



**Hewlett Packard**  
Enterprise

# **ONE-DAY CHAPEL TUTORIAL**

## **SESSION 4: MORE PARALLELISM**



Chapel Team  
October 16, 2023

# ONE DAY CHAPEL TUTORIAL

---

- 9-10:30: Getting started using Chapel for parallel programming
- 10:30-10:45: break
- 10:45-12:15: Chapel basics in the context of the n-body example code
- 12:15-1:15: lunch
- 1:15-2:45: Distributed and shared-memory parallelism especially w/arrays (data parallelism)
- 2:45-3:00: break
- 3:00-4:30: More parallelism including for asynchronous parallelism (task parallelism)
- 4:30-5:00: Wrap-up including gathering further questions from attendees



## OUTLINE: MORE PARALLELISM AND SOME BEST PRACTICES

---

- Spectrum of Chapel loops
- Task intents including reduce intents, and atomics
- Parallelizing histogram (Hands On)
- Story of index gather parallelization
- Other parallel constructs: 'cobegin', 'begin', 'sync',
- Avoiding races with task intents and task-private variables
- Performance gotchas
- Memory in Chapel and Arkouda
- Using CommDiagnostics



# **SPECTRUM OF CHAPEL LOOPS**

# SPECTRUM OF CHAPEL FOR-LOOP STYLES

See <https://chapel-lang.org/docs/primers/loops.html>  
for more details on loops.

**for loop:** each iteration is executed serially by the current task

- predictable execution order, similar to conventional languages

**foreach loop:** all iterations executed by the current task, but in no specific order

- a candidate for vectorization, SIMD execution on GPUs

**forall loop:** all iterations are executed by one or more tasks in no specific order

- implemented using one or more tasks, locally or distributed, as determined by the iterand expression

```
forall i in 1..n do ... // forall loops over ranges use local tasks only
forall (i,j) in {1..n, 1..n} do ... // ditto for local domains...
forall elem in myLocArr do ... // ...and local arrays
forall elem in myDistArr do ... // distributed arrays use tasks on each locale owning part of the array
forall i in myParIter(...) do ... // you can also write your own iterators that use the policy you want
```

**coforall loop:** each iteration is executed concurrently by a distinct task

- explicit parallelism; supports synchronization between iterations (tasks)

# IMPLICIT LOOPS: PROMOTION OF SCALAR SUBROUTINES & ARRAY OPS

- Any function or operator that takes scalar arguments can be called with array expressions instead

```
proc foo(x: real, y: real, z: real) {  
  return x**y + 10*z;  
}
```

- Interpretation is similar to that of a zippered forall loop, thus:

```
C = foo(A, 2, B);
```

is equivalent to:

```
forall (c, a, b) in zip(C, A, B) do  
  c = foo(a, 2, b);
```

as is:

```
C = A**2 + 10*B;
```



**TASK INTENTS INCLUDING REDUCE INTENTS**

# USING TASK INTENTS IN LOOPS

---

## Procedure argument intents (<https://chapel-lang.org/docs/primers/procedures.html?highlight=intents#argument-intents>)

- Tell how to pass a symbol actual argument into a formal parameter
- Default intent is 'const', which means formal can't be modified in procedure body
- 'ref' means formal can be changed AND that change will be visible elsewhere, e.g., at the callsite
- Others: 'in', 'out', and 'inout' refer to copying the actual argument in, the formal out, or both

## Task intents in loops

- Similar to argument intents in syntax and philosophy
- Also have a 'reduce' intent similar to OpenMP
- 'reduce' intent means each task has its own copy and specified operation like '+' will combine at end of loop

## Design principles

- Avoid common race conditions
- Avoid copies of (potentially) large data structures





# TASK INTENTS IN FORALL LOOPS: SCALARS

```
var sum: real;  
forall i in 1..n do  
  sum += computeMyResult(i);
```

Default intent of scalars is 'const in' so this is illegal (and avoids a race)

```
var sum: real;  
forall i in 1..n with (ref sum) do  
  sum += computeMyResult(i);
```

With 'ref' intent, we are requesting a race

```
var sum: real;  
forall i in 1..n with (+ reduce sum) do  
  sum += computeMyResult(i);
```

Override default intent so that each task accumulates its own copy. On loop exit, all tasks combine their results into original 'sum'

# FORALL INTENT EXAMPLES: ARRAYS



```
var bucketCount: [0..<m] real;  
forall i in 1..n with (ref bucketCount) do  
  bucketCount[i % m] += 1;
```

'ref' intent avoids array copies,  
but can result in data races

```
var bucketCount: [0..<m] real;  
forall i in 1..n with (in bucketCount) do  
  bucketCount[i % m] += 1;
```

'in' intent will result in  
each task having its own  
copy

```
var bucketCount : [0..<m] real;  
forall i in 1..n with (+ reduce bucketCount) do  
  bucketCount[i % m] += 1;
```

'reduce' intent will result in  
each task having own copy,  
but then on loop exit tasks  
combine their results into the  
original 'bucketCount' variable



# **ATOMIC VARIABLES**

# ATOMIC VARIABLES

---

## Meaning

- Atomic means 'indivisible'
- An atomic operation is indivisible.
- A thread of computation cannot interfere with another thread that is doing an atomic operation.

## Atomic Type Semantics in Chapel

- Supports operations on variable atomically w.r.t. other tasks
- Based on C/C++ atomic operations

## Example: Counting barrier

```
var count: atomic int, done: atomic bool;

proc barrier(numTasks) {
  const myCount = count.fetchAdd(1);
  if (myCount < numTasks - 1) then
    done.waitFor(true);
  else
    done.testAndSet();
}
```

# ARRAY OF ATOMIC

```
var bucketCount: [0..<m] atomic real;  
forall i in 1..n with (ref bucketCount) do  
  bucketCount[i % m].add(1);
```

Make the 'bucketCount' array  
contain 'atomic real's

Use the atomic 'add' operation

```
var bucketCount: [0..<m] atomic real;  
forall i in 1..n do  
  bucketCount[i % m].add(1);
```

Can leave off 'ref' intent, since that  
is the default for 'atomic' types



# **PARALLELIZING HISTOGRAM (HANDS ON)**

# HANDS ON: PARALLELIZING HISTOGRAM

## Goals

- Parallelize a program that computes a histogram using reductions
- Parallelize it using an array of atomic integers
- Compare the performance of both versions versus each other and the serial version

## Parallelize 'histogram-serial.chpl' using a 'forall' loop and a 'reduction' intent

1. Copy 'histogram-serial.chpl' into 'histogram-reduce.chpl'
2. Parallelize the serial 'for' loop using concepts from '04-task-intents.chpl'

## Parallelize 'histogram-serial.chpl' using an array of atomic integers

1. Copy 'histogram-serial.chpl' into 'histogram-atomic.chpl'
2. Parallelize the serial 'for' loop using concepts from '04-atomic-type.chpl'

## Compare the performance of all three

```
./histogram-serial --numNumbers=100000000 --printRandomNumbers=false --useRandomSeed=false  
./histogram-reduce --numNumbers=100000000 --printRandomNumbers=false --useRandomSeed=false  
./histogram-atomic --numNumbers=100000000 --printRandomNumbers=false --useRandomSeed=false
```

# HANDS ON EXTRA CREDIT: PARALLELIZE N-BODY



nbody.chpl

## Goals and Questions to Answer

- Parallelize as many loops in n-body as possible
- Determine when a 'reduce' intent or 'atomic' variable type is needed
- How can you check if you got the same answer?
- Is it possible for floating-point roundoff differences to change what the answers are slightly? For which loops?
- Did you get a performance improvement by doing the parallelization?





# ATOMIC METHODS

---

- `read () : t` return current value
- `write (v : t)` store *v* as current value
- `exchange (v : t) : t` store *v*, returning previous value
- `compareExchange (old : t, new : t) : bool`  
store *new* iff previous value was *old*; returns true on success
- `waitFor (v : t)` wait until the stored value is *v*
- `add (v : t)` add *v* to the value atomically
- `fetchAdd (v : t)` same, returning pre-sum value  
(*sub, or, and, xor* also supported similarly)
- `testAndSet ()` like `exchange(true)` for atomic bool
- `clear ()` like `write(false)` for atomic bool



# REDUCTIONS IN CHAPEL

- Recall the following snippet of code from the histogram exercise

```
// verify number of items in histogram is equal to number of random
// numbers and output timing results
if + reduce histogram != numNumbers then
  halt("Number of items in histogram does not match number of random numbers");
writeln("Histogram computed in ", timer.elapsed( ), " seconds\n");
```

- Standard reductions supported by default:

```
+, *, min, max, &, |, &&, ||, minloc, maxloc, ...
```

- Reductions can reduce arbitrary iterable expressions:

```
const total = + reduce Arr,
  factN = * reduce 1..n,
  biggest = max reduce (forall i in myIter() do foo(i));
```

# **STORY OF INDEX GATHER PARALLELIZATION**

# STORY ABOUT PARALLELIZING INDEX GATHER

- Computation in Bale that gathers spread out data into a packed array

```
for i in D do
  Dest[i] = Src[Inds[i]];
```

- Parallelize over threads using a 'forall'

```
forall i in D with (ref Dest) do
  Dest[i] = Src[Inds[i]];
```

- Parallelize by distributing the D2 domain and using a 'forall'

```
const D = blockDist.createDomain({0..numUpdates-1});
var Inds: [D] int;

forall i in D with (ref Dest) do
  Dest[i] = Src[Inds[i]];
```



# CHAPEL TENDS TO BE COMPACT, CLEAN, AND FAST (BALE INDEX-GATHER)

## Exstack version

```

i=0;
while( exstack_proceed(ex, (i==l_num_req)) ) {
  i0 = i;
  while(i < l_num_req) {
    l_indx = pckindx[i] >> 16;
    pe = pckindx[i] & 0xffff;
    if(!exstack_push(ex, &l_indx, pe))
      break;
    i++;
  }

  exstack_exchange(ex);

  while(exstack_pop(ex, &idx, &fromth)) {
    idx = ltable[idx];
    exstack_push(ex, &idx, fromth);
  }
  lgp_barrier();
  exstack_exchange(ex);

  for(j=i0; j<i; j++) {
    fromth = pckindx[j] & 0xffff;
    exstack_pop_thread(ex, &idx, (uint64_t)fromth);
    tgt[j] = idx;
  }
  lgp_barrier();
}

```

## Conveyors version

```

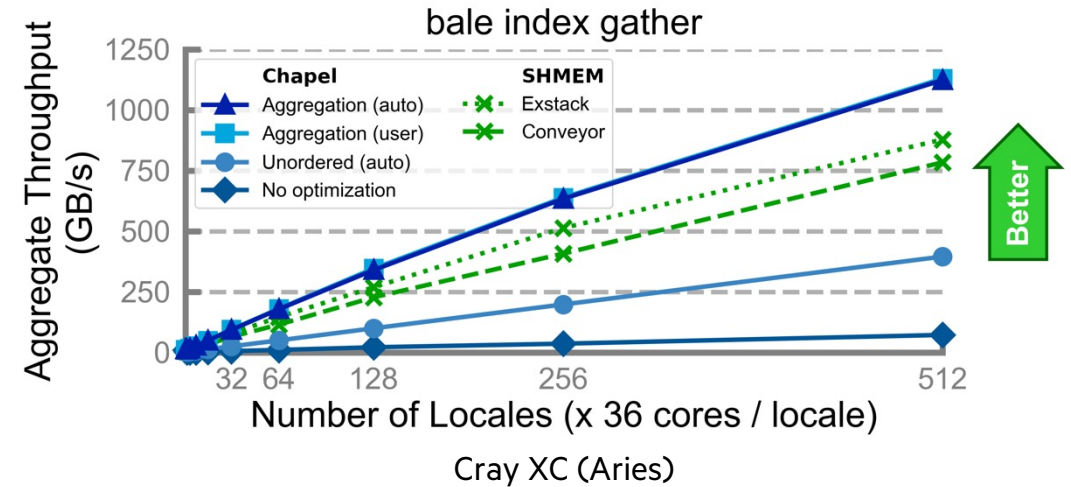
i = 0;
while (more = convey_advance(requests, (i == l_num_req)),
       more | convey_advance(replies, !more)) {

  for (; i < l_num_req; i++) {
    pkg.idx = i;
    pkg.val = pckindx[i] >> 16;
    pe = pckindx[i] & 0xffff;
    if (!convey_push(requests, &pkg, pe))
      break;
  }

  while (convey_pull(requests, ptr, &from) == convey_OK) {
    pkg.idx = ptr->idx;
    pkg.val = ltable[ptr->val];
    if (!convey_push(replies, &pkg, from)) {
      convey_unpull(requests);
      break;
    }
  }

  while (convey_pull(replies, ptr, NULL) == convey_OK)
    tgt[ptr->idx] = ptr->val;
}

```



## Manually Tuned Chapel version (using explicit aggregator type)

```

forall (d, i) in zip(Dest, Inds) with (var agg = new SrcAggregator(int)) do
  agg.copy(d, Src[i]);

```

## Elegant Chapel version (compiler-optimized w/ '--auto-aggregation')

```

forall (d, i) in zip(Dst, Inds) do
  d = Src[i];

```

# **OTHER PARALLEL CONSTRUCTS**

## DEFINING OUR TERMS

---

**Task:** a unit of computation that can/should execute in parallel with other tasks

**Thread:** a system resource that executes tasks

- not exposed in the language
- occasionally exposed in the implementation

**Task Parallelism:** a style of parallel programming in which parallelism is driven by programmer-specified tasks

(in contrast with):

**Data Parallelism:** a style of parallel programming in which parallelism is driven by computations over collections of data elements or their indices



# PARALLELISM SUPPORTED BY CHAPEL

## Synchronous task parallelism

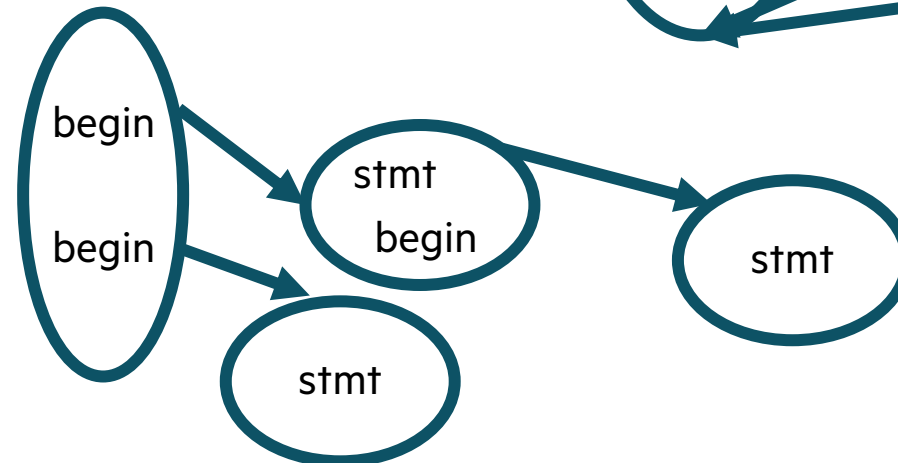
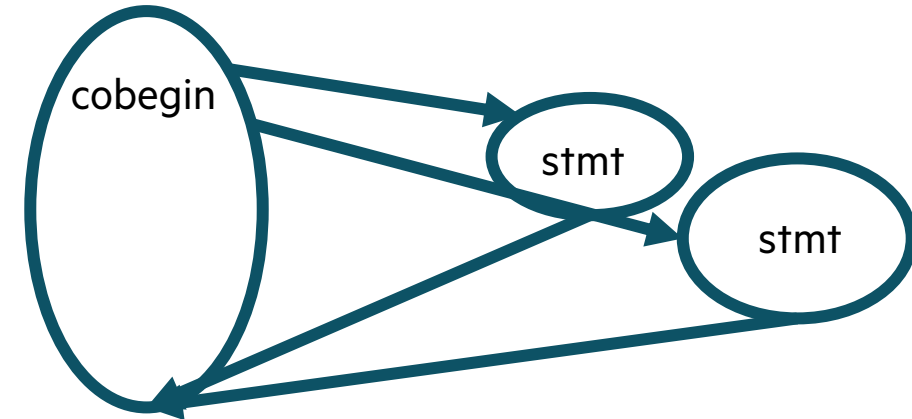
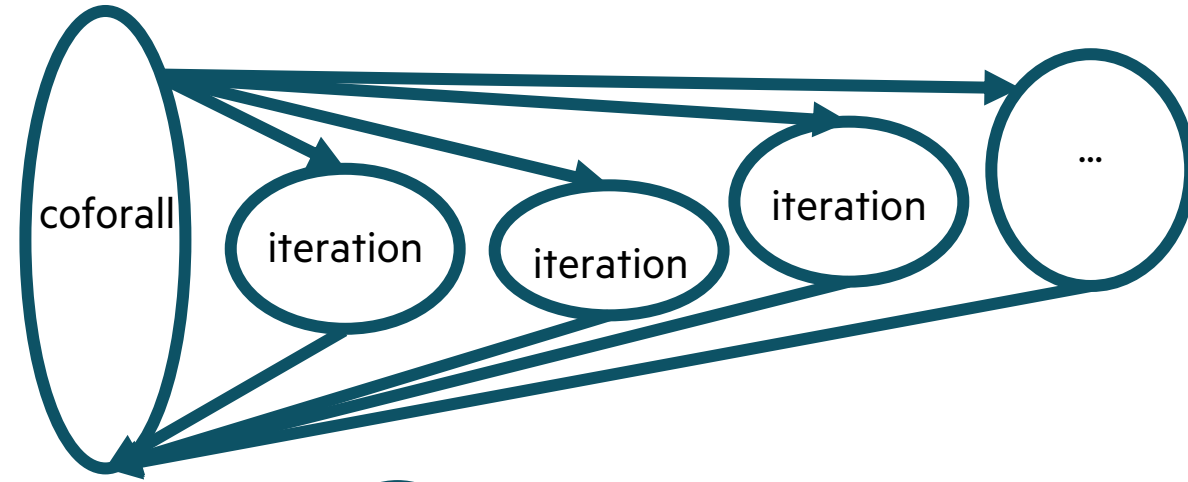
- 'coforall', parallel task per iteration
- 'cobegin', executes all statements in block in parallel

## Asynchronous task parallelism

- 'begin', creates an asynchronous task
- 'sync' and 'atomic' vars for task coordination

## Higher-level parallelism abstractions

- 'forall', data parallelism and iterator abstraction
- 'foreach', SIMD parallelism
- 'scan', operations such as cumulative sums
- 'reduce', operations such as summation





# PARALLELISM SUPPORTED BY CHAPEL



## Synchronous task parallelism

- 'coforall', parallel task per iteration
- 'cobegin', executes all statements in block in parallel

## Asynchronous task parallelism

- 'begin', creates an asynchronous task
- 'sync' and 'atomic' vars for task coordination

## Higher-level parallelism abstractions

- 'forall', data parallelism and iterator abstraction
- 'foreach', SIMD parallelism
- 'scan', operations such as cumulative sums
- 'reduce', operations such as summation

```
coforall loc in Locales do on loc { /* ... */ }
coforall tid in 0..<numTasks { /* ... */ }

cobegin { doTask0(); doTask1(); ... doTaskN(); }

var x : atomic int = 0, y : sync int;
sync {
  begin x.add(1);
  begin y.writeEF(1);
  begin x.sub(1);
  begin { y.readFE(); y.writeEF(0); }
}
assert(x.read() == 0);
assert(y.readFE() == 0);

var n = [i in 1..10] i*i;
forall x in n do x += 1;

var nPartialSums = + scan n;
var nSum = + reduce n;
```



# OTHER TASK PARALLEL FEATURES

- **begin / cobegin statements:** the two other ways of creating tasks

```
begin stmt; // fire off an asynchronous task to run 'stmt'
```

```
cobegin { // fire off a task for each of 'stmt1', 'stmt2', ...  
  stmt1;  
  stmt2;  
  stmt3;  
  ...  
} // wait here for these tasks to complete before proceeding
```

- **atomic / synchronized variables:** types for safe data sharing & coordination between tasks

```
var sum: atomic int; // supports various atomic methods like .add(), .compareExchange(), ...  
var cursor: sync int; // stores a full/empty bit governing reads/writes, supporting .readFE(), .writeEF()
```

- **task intents / task-private variables:** control how variables and tasks relate

```
coforall i in 1..nitems with (ref x, + reduce y, var z: int) { ... }
```

# USE OF PARALLELISM IN SOME APPLICATIONS AND BENCHMARKS

Application	Distributed 'coforall'	Threaded 'coforall'	Asynchronous 'begin'	'cobegin'	sync or atomic	forall	scan
Arkouda	✓	✓			✓	✓	✓
CHAMPS	✓	✓			✓		
ChOp	✓		✓		✓	✓	
ParFlow						✓	
Coral Reef	✓	✓		✓		✓	



# TASK PARALLELISM: BEGIN STATEMENTS

---

```
// create a fire-and-forget task for a statement  
begin writeln("hello world");  
writeln("goodbye");
```

Possible outputs:

```
hello world  
goodbye
```

```
goodbye  
hello world
```



# JOINING SUB-TASKS: SYNC-STATEMENTS

## Syntax

```
sync-statement:  
sync stmt
```

## Definition

- Executes *stmt*
- Waits for all *dynamically-scoped* begins to complete

## Examples

```
sync {  
  for i in 1..numConsumers {  
    begin consumer(i);  
  }  
  producer();  
}
```

```
proc search(node: TreeNode) {  
  if (node != nil) {  
    begin search(node.left);  
    begin search(node.right);  
  }  
}  
sync { search(root); }
```

# TASK PARALLELISM: COBEGIN STATEMENTS

---

```
// create a task per child statement  
cobegin {  
    producer(1);  
    producer(2);  
    consumer(1);  
} // implicit join of the three tasks here
```



# COBEGINS/SERIAL BY EXAMPLE: QUICKSORT



'cobegin' will start both 'quickSort' calls in parallel unless the number of running tasks would exceed the available HW parallelism

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

# TASK PARALLELISM: COFORALL LOOPS

---

```
// create a task per iteration  
coforall t in 0..#numTasks {  
    writeln("Hello from task ", t, " of ", numTasks);  
} // implicit join of the numTasks tasks here  
  
writeln("All tasks done");
```

## Sample output:

```
Hello from task 2 of 4  
Hello from task 0 of 4  
Hello from task 3 of 4  
Hello from task 1 of 4  
All tasks done
```





# COMPARISON OF BEGIN, COBEGIN, AND COFORALL

---

## **begin:**

- Use to create a dynamic task with an unstructured lifetime
- “fire and forget” (or at least “leave running for awhile”)

## **cobegin:**

- Use to create a related set of heterogeneous tasks  
...or a small, fixed set of homogenous tasks
- The parent task depends on the completion of the tasks

## **coforall:**

- Use to create a fixed or dynamic # of homogenous tasks
- The parent task depends on the completion of the tasks

**Note:** All these concepts can be composed arbitrarily



# **SYNCHRONIZATION VARIABLES**

# TASK PARALLELISM: DATA-DRIVEN SYNCHRONIZATION

---

- **sync variables:** store full-empty state along with value
- **atomic variables:** support atomic operations
  - e.g., compare-and-swap; atomic sum, multiply, etc.
  - similar to C/C++



# BOUNDED BUFFER PRODUCER/CONSUMER EXAMPLE



```
// 'sync' types store full/empty state along with value
var buff: [0..#buffersize] sync real;

begin producer();
consumer();

proc producer() {
  var i = 0;
  for ... {
    i = (i+1) % buffersize;
    buff[i].writeEF( ... ); // wait for empty, write, leave full
  }
}

proc consumer() {
  var i = 0;
  while ... {
    i = (i+1) % buffersize;
    ..buff[i].readFE()...; // wait for full, read, leave empty
  }
}
```



# SYNCHRONIZATION VARIABLES



## Syntax

```
sync-type:  
  sync type
```

## Semantics

- Stores *full/empty* state along with normal value
- Initially *full* if initialized, *empty* otherwise

## Examples: Critical sections and futures

```
var lock: sync bool;  
  
lock.writeEF(true);  
critical();  
lock.readFE();
```

```
var future: sync real;  
  
begin future.writeEF(compute());  
res = computeSomethingElse();  
useComputedResults(future.readFE(), res);
```

# SYNCHRONIZATION VARIABLE METHODS

---

- **readFE** () : t      block until *full*, leave *empty*, return value
- **readFF** () : t      block until *full*, leave *full*, return value
- **writeEF** (v : t)    block until *empty*, set value to  $v$ , leave *full*



# COMPARISON OF SYNCHRONIZATION TYPES

---

## **sync:**

- Best for producer/consumer style synchronization
  - “this task should block until something happens”
  - use single for write-once values

## **atomic:**

- Best for uncoordinated accesses to shared state
  - “these tasks are unlikely to interfere with each other, at least for very long...”



# **AVOIDING RACES WITH TASK INTENTS AND TASK PRIVATE VARIABLES**



# TASK INTENTS

---

- Tells how to “pass” variables from outer scopes to tasks
  - Similar to argument intents in syntax and philosophy
    - also adds a “reduce intent”, similar to OpenMP
  - Design principles:
    - “principle of least surprise”
    - avoid simple race conditions
    - avoid copies of (potentially) expensive data structures
    - support coordination via sync/atomic variables



# TASK INTENT EXAMPLES



```
var sum: real;  
coforall i in 1..n do  
    sum += computeMyResult(i);
```

Default task intent of scalars is 'const in' so this is illegal (and avoids a race)

```
var sum: real;  
coforall i in 1..n with (ref sum) do  
    sum += computeMyResult(i);
```

Use a 'ref' task intent for 'sum' variable. We've now requested a race.

```
var sum: real;  
coforall i in 1..n with (+ reduce sum) do  
    sum += computeMyResult(i);
```

Use a 'reduce' task intent. Per-task sums will be reduced on task exit.

```
var sum: atomic real;  
coforall i in 1..n do  
    sum.add(computeMyResult(i));
```

Default task intent of atomics is 'ref' so this is legal, meaningful, and safe

# TASK-PRIVATE VARIABLES



- Task-parallel features support task-private variables easily

```
coforall i in 1..numTasks {  
  var mySum: real; // each task gets its own copy of mySum  
  for j in 1..n do  
    mySum += A[i][j];  
}
```

- Forall loops need special support for task-private variables

```
var oneSingleVariable: real;  
forall i in 1..n {  
  var onePerIteration: real;  
}
```



# TASK-PRIVATE VARIABLES

- Task-parallel features support task-private variables easily

```
coforall i in 1..numTasks {  
  var mySum: real; // each task gets its own copy of mySum  
  for j in 1..n do  
    mySum += A[i][j];  
}
```

- Forall loops need special support for task-private variables

```
var oneSingleVariable: real;  
forall i in 1..n with (var onePerTask: real) {  
  var onePerIteration: real;  
}
```



# TASK-PRIVATE VARIABLES

- Task-parallel features support task-private variables easily

```
coforall i in 1..numTasks {  
    var mySum: real; // each task gets its own copy of mySum  
    for j in 1..n do  
        mySum += A[i][j];  
}
```

- Forall loops need special support for task-private variables

```
var oneSingleVariable: real;  
forall i in 1..n with (var onePerTask = 3.14) {  
    var onePerIteration: real;  
}
```



## PERFORMANCE / ARKOUDA ROADMAP

---

- Chapel best practices: General and for performance
  - Tips for compiling Arkouda faster
- Performance gotchas
- Memory in Chapel and Arkouda
- Stopwatches and benchmarks
- Using CommDiagnostics



# **CHAPEL BEST PRACTICES**

# CHAPEL BEST PRACTICES: GENERAL AND FOR PERFORMANCE

---

The three most common ways to build Chapel

- **‘quickstart’** configuration
  - Low performance, quickest build time, minimal dependency requirements
  - Not recommended for testing performance, not a fully-featured version of Chapel
- **‘CHPL\_COMM=none’** local configuration
  - Fully featured and best performance when running on a non-distributed system (e.g., your laptop)
  - Can potentially hit scaling issues when extending to multi-locale, as communication does not factor in
  - When comparing performance against non-distributed code from other languages, typically preferred configuration
- **‘CHPL\_COMM=gasnet’** multi-locale configuration
  - Enables multi-locale features, inserts code for remote accesses, works everywhere, but not always most optimized
  - Can be run on laptop for debugging purposes, but distribution is only simulated, so performance doesn’t mean much
- See <https://chapel-lang.org/docs/usingchapel/QUICKSTART.html> for more info

Make sure you are in the correct configuration for your system when testing performance

- Ask for help in the Chapel Discourse if you need help determining correct configuration!





# TIPS FOR COMPILING ARKOUDA FASTER

Quick, 1-step compilation time improvements that all developers should be using

- `export ARKOUDA_QUICK_COMPILE=1`
  - Disable optimizations, but performance will be worse (does not compile with `--fast`)
  - Recommended when developing new features or running correctness tests
- `export ARKOUDA_SKIP_CHECK_DEPS=1`
  - Skip compiling and running each of the Arkouda dependency tests when building Arkouda
  - Typically, the dependency tests only need to be run once per-machine, once you know they pass, they can be disabled
- Make sure that `CHPL_DEVELOPER` is unset
  - If the `chpl` compiler was built when `CHPL_DEVELOPER` was set, this can have an adverse effect on compilation times
  - If this was set when `chpl` was built, `chpl` should be rebuilt without it (doesn't apply to brew installs)

Effectively use the modular build system, only compiling features necessary for what is being tested

- See <https://bears-r-us.github.io/arkouda/setup/MODULAR.html> for more info

See <https://github.com/Bears-R-Us/arkouda/issues/2073> for more tips on speeding up compilation



# **PERFORMANCE GOTCHAS**

# PERFORMANCE GOTCHAS

---

- Compile with ‘--fast’
- Locate bottlenecks using stopwatches to identify which portion of code is running slowly
- Check the slow portion of the code for...
  - ...algorithmic overheads or tight-loop complexity
  - ...excessive remote accesses (use ‘CommDiagnostics’ to assess)
    - Are arrays distributed that should be? Are all locales accessing a variable declared on locale 0? Can you use aggregators?
  - ...extraneous dynamic (i.e., class object) allocations (use ‘MemDiagnostics’ to assess)
    - Could class allocations be reused in loops? Could a record be used instead of a class?
- Can be informative to compare standalone Chapel program to equivalent C/C++ code for local codes
  - Is Chapel slower than the same code in other languages?



# **MEMORY IN CHAPEL AND ARKOUDA**

# MEMORY IN CHAPEL AND ARKOUDA

---

## Arkouda's Memory Tracking

- Arkouda uses MemDiagnostics to estimate the total memory available and the total memory used
  - The maximum memory usage is set as 90% of Chapel's estimated available memory
- Only allocations larger than 1 MB are tracked
  - Tracking all allocations would have too large a performance impact

## Most Common Memory Allocation Modes in Chapel

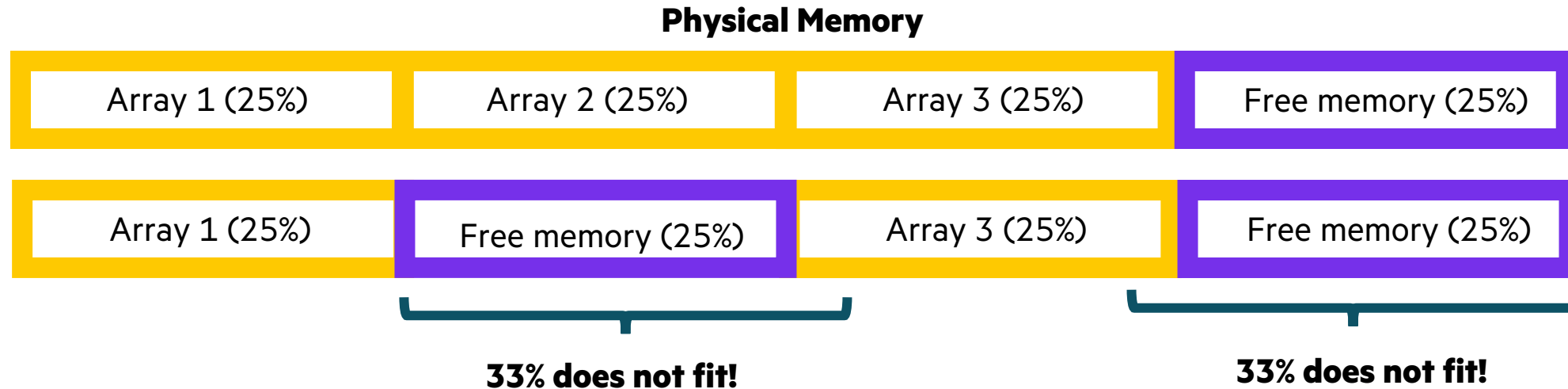
- **simple**: comm none, gasnet-everything
  - Fragmentation is handled completely by allocator, can use virtual memory and mmap for large arrays
- **fixed-heap**: gasnet-ibv, ofi-cxi
  - At program startup, a fixed segment of memory is allocated from comm layer
  - Allocator can only use provided memory region and not entire virtual address range



# MEMORY IN CHAPEL AND ARKOUDA

## Fragmentation

- Fragmentation can occur when allocating and freeing large blocks of memory
  - Common pattern in Arkouda



1. Allocate 3 arrays that are 25% of maximum memory each (25% memory available)
2. Free the second array (50% memory available)
3. Attempt to allocate an array that is 33% of maximum memory
  - Oh no! Our memory is fragmented, so we can't satisfy allocation, even though 50% of memory is available



# **STOPWATCHES AND BENCHMARKS**

# BENCHMARKS AND STOPWATCHES

---

## Benchmarks

- The Arkouda repository runs a number of performance tests that time Arkouda operations nightly
  - See <https://chapel-lang.org/perf/arkouda/>
- Benchmarks are useful for tracking historical performance data and gauging overall performance

## Stopwatches

- Stopwatches can be used to segment out portions of Chapel code and identify bottlenecks
- Rather than trying to guess what part of a function is slow, can isolate the code to optimize





**USING COMMDIAGNOSTICS**

# USING COMM DIAGNOSTICS

Chapel provides a comm table as well as a more advanced verbose comm mode

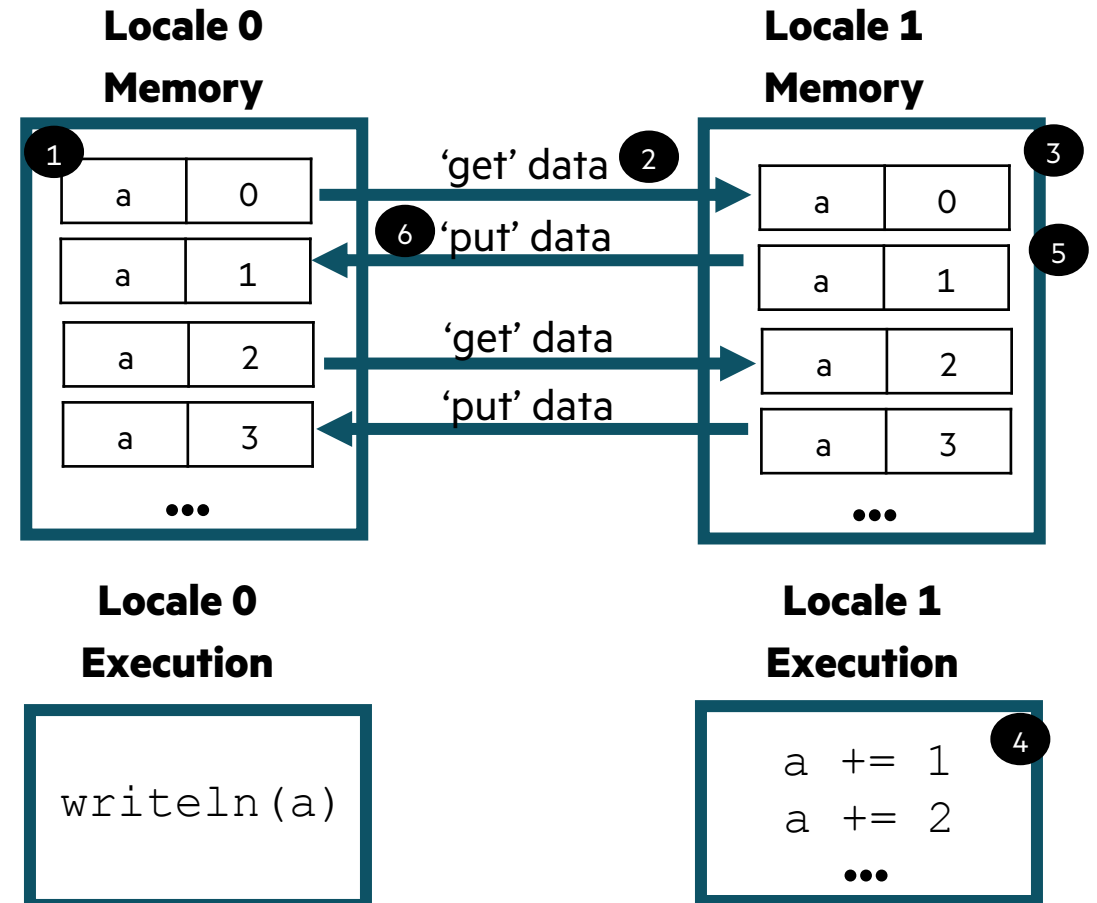
- Let's look at some code...

commTable.chpl

```
var a = 0; // allocated on locale 0

on Locales[1] do
  for i in 0..#5 do
    a += 1;

writeln(a); // print on locale 0
```



# USING COMMDIAGNOSTICS

commTable.chpl

```
use CommDiagnostics;
var a = 0; // allocated on locale 0

startCommDiagnostics();
on Locales[1] do
  for i in 0..#5 do
    a += 1;
stopCommDiagnostics();
printCommDiagnosticsTable();

writeln(a); // print on locale 0
```

```
prompt> chpl commTable.chpl --no-cache-remote
prompt> ./commTable -nl 2
```

locale	get	put	execute_on
0	0	0	1
1	5	5	0

5 'get's = 5 remote reads

5 'put's = 5 remote writes

```
prompt> chpl commTable.chpl
prompt> ./commTable -nl 2
```

locale	get_nb	put_nb	execute_on	cache_get_hits	cache_get_misses	cache_put_hits	cache_put_misses
0	0	0	1	0	0	0	0
1	1	1	0	4	1	4	1

# USING COMMDIAGNOSTICS

verboseComm.chpl

```
var a = 0; // allocated on locale 0

startVerboseComm();
on Locales[1] do
  for i in 0..#5 do
    a += 1;
stopVerboseComm();

writeln(a); // print on locale 0
```

```
prompt> chpl verboseComm.chpl --no-cache-remote
prompt> ./verboseComm -nl 2
0: verboseComm.chpl:6: remote executeOn, node 1
1: verboseComm.chpl:8: remote get, node 0, 8 bytes, commid 5
1: verboseComm.chpl:8: remote put, node 0, 8 bytes, commid 6
1: verboseComm.chpl:8: remote get, node 0, 8 bytes, commid 5
1: verboseComm.chpl:8: remote put, node 0, 8 bytes, commid 6
1: verboseComm.chpl:8: remote get, node 0, 8 bytes, commid 5
1: verboseComm.chpl:8: remote put, node 0, 8 bytes, commid 6
1: verboseComm.chpl:8: remote get, node 0, 8 bytes, commid 5
1: verboseComm.chpl:8: remote put, node 0, 8 bytes, commid 6
1: verboseComm.chpl:8: remote get, node 0, 8 bytes, commid 5
1: verboseComm.chpl:8: remote put, node 0, 8 bytes, commid 6
```

# USING COMMDIAGNOSTICS

- A more interesting example...

commOverDom.chpl

```
use CommDiagnostics;
use BlockDist;

config const size = 5;
var D = blockDist.createDomain(0..#size);
var a = 0;
startCommDiagnostics();
forall i in D with (ref a) do
  a += 1; // race condition!
stopCommDiagnostics();
printCommDiagnosticsTable();

writeln(a); // print on locale 0
```

```
prompt> chpl commOverDom.chpl --no-cache-remote
prompt> ./commOverDom -nl 2
```

locale	get	put	execute_on_nb
-----:	--:	--:	-----:
0	0	0	1
1	2	2	0

```
prompt> chpl commOverDom.chpl --no-cache-remote
prompt> ./commOverDom -nl 4 --size=100
```

locale	get	put	execute_on_nb
-----:	--:	--:	-----:
0	0	0	3
1	25	25	0
2	25	25	0
3	25	25	0

# GENERAL TIPS WHEN GETTING STARTED WITH CHAPEL

---

Online **documentation** is here: <https://chapel-lang.org/docs/>

- The primers can be particularly valuable for learning a concept: <https://chapel-lang.org/docs/primers/index.html>
  - These are also available from a Chapel release in ‘\$CHPL\_HOME/examples/primers/’  
or ‘\$CHPL\_HOME/test/release/examples/primers/’ if you clone from GitHub

When debugging, **almost anything in Chapel can be printed out** with ‘writeln(expr1, expr2, expr3);’

- Types can be printed after being cast to strings, e.g. ‘writeln(“Type of “, expr, “ is “, expr.type:string);’
- A quick way to print a bunch of values out clearly is to print a tuple made up of them ‘writeln((x, y, z));’

Once your code is correct, before doing any performance timings, be sure to re-compile with ‘**--fast**’

- Turns on optimizations, turns off safety checks, slows down compilation, speeds up execution significantly
- Then, when you go back to making modifications, be sure to stop using ‘—fast’ in order to turn checks back on

For vim / emacs users, **syntax highlighters** are in \$CHPL\_HOME/highlight

- Imperfect, but typically better than nothing
- Emacs MELPA users may want to use the chapel-mode available there (better in many ways, weird in others)



## OUTLINE: MORE PARALLELISM AND SOME BEST PRACTICES

---

- Spectrum of Chapel loops
- Task intents including reduce intents, and atomics
- Parallelizing histogram (Hands On)
- Story of index gather parallelization
- Other parallel constructs: 'cobegin', 'begin', 'sync',
- Avoiding races with task intents and task-private variables
- Performance gotchas
- Memory in Chapel and Arkouda
- Using CommDiagnostics



# LEARNING OBJECTIVES FOR TODAY'S CHAPEL TUTORIAL

---

- Familiarity with the Chapel execution model including how to run codes in parallel on a single node, across nodes, and both
- Learn Chapel concepts by compiling and running provided code examples
  - ✓ Serial code using map/dictionary, (k-mer counting from bioinformatics)
  - ✓ Parallelism and locality in Chapel
  - ✓ Distributed parallelism and 1D arrays, (processing files in parallel)
  - ✓ Chapel basics in the context of an n-body code
  - ✓ Distributed parallelism and 2D arrays, (heat diffusion problem)
  - ✓ How to parallelize histogram
  - ✓ Using CommDiagnostics for counting remote reads and writes
  - ✓ Chapel and Arkouda best practices including avoiding races and performance gotchas
- Where to get help and how you can participate in the Chapel community





# TUTORIAL SUMMARY

---

## • Takeaways

- Chapel is a general-purpose programming language designed to leverage parallelism
- It is being used in some large production codes
- Our team is responsive to user questions and would enjoy having you participate in our community

## • How to get more help and engage with the community

- Ask us questions on discourse, gitter, or stack overflow
- Share your sample codes with us and your research community!
- Join us at our free, virtual workshop in June, <https://chapel-lang.org/CHI UW.html>

## • Potential follow-on topics

- Using classes in Chapel including memory management
- Generics in Chapel: enabling the same code to work for multiple types
- Chapel interoperability with C
- Your suggestions?



# ONE DAY CHAPEL TUTORIAL

---

- 9-10:30: Getting started using Chapel for parallel programming
- 10:30-10:45: break
- 10:45-12:15: Chapel basics in the context of the n-body example code
- 12:15-1:15: lunch
- 1:15-2:45: Distributed and shared-memory parallelism especially w/arrays (data parallelism)
- 2:45-3:00: break
- 3:00-4:30: More parallelism including for asynchronous parallelism (task parallelism)
- 4:30-5:00: Wrap-up including gathering further questions from attendees



# CHAPEL RESOURCES

Chapel homepage: <https://chapel-lang.org>


- (points to all other resources)

## Social Media:

- Twitter: [@ChapelLanguage](https://twitter.com/ChapelLanguage)
- Facebook: [@ChapelLanguage](https://www.facebook.com/ChapelLanguage)
- YouTube: <http://www.youtube.com/c/ChapelParallelProgrammingLanguage>

## Community Discussion / Support:

- Discourse: <https://chapel.discourse.group/>
- Gitter: <https://gitter.im/chapel-lang/chapel>
- Stack Overflow: <https://stackoverflow.com/questions/tagged/chapel>
- GitHub Issues: <https://github.com/chapel-lang/chapel/issues>



### The Chapel Parallel Programming Language

**What is Chapel?**

Chapel is a programming language designed for productive parallel computing at scale.

**Why Chapel?** Because it simplifies parallel programming through elegant support for:

- **distributed arrays** that can leverage thousands of nodes' memories and cores
- a **global namespace** supporting direct access to local or remote variables
- **data parallelism** to trivially use the cores of a laptop, cluster, or supercomputer
- **task parallelism** to create concurrency within a node or across the system

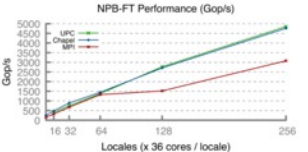
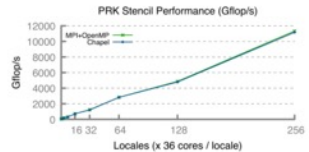
**Chapel Characteristics**

- **productive:** code tends to be similarly readable/writable as Python
- **scalable:** runs on laptops, clusters, the cloud, and HPC systems
- **fast:** performance *competes with or beats* C/C++ & MPI & OpenMP
- **portable:** compiles and runs in virtually any \*nix environment
- **open-source:** hosted on GitHub, permissively licensed

**New to Chapel?**

As an introduction to Chapel, you may want to...

- watch an [overview talk](#) or browse its [slides](#)
- read a [blog-length](#) or [chapter-length](#) introduction to Chapel
- learn about [projects powered by Chapel](#)
- check out [performance highlights](#) like these:



PRK Stencil Performance (Gflop/s)

Locales (x 36 cores / locale)	OpenMP	Chapel
16	~1000	~1000
32	~2000	~2000
64	~4000	~4000
128	~8000	~8000
256	~12000	~12000

NPB-FT Performance (Gop/s)

Locales (x 36 cores / locale)	OpenMP	Chapel	MPI
16	~1000	~1000	~1000
32	~2000	~2000	~2000
64	~4000	~4000	~4000
128	~8000	~8000	~8000
256	~12000	~12000	~12000

- browse [sample programs](#) or learn how to write distributed programs like this one:

```
use CyclicDist;           // use the Cyclic distribution library
config const n = 100;     // use --n=<val> when executing to override this default

forall i in {1..n} dmapped Cyclic(startIdx=1) do
  writeln("Hello from iteration ", i, " of ", n, " running on node ", here.id);
```

