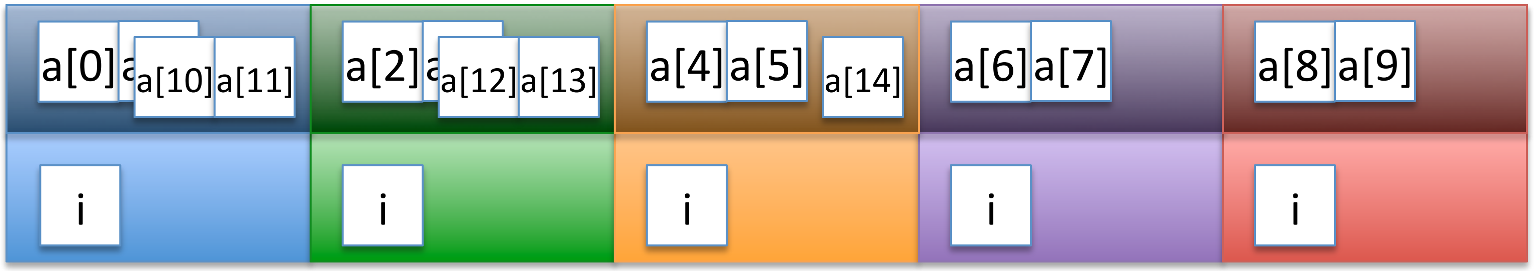
```
#define N 10
shared [3] float a[N], b[N], c[N];
upc_forall (int i=0; i<N; i++; &c[i]) {
  c[i] = a[i] + alpha * b[i];
}
```



Distributed Arrays in UPC

- Arrays declared with the 'shared' keyword are distributed within the shared space
  - can specify a block-cyclic distribution as well

```
#define N 15
shared [2] float a[N], b[N], c[N];
upc_forall (int i=0; i<N; i++; &c[i]) {
  c[i] = a[i] + alpha * b[i];
}
```

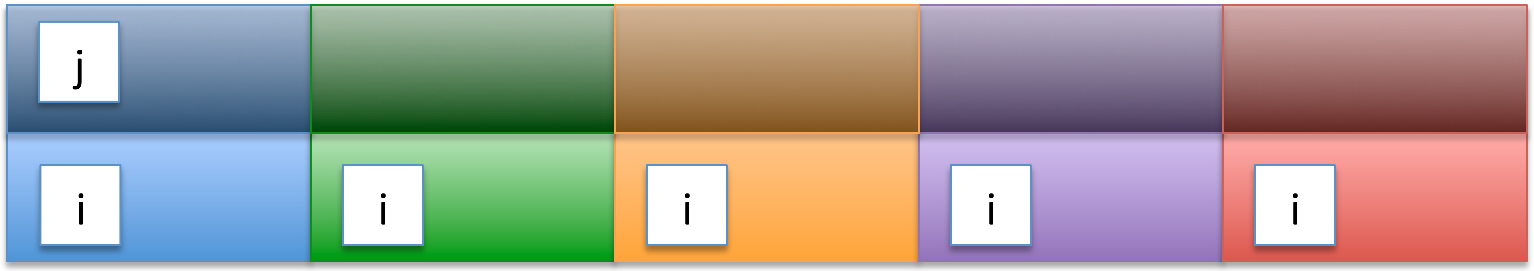


Scalars in UPC

- Somewhat confusingly (to me anyway*), shared scalars in UPC result in a single copy on thread 0

```
int i;
shared int j;
```

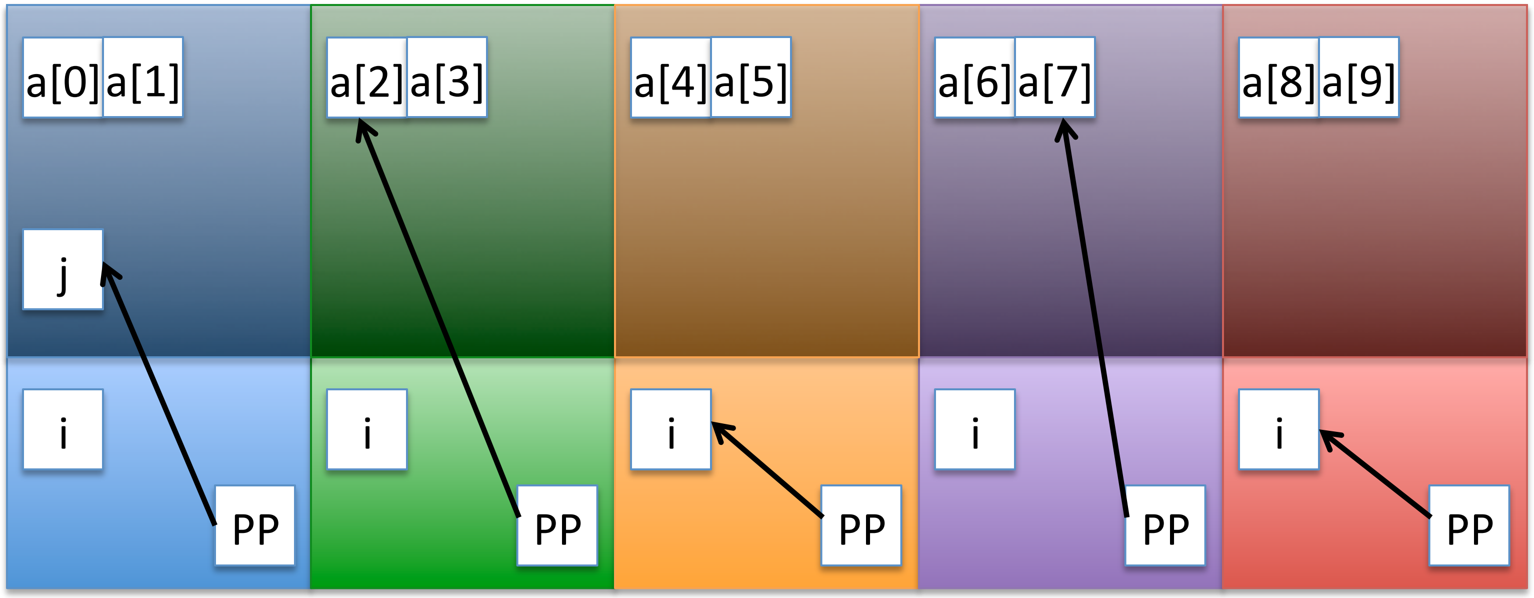
* = because it seems contrary to SPMD programming



Pointers in UPC

- UPC Pointers may be private/shared and may point to private/shared

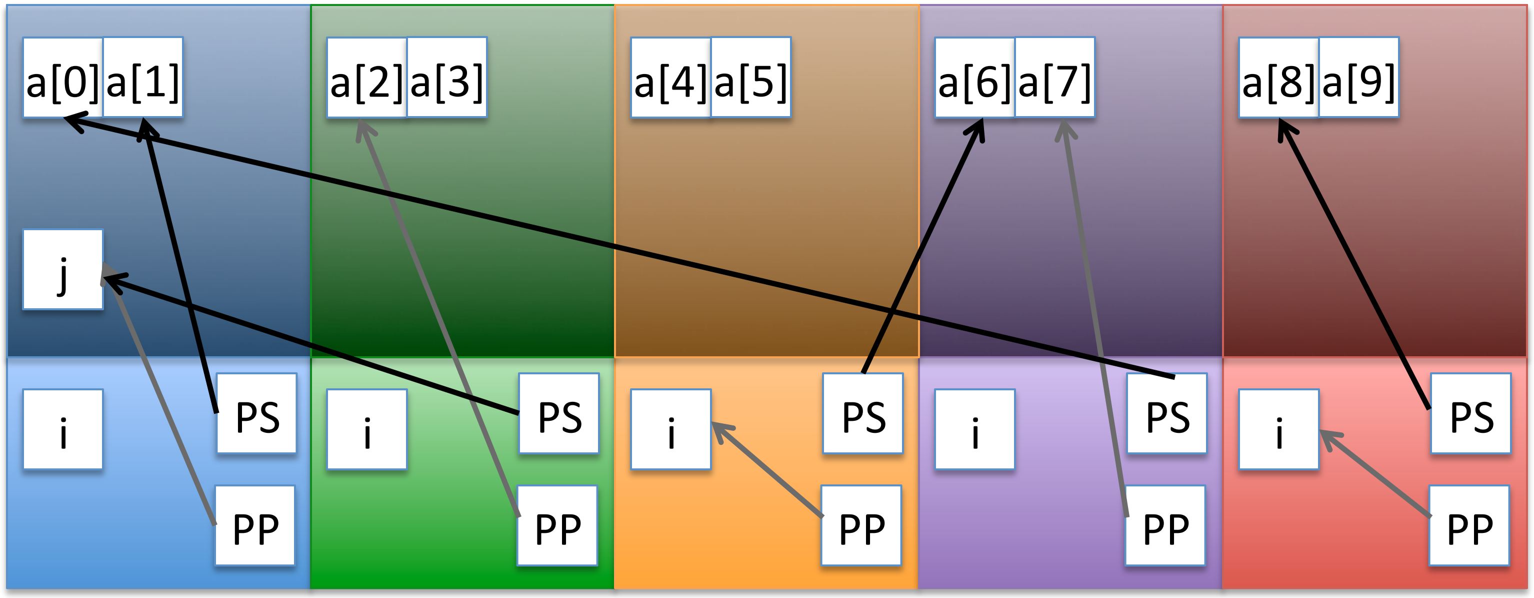
```
int* PP; // private pointer to local data
```



Pointers in UPC

- UPC Pointers may be private/shared and may point to private/shared

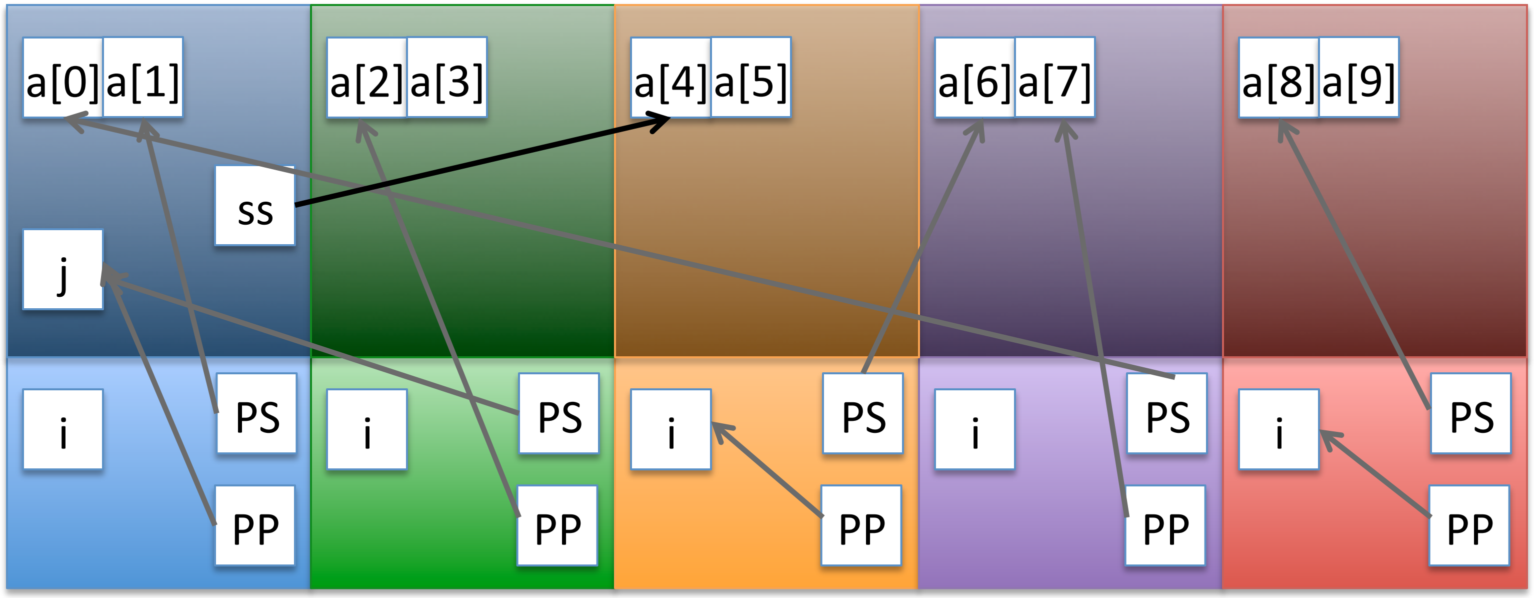
```
int* PP; // private pointer to local data
shared int* PS; // private pointer to shared data
```



Pointers in UPC

- UPC Pointers may be private/shared and may point to private/shared

```
int* PP;                   // private pointer to local data
shared int* PS;            // private pointer to shared data
shared int* shared ss;     // shared pointer to shared data
```



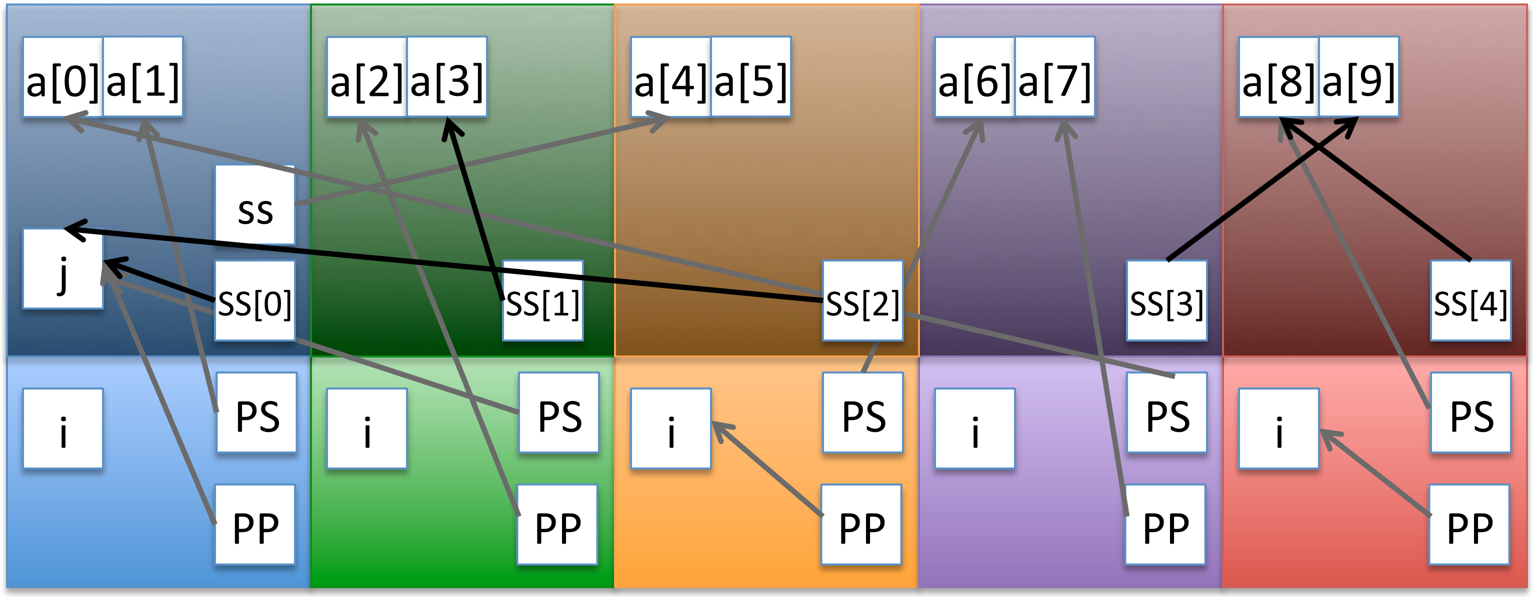
Arrays of Pointers in UPC

- Of course, one can also create arrays of pointers

  // array of shared pointer to shared data

```
shared int* shared SS[THREADS];
```

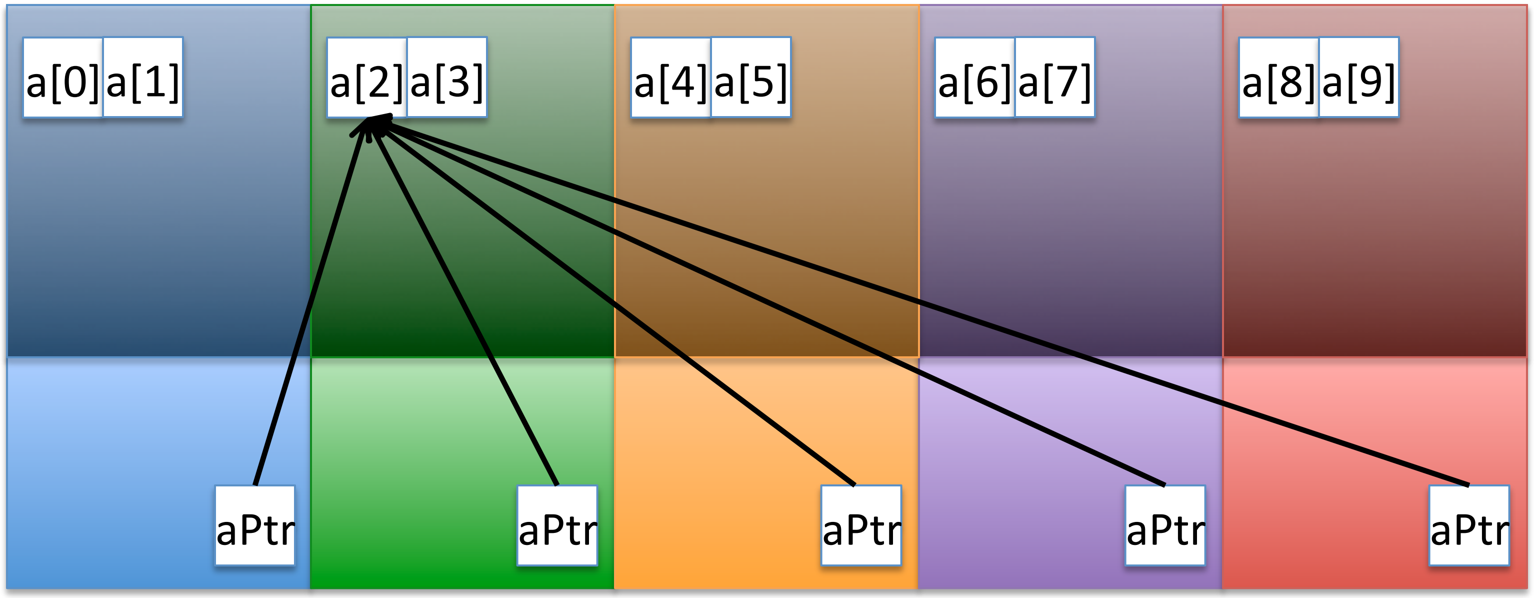
- As you can imagine, one UPC's strengths is its ability to create fairly arbitrary distributed data structures



Array/Pointer Equivalence in UPC

- As in C, pointers can be walked through memory

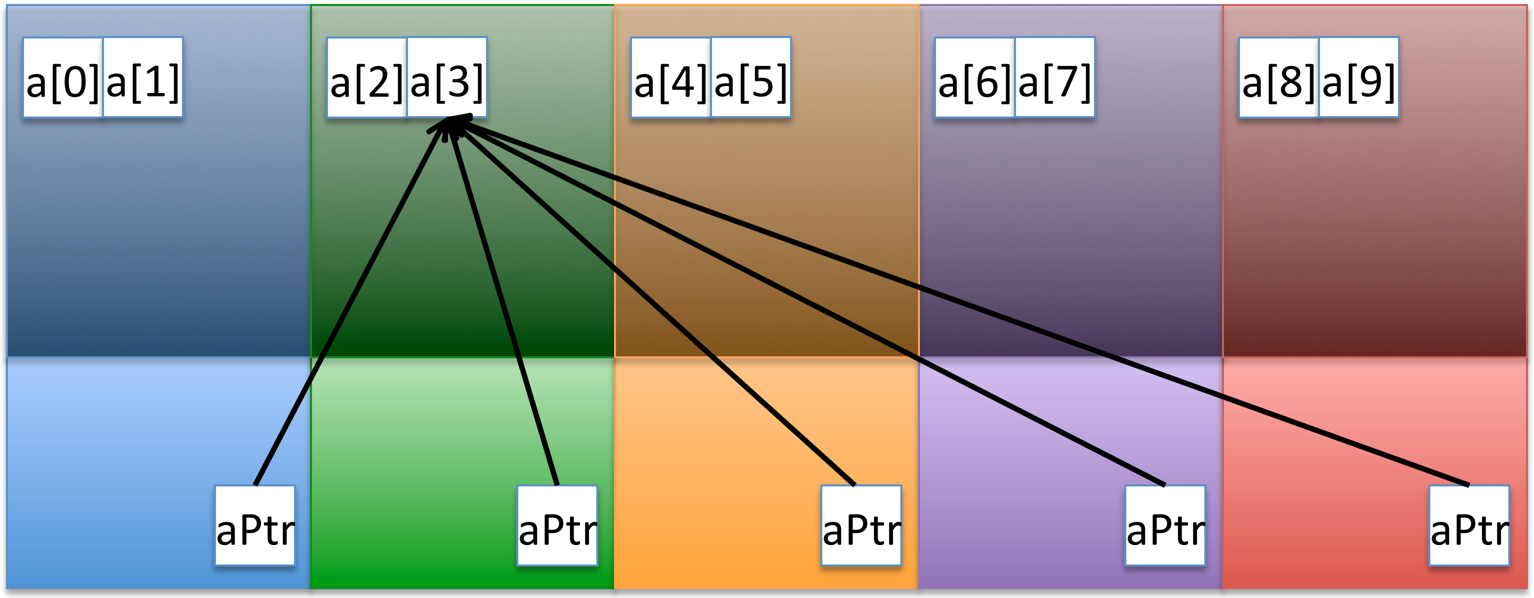
```
shared [2] float a[N];
shared [2] float* aPtr = &(a[2]);
```



Array/Pointer Equivalence in UPC

- As in C, pointers can be walked through memory

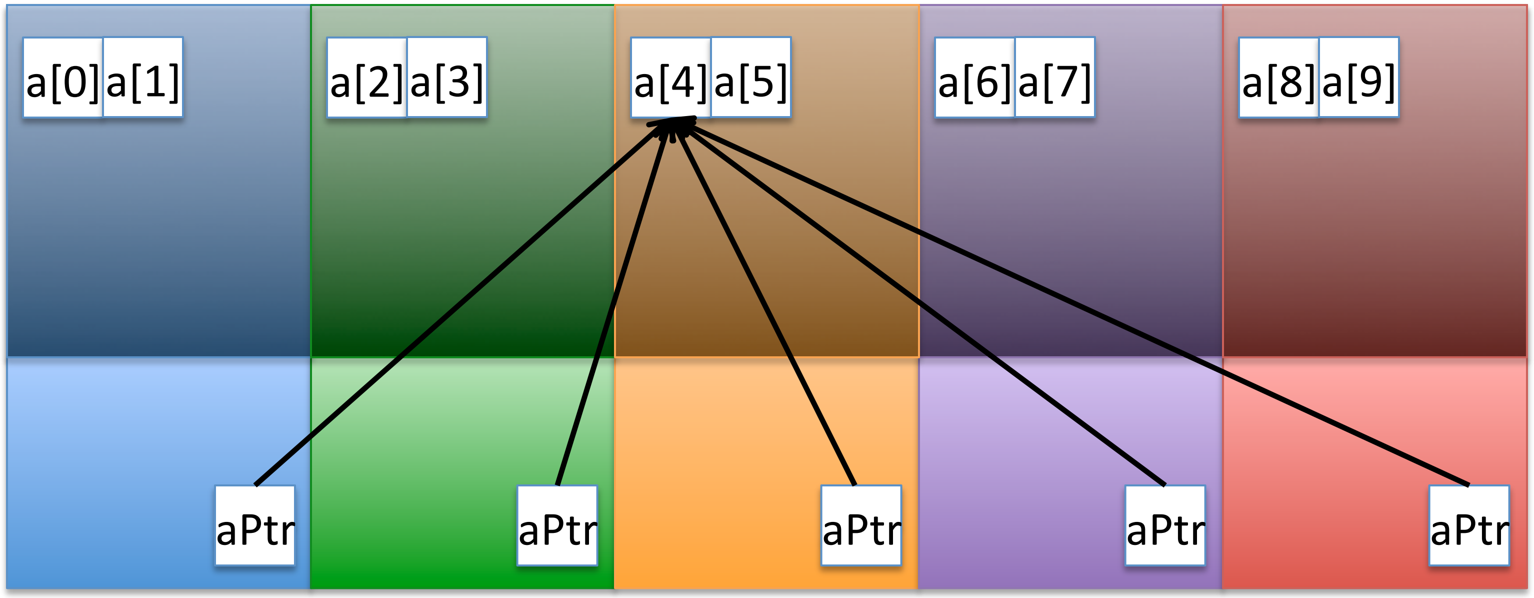
```
shared [2] float a[N];
shared [2] float* aPtr = &(a[2]);
aPtr++;
```



Array/Pointer Equivalence in UPC

- As in C, pointers can be walked through memory

```
shared [2] float a[N];
shared [2] float* aPtr = &(a[2]);
aPtr++;
aPtr++;
```



How are UPC Pointers Implemented?

Local pointers to local: just an address, as always

Pointers to shared: 3 parts

- thread ID
- base address of block within the thread
- phase/offset within the block (0..blocksize-1)

- UPC supports a number of utility functions that permit you to query this information from pointers

- Casting between pointer types is permitted
  - but can be dangerous (as in C) and/or lossy

UPC: Local-view or Global-view?

Global arrays and pointers: global-view

upc_forall loops: global-view

Shared scalars: global-view-ish (but constrained)

Private scalars: local-view

SPMD model: local-view

⇒ a bit of both

Other Features in UPC

- Collectives Library
- Memory Consistency Model
  - among the first/foremost memory models in HPC
  - ability to move between strict and relaxed models
  - fence operations
- Dynamic Memory Management
- Locks
- Parallel I/O
- …

Titanium: Java-based PGAS language

Titanium: The third traditional PGAS language
- And in my opinion, the most promising in terms of features
- Based on Java, though loosely at times

- Unfortunately didn't catch on as well
  - in part because Java not dominant in HPC
  - in part because of "superset of subset" problem
    - it's like Java except for when it's completely different
- Last I heard, "not quite dead yet"

PGAS: What's in a Name?

		memory model	*programming model*	*execution model*	*data structures*	*communication*
	MPI	distributed memory	cooperating executables (often SPMD in practice)		manually fragmented	APIs
	OpenMP	shared memory	global-view parallelism	shared memory multithreaded	shared memory arrays	N/A
PGAS Languages	CAF	PGAS	Single Program, Multiple Data (SPMD)		co-arrays	co-array refs
	UPC				1D block-cyc arrays/ distributed pointers	implicit
	Titanium				class-based arrays/ distributed pointers	method-based
	Chapel	PGAS	global-view parallelism	distributed memory multithreaded	global-view distributed arrays	implicit

Chapel and PGAS

- Chapel differs from UPC/CAF since it's not SPMD

⇒ "global name-/address space" comes from lexical scoping

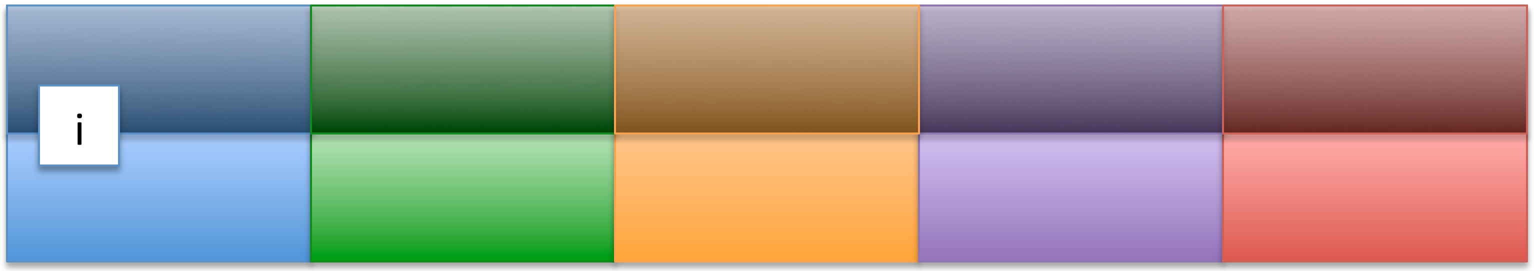
  - rather than: "We're all running the same program, so we must all have a variable named *x*"
  - as in traditional languages, each declaration yields one variable
  - stored on locale where task executes, not everywhere/thread 0

⇒ user-level concept of locality is central to language

  - parallelism and locality are two distinct things
  - shouldn't think in terms of "that other copy of the program"

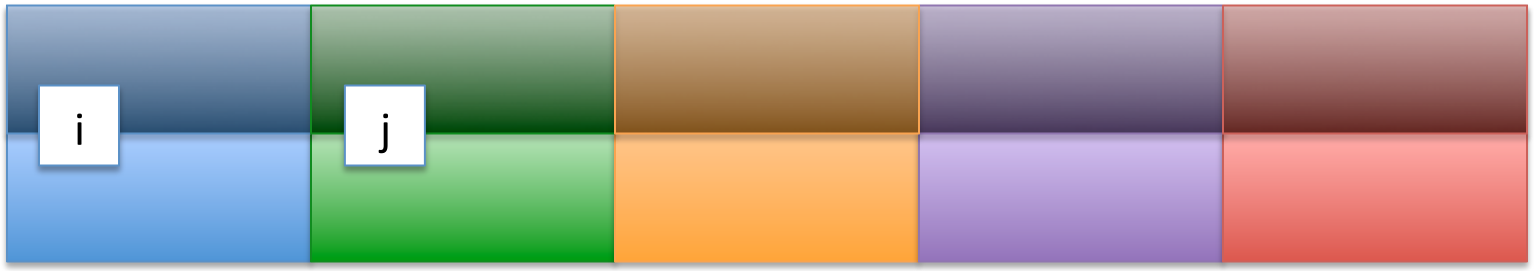
Chapel and PGAS

```
var i: int;
```



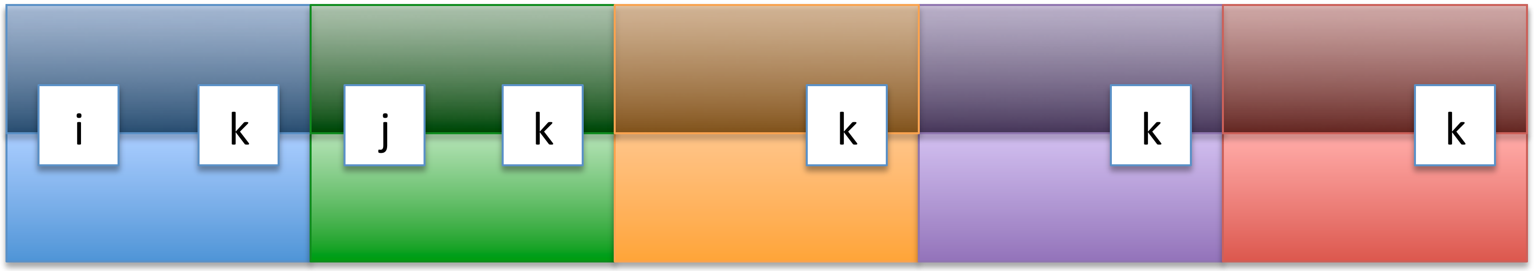
Chapel and PGAS

```
var i: int;
on Locales[1] {
  var j: int;
```



Chapel and PGAS

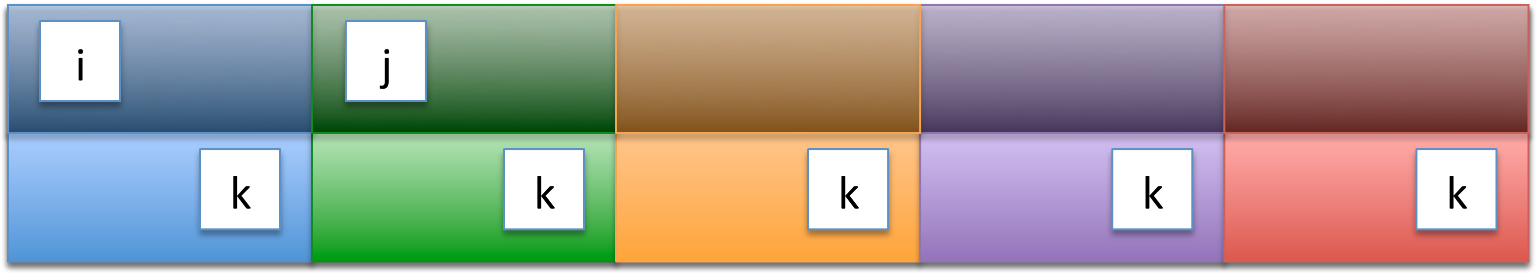
```
var i: int;
on Locales[1] {
  var j: int;
  coforall loc in Locales {
    on loc {
      var k: int;
    }
  }
}
```



Chapel and PGAS: Public vs. Private

- How public a variable is depends only on scoping
  - who can see it?
  - who actually bothers to refer to it?

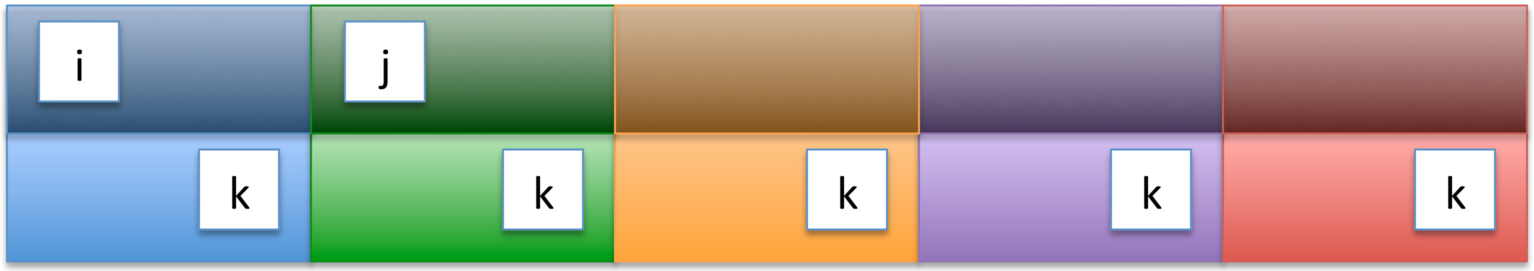
```
var i: int;
on Locales[1] {
  var j: int;
  coforall loc in Locales {
    on loc {
      var k = i + j;
    }
```



```
}
```

Chapel and PGAS: Public vs. Private

- How public a variable is depends only on scoping
  - who can see it?
  - who actually bothers to refer to it?
- Chapel represents variables that are referred to non-locally using *wide pointers*
  - locale ID + local address
  - note: no need for phase/offset as in UPC
    - because no block-cyclic pointer math required



Single-Sided Communication

But First: Two-Sided Communication

two-sided communication: What we did in MPI

- one process *sends* a message
- another process *receives*
- both sides necessary for data to be transferred
  - else, deadlock

Implementing PGAS Languages: 1-sided comm.

single-sided (one-sided) communication: the backbone of most PGAS language implementations

primitive operations:

- ***get():*** reads from a remote process's address space
- ***put():*** writes to a remote process's address space

- No matching operation required!

Prototypical 1-sided comm. routines

```
void get(void* localAddr,      // local destination
         int remoteProcID,     // remote process/image
         void* remoteAddr,     // remote source address
         int numBytes);        // amount of data

void put(void* remoteAddr,     // remote destination
         int remoteProcID,     // remote process/image
         void* localAddr,      // local source address
         int numBytes);        // amount of data
```

(Many implementations will also support variations for strided puts/gets, multidimensional puts/gets, gather/scatter puts/gets)

Why does PGAS need/want 1-sided comm?

- Communication is expressed via naming variables that happen to live on another process
  - generally, one process will have no idea what other is doing
  - even in SPMD programming models
    - control flow may take different paths
    - local/private variables are likely to have different values
  - as a result, I can’t guess what data of mine you might need
    - so I can’t call the matching sends/recvs to fulfill your requests

Summary of 1-sided comm.

- Characteristics:
  - notably, the text of the remote program need do nothing
  - in effect, implements load/store for non-trivial data sizes over distributed memory
  - interestingly, has not become an end-user model like MPI
  - key supporting network technology to work well: *RDMA*
    - Remote Direct Memory Access
- Benefits:
  - results in fewer copies/buffers within the SW stack (often 0)
  - separates data transfer from synchronization of processes
  - with RDMA, doesn't require remote CPU to be involved

Summary of 1-sided comm.

- Drawbacks:
  - if network has no RDMA support, performance can suffer
    - e.g., may require devoting a thread to handling incoming requests
    - (in particular, 1-sided comm. can be implemented using MPI)
  - re-opens door to memory consistency issues

1-Sided Communication Implementations

SHMEM/OpenSHMEM (Cray/community)
- the first (? major, anyway) single-sided comm. interface

GASNet (Berkeley)
- (what Chapel uses by default)

ARMCI (PNNL)

GASPI (Germany)

MPI-3
- as mentioned last week, part of newest feature set

Chapel's Extra Communication Requirement

In addition to puts/gets Chapel needs *active messages*

- "run this code over there with these arguments"
- can think of as a style of 1-sided communication
- *active* ⇒ control is transferred, not just data

Used to implement on-clauses

```
var i: int;
on Locales[1] {
  …  // send an active message to execute this code
}
```

Conceptual active message interface

```
void am(int remoteProcID,     // remote process/image
        int routineID,        // ID of function to exec
        void* args,           // arguments to send
        int argLen);          // length of arguments
```

Active Message Support?

SHMEM/OpenSHMEM (Cray/community)

GASNet (Berkeley)

ARMCI (PNNL)

GASPI (Germany)

MPI-3

Smith-Waterman Algorithm for Sequence Alignment

Smith-Waterman

Goal: Determine the similarities/differences between two protein sequences/nucleotides.

– e.g., ACACACTA and AGCACACA*

Basis of Computation: Defined via a recursive formula:

$$H(i,0) = 0$$

$$H(0,j) = 0$$

$$H(i,j) = f(H(i\text{-}1, j\text{-}1), H(i\text{-}1, j), H(i, j\text{-}1))$$

Caveat: This is a classic, rather than cutting-edge sequence alignment algorithm, but it illustrates an important parallel paradiagm: wavefront computation

*Source of running example: Wikipedia

Smith-Waterman

Naïve Task-Parallel Approach:

```
proc computeH(i,j) {
  if (i==0 || j == 0) then
    return 0;
  else
    var h1, h2, h3: int;

    begin h1 = computeH(i-1, j-1);
    begin h2 = computeH(i-1, j);
    begin h3 = computeH(i, j-1);

    return f(h1,h2,h3);
}
```

Note: Recomputes most subexpressions redundantly

This is a case for dynamic programming!

Smith-Waterman

Dynamic Programming Approach:

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

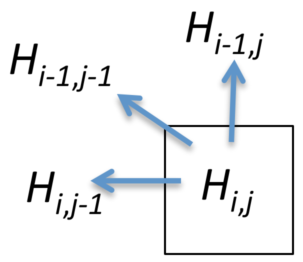
Step 1: Initialize boundaries to 0

Smith-Waterman

Dynamic Programming Approach:

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	🟥	🟧	🟨	🟩	🟦	🟪		
2	0	🟧	🟨	🟩	🟦	🟪			
3	0	🟨	🟩	🟦	🟪				
4	0	🟩	🟦	🟪	etc.				
5	0	🟦	🟪						
6	0	🟪							
7	0								
8	0								

Step 2: Compute cells as we're able to



Smith-Waterman

Dynamic Programming Approach:

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	2	1	2	1	2	1	0	2
2	0	1	1	1	1	1	1	0	1
3	0	0	3	2	3	2	3	2	1
4	0	2	2	5	4	5	4	3	4
5	0	1	4	4	7	6	7	6	5
6	0	2	3	6	6	9	8	7	8
7	0	1	4	5	8	8	11	10	9
8	0	2	3	6	7	10	10	10	12

Step 3: Follow trail of breadcrumbs back

Smith-Waterman

Dynamic Programming Approach:

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	2	1	2	1	2	1	0	2
2	0	1	1	1	1	1	1	0	1
3	0	0	3	2	3	2	3	2	1
4	0	2	2	5	4	5	4	3	4
5	0	1	4	4	7	6	7	6	5
6	0	2	3	6	6	9	8	7	8
7	0	1	4	5	8	8	11	10	9
8	0	2	3	6	7	10	10	10	12

Step 3: Follow trail of breadcrumbs back

Smith-Waterman

Dynamic Programming Approach:

Step 4: Interpret the path against the original sequences

		A	C	A	C	A	C	T	A
	0	0	0	0	0	0	0	0	0
A	0	2	1	2	1	2	1	0	2
G	0	1	1	1	1	1	1	0	1
C	0	0	3	2	3	2	3	2	1
A	0	2	2	5	4	5	4	3	4
C	0	1	4	4	7	6	7	6	5
A	0	2	3	6	6	9	8	7	8
C	0	1	4	5	8	8	11	10	9
A	0	2	3	6	7	10	10	10	12

```
AGCACAC-A
A-CACACTA
```

How could we do this in parallel?

Smith-Waterman

Data-Parallel Approach:

```
proc computeH(H: [0..n, 0..n] int) {
  for upperDiag in 1..n do
    forall diagPos in 0..#upperDiag {
      const (i,j) = [diagPos+1, upperDiag-diagPos];
      H[i,j] = f(H[i-1,j-1], H[i-1,j], H[i,j-1]);
    }
  for lowerDiag in 1..n-1 do
    forall diagPos in lowerDiag..n-1 by -1 {
      const (i,j) = [diagPos+1, lowerDiag+diagPos];
      H[i,j] = f(H[i-1,j-1], H[i-1,j], H[i,j-1]);
    }
}
```

Loop over upper diagonals serially

Traverse each diagonal in parallel

Repeat for lower diagonals

Advantages:
- Reasonably clean (if I got my indexing correct)

Disadvantages:
- Not so great in terms of cache use
- A bit fine-grained
  - max parallelism = N/P
- Not ideal for distributed memory

Smith-Waterman

Naïve Data-Driven Task-Parallel Approach:

```
proc computeH(H: [0..n, 0..n] int) {
  const ProbSpace = H.domain.translate(1,1);
  var NeighborsDone: [ProbSpace] atomic int = 0;
  var Ready$: [ProbSpace] sync int;
  NeighborsDone[1, ..].add(1);
  NeighborsDone[.., 1].add(1);
  NeighborsDone[1, 1].add(1);
  Ready$[1,1] = 1;
  coforall (i,j) in ProbSpace {
    const goNow = Ready$[i,j];
    H[i,j] = f(H[i-1,j-1], H[i-1,j], H[i,j-1]);
    const eastReady = NeighborsDone[i,j+1].fetchAdd(1);
    const seReady = NeighborsDone[i+1,j+1].fetchAdd(1);
    const southReady = NeighborsDone[i+1,j].fetchAdd(1);
    if (eastReady == 2) then Ready$[i,j+1] = 1;
    if (seReady == 2) then Ready$[i+1,j+1] = 1;
    if (southReady == 2) then Ready$[i+1,j] = 1;
  }
}
```

Create domain describing shifted version off H's domain

Arrays to count how many of our 3 neighbors are done; and to signal when we can compute

Set up boundaries: north and west elements have a neighbor done; top-left is ready

Create a task per matrix element and have it block until ready

Compute our element

Increment our neighbors' counts

Signal our neighbors as ready if we're the third

Comparison of Synchronization Types in Chapel

sync/single:

- Best for producer/consumer style synchronization
- Imply a memory fence w.r.t. other loads/stores
- Use single to express write-once values

atomic:

- Best for uncoordinated accesses to shared state

Smith-Waterman

Naïve Data-Driven Task-Parallel Approach:

```
proc computeH(H: [0..n, 0..n] int) {
  const ProbSpace = H.domain.translate(1,1);
  var NeighborsDone: [ProbSpace] atomic int = 0;
  var Ready$: [ProbSpace] sync int;
  NeighborsDone[1, ..].add(1);
  NeighborsDone[.., 1].add(1);
  NeighborsDone[1, 1].add(1);
  Ready$[1,1] = 1;
  coforall (i,j) in ProbSpace {
    const goNow = Ready$[i,j];
    H[i,j] = f(H[i-1,j-1], H[i-1,j], H[i,j-1]);
    const eastReady = NeighborsDone[i,j+1].fetchAdd(1);
    const seReady = NeighborsDone[i+1,j+1].fetchAdd(1);
    const southReady = NeighborsDone[i+1,j].fetchAdd(1);
    if (eastReady == 2) then Ready$[i,j+1] = 1;
    if (seReady == 2) then Ready$[i+1,j+1] = 1;
    if (southReady == 2) then Ready$[i+1,j] = 1;
  }
}
```

Disadvantages:
- Still not great in cache use
- Uses n^2 tasks
- Most spend most of their time blocking

Smith-Waterman

Slightly Less Naïve Data-Driven Task-Parallel Approach:

```
proc computeH(H: [0..n, 0..n] int) {
  const ProbSpace = H.domain.translate(1,1);
  var NeighborsDone: [ProbSpace] atomic int = 0;
  NeighborsDone[1, ..].add(1);
  NeighborsDone[.., 1].add(1);
  NeighborsDone[1, 1].add(1);
  sync { computeHHelp(1,1); }

  proc computeHHelp(i,j) {
    H[i,j] = f(H[i-1,j-1], H[i-1,j], H[i,j-1]);
    const eastReady = NeighborsDone[i,j+1].fetchAdd(1);
    const seReady = NeighborsDone[i+1,j+1].fetchAdd(1);
    const southReady = NeighborsDone[i+1,j].fetchAdd(1);
    if (eastReady == 2) then begin computeHHelp(i,j+1);
    if (seReady == 2) then begin computeHHelp(i+1,j+1);
    if (southReady == 2) then begin computeHHelp(i+1,j);
  }
}
```

Rather than create the tasks *a priori*, fire them off once we know they're legal

sync to ensure they're all done before we go on

Smith-Waterman

Slightly Less Naïve Data-Driven Task-Parallel Approach:

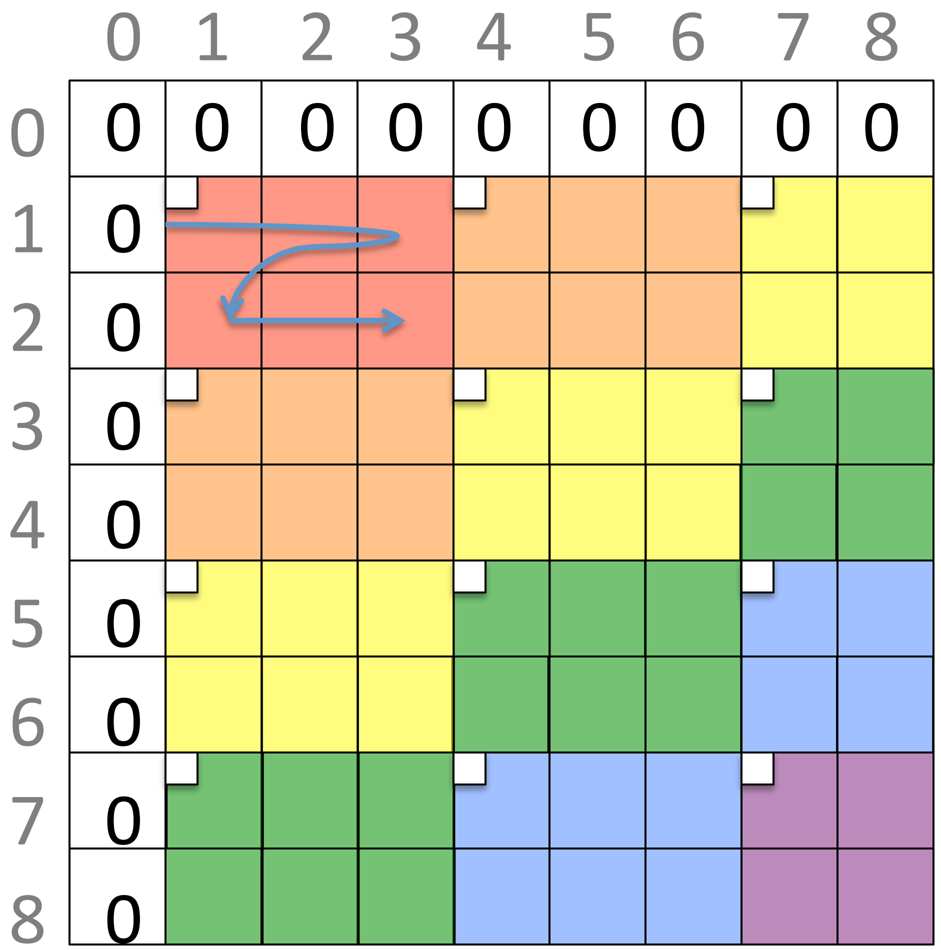
```
proc computeH(H: [0..n, 0..n] int) {
  const ProbSpace = H.domain.translate(1,1);
  var NeighborsDone: [ProbSpace] atomic int = 0;
  NeighborsDone[1, ..].add(1);
  NeighborsDone[.., 1].add(1);
  NeighborsDone[1, 1].add(1);
  sync { computeHHelp(1,1); }

  proc computeHHelp(i,j) {
    H[i,j] = f(H[i-1,j-1], H[i-1,j], H[i,j-1]);
    const eastReady = NeighborsDone[i,j+1].fetchAdd(1);
    const seReady = NeighborsDone[i+1,j+1].fetchAdd(1);
    const southReady = NeighborsDone[i+1,j].fetchAdd(1);
    if (eastReady == 2) then begin computeHHelp(i,j+1);
    if (seReady == 2) then begin computeHHelp(i+1,j+1);
    if (southReady == 2) then begin computeHHelp(i+1,j);
  }
}
```

Disadvantages:
- Still uses a lot of tasks
- Each task is very fine-grained

Smith-Waterman

Coarsening the Parallelism:



Smith-Waterman

Stride indices to get to next chunk

Blocked Data-Driven Task-Parallel Approach:

```
proc computeH(H: [0..n, 0..n] int) {
  const ProbSpace = H.domain.translate(1,1) by (rowsPerChunk, colsPerChunk);
  var NeighborsDone: [ProbSpace] atomic int = 0;
  NeighborsDone[1, ..].add(1);
  NeighborsDone[.., 1].add(1);
  NeighborsDone[1, 1].add(1);
  sync { computeHHelp({1..rowsPerChunk,1..colsPerChunk}); }

  proc computeHHelp(inds) {
    for (i,j) in H.domain[inds] do
      H[i,j] = f(H[i-1,j-1], H[i-1,j], H[i,j-1]);
    const (i,j) = inds.low;
    const eastReady = NeighborsDone[i,j+colsPerChunk].fetchAdd(1);
    const seReady = NeighborsDone[i+rowsPerChunk,j+colsPerChunk].fetchAdd(1);
    const southReady = NeighborsDone[i+rowsPerChunk,j].fetchAdd(1);
    if (eastReady == 2) then begin computeHHelp(i,j+colsPerChunk);
    if (seReady == 2) then begin computeHHelp(i+rowsPerChunk,j+colsPerChunk);
    if (southReady == 2) then begin computeHHelp(i+rowsPerChunk,j);
  }
}
```

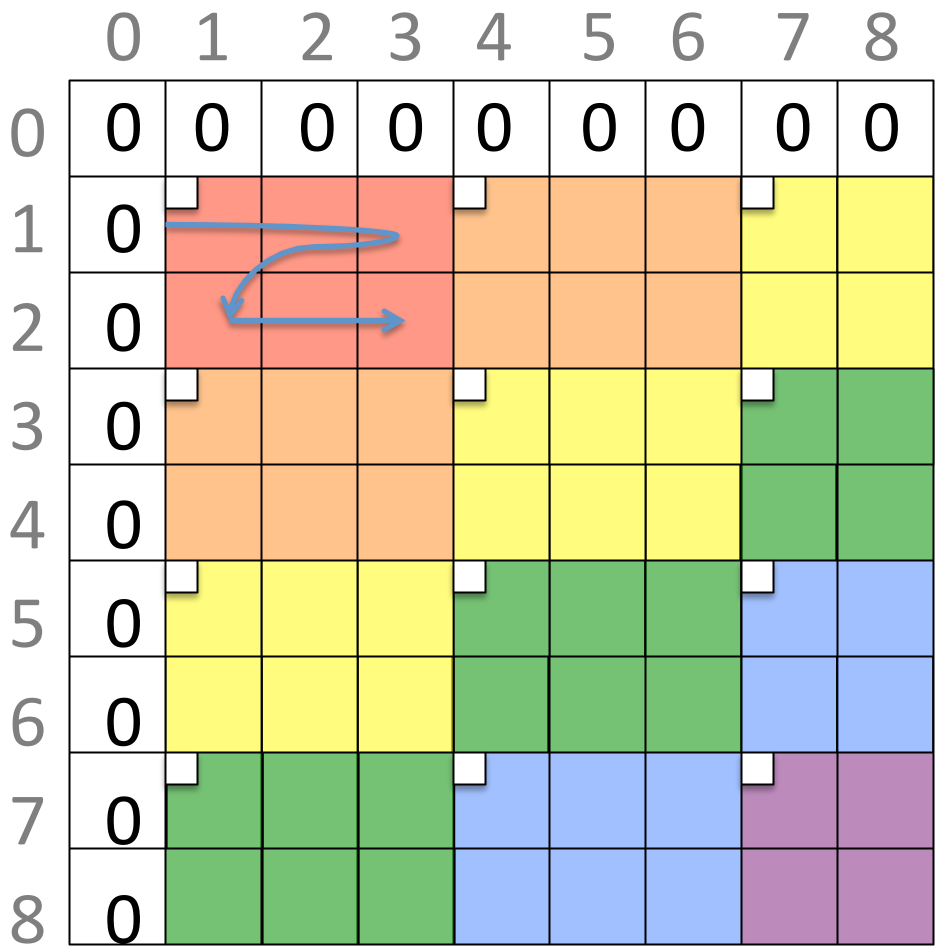
Can now use strided array for atomics

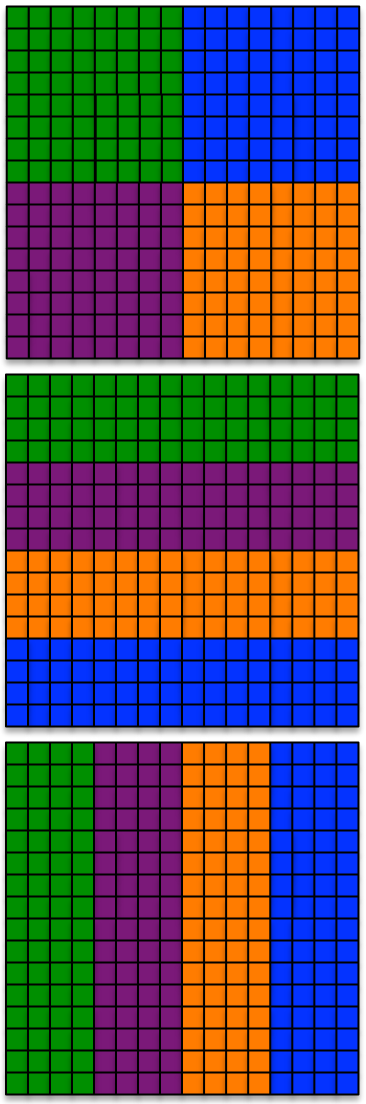
Change helper to take a domain describing the chunk to compute

Compute over chunk serially

Smith-Waterman

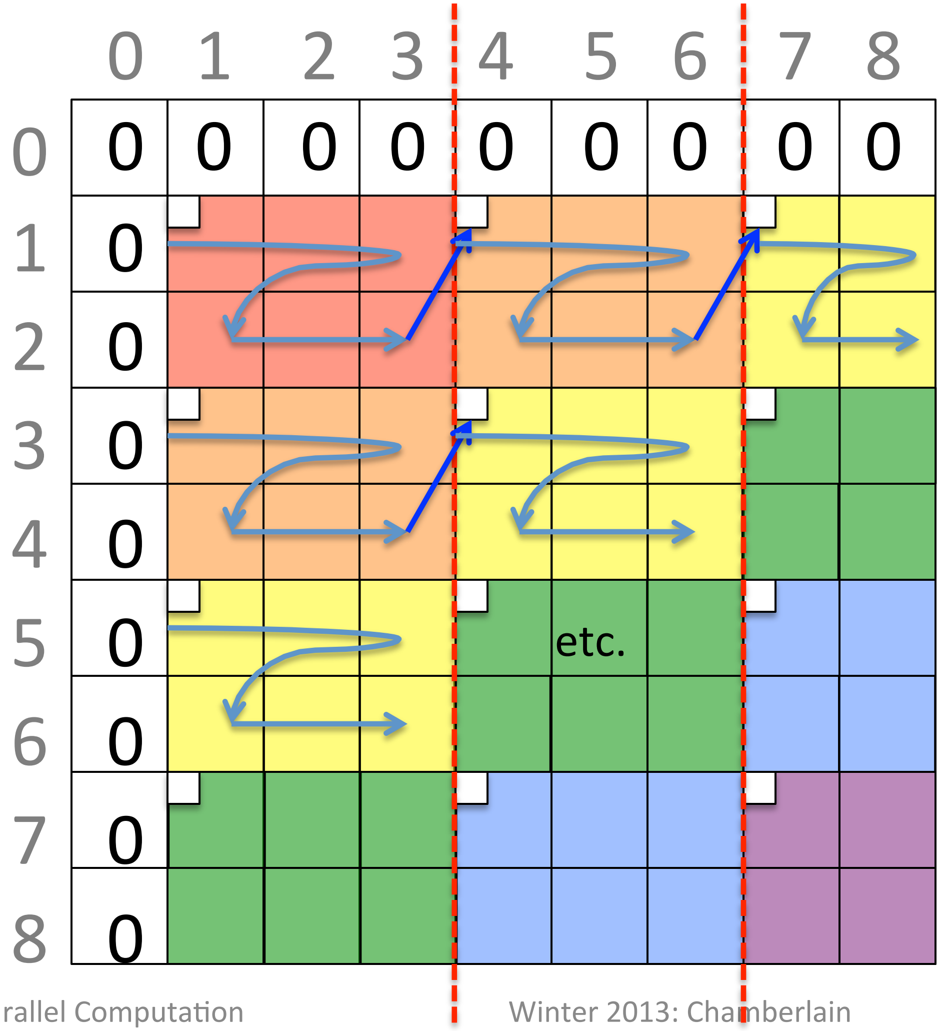
Now, what about distributed memory?





Smith-Waterman

Now, what about distributed memory?



Advantages:

- Good cache behavior: Nice fat blocks of data touchable in memory order
- Pipeline parallelism: Good utilization once pipeline is filled

Other notes:

- Communication pattern?
- Hybrid distributed + shared memory approach?

Chapel Domain Maps

(switch to other slide deck)

From the Course Description…

- **styles of parallelism**
  - data-parallel
  - task-parallel
  - concurrency
  - pipelined parallelism
  - nested parallelism
- **abstract programming models**
  - shared memory
  - Single Program, Multiple Data (SPMD)
  - message passing
  - Partitioned Global Address Space (PGAS)
- **architectural implications**
  - shared vs. distributed memory
  - multicore processors and accelerators
  - networks
  - caches and memory
- **programming issues and hazards**
  - synchronization
  - memory consistency
  - race conditions
  - deadlock and livelock
- **performance tuning**
  - scalability
  - locality
  - communication
  - scalar concerns
- **programming languages and notations**
  - OpenMP
  - MPI
  - UPC
  - Chapel
  - CUDA/OpenCL/OpenACC (?)
- **algorithms and patterns**
  - reductions and scans
  - stencils
  - graph algorithms
  - ...

Requests for next week?

- Amdahl’s Law
- modern compute nodes: CPU+GPU, NUMA nodes
- software transactional memory
- ZPL/HPF : Grand failures of the 90’s
- advanced Chapel concepts: user-defined arrays/foralls
- Dragonfly network
- open discussion questions
- more algorithms