



MORE PARALLELISM IN CHAPEL

Chapel Team, edited by Michelle Strout for CSc 372 at UArizona

April 22, 2025

PLAN

- **Announcements**

- Brad Chamberlain, Chapel co-creator, visiting Thursday April 24th
- LA3 is due on Friday April 25th (3 days)
- Final projects are due Friday May 2nd (10 days)

- **Last time**

- Recall extra credit opportunities
- Memory safety comparison between languages

- **Today**

- TopHat questions about Questions for Brad, Memory safety, and Chapel implicit communication
- Parallelizing histogram in Chapel
- Other parallel constructs

OUTLINE: MORE PARALLELISM

- Spectrum of Chapel loops
- Task intents including reduce intents
- Parallelizing histogram
- Story of index gather parallelization
- Other parallel constructs: 'cobegin', 'begin', 'sync`
- Avoiding races with task intents and task-private variables
- Performance gotchas



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;
```



IN CLASS ACTIVITY: UNDERSTANDING PROMOTION AND ZIP

- Questions to answer with Chapel code examples from previous slide
 - What sizes can arrays A, B, and C be declared with relative to each other for the following to work? What happens if C is bigger? A and B are bigger?

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

- Using zip and unbounded ranges, how could we get an array of tuples (#, elem), where # is the index of each element in an array?

```
<TODO in class>
```

- Using zip and unbounded ranges, how could we get an array of tuples (#,elem), where # is the count of each element in an array?

```
<TODO in class>
```



TASK INTENTS INCLUDING REDUCE INTENTS

USING TASK INTENTS IN LOOPS

Recall 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'*

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 FURTHER INVESTIGATION: 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_idx = pckindx[i] >> 16;
    pe = pckindx[i] & 0xffff;
    if(!exstack_push(ex, &l_idx, 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();
}
```

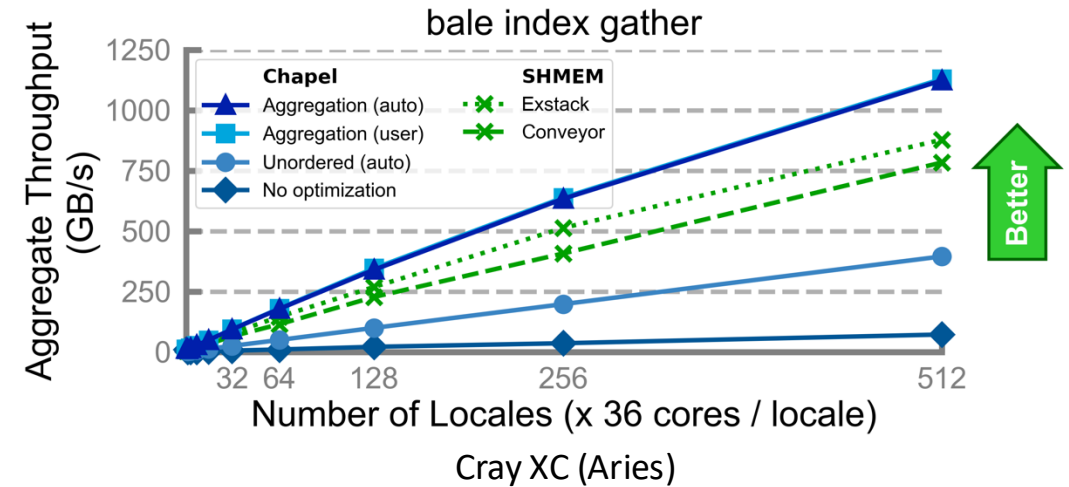
Conveyors version

```
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

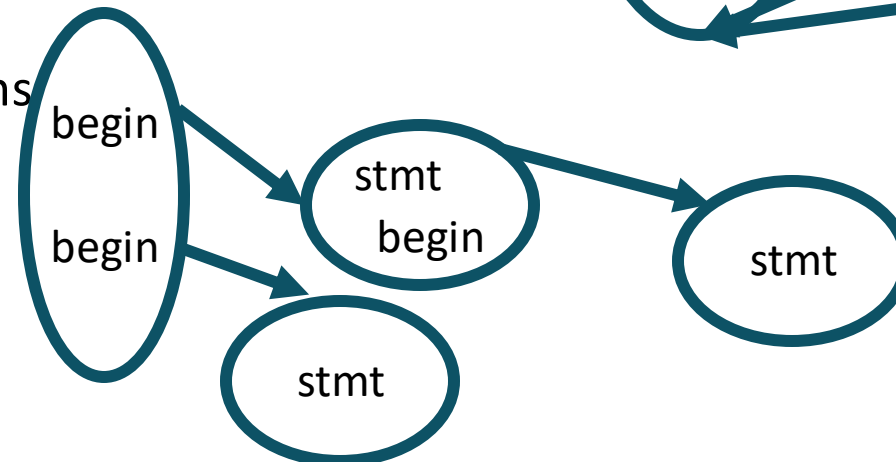
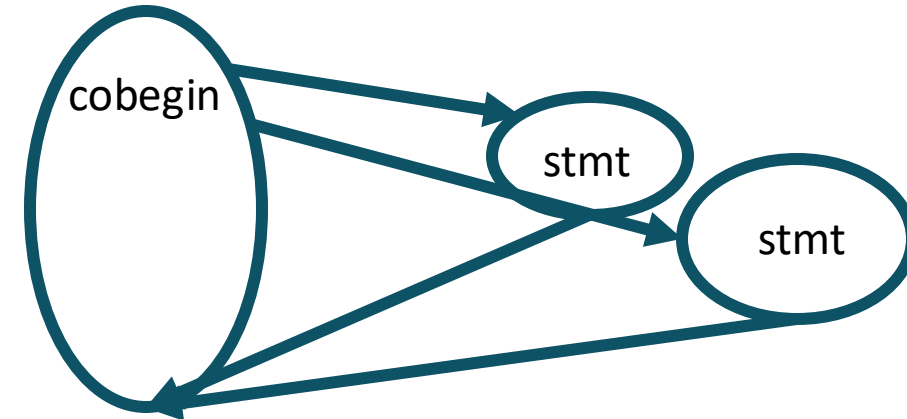
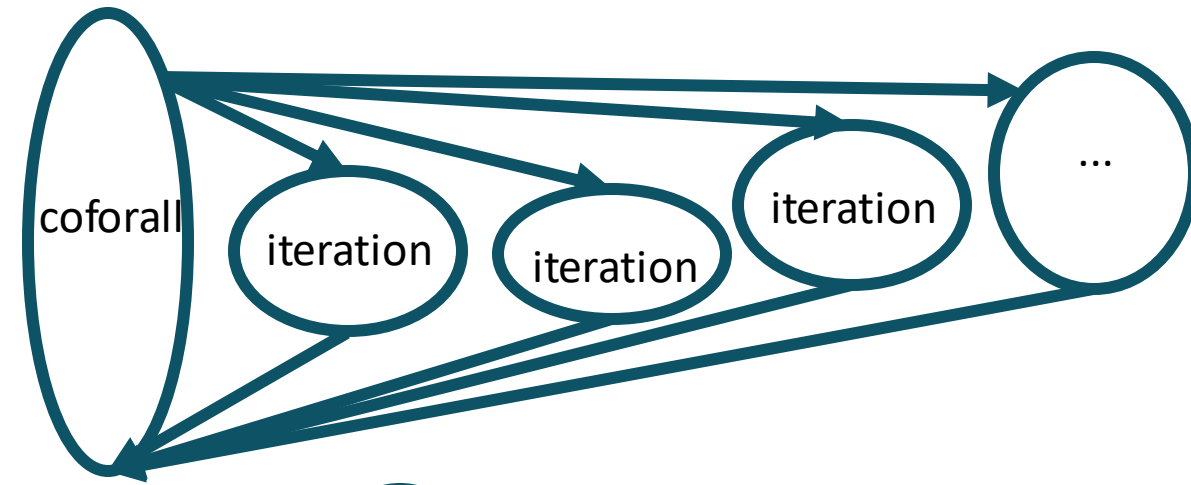
- 'coforall', parallel task per iteration
- 'cobegin', executes all statements in block in parallel

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

```
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()
```

- **coforall** *i* in 1..*n*iters **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



04-quicksort.chpl

'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
  }
}
```

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;  
}
```



CHAPEL RESOURCES

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

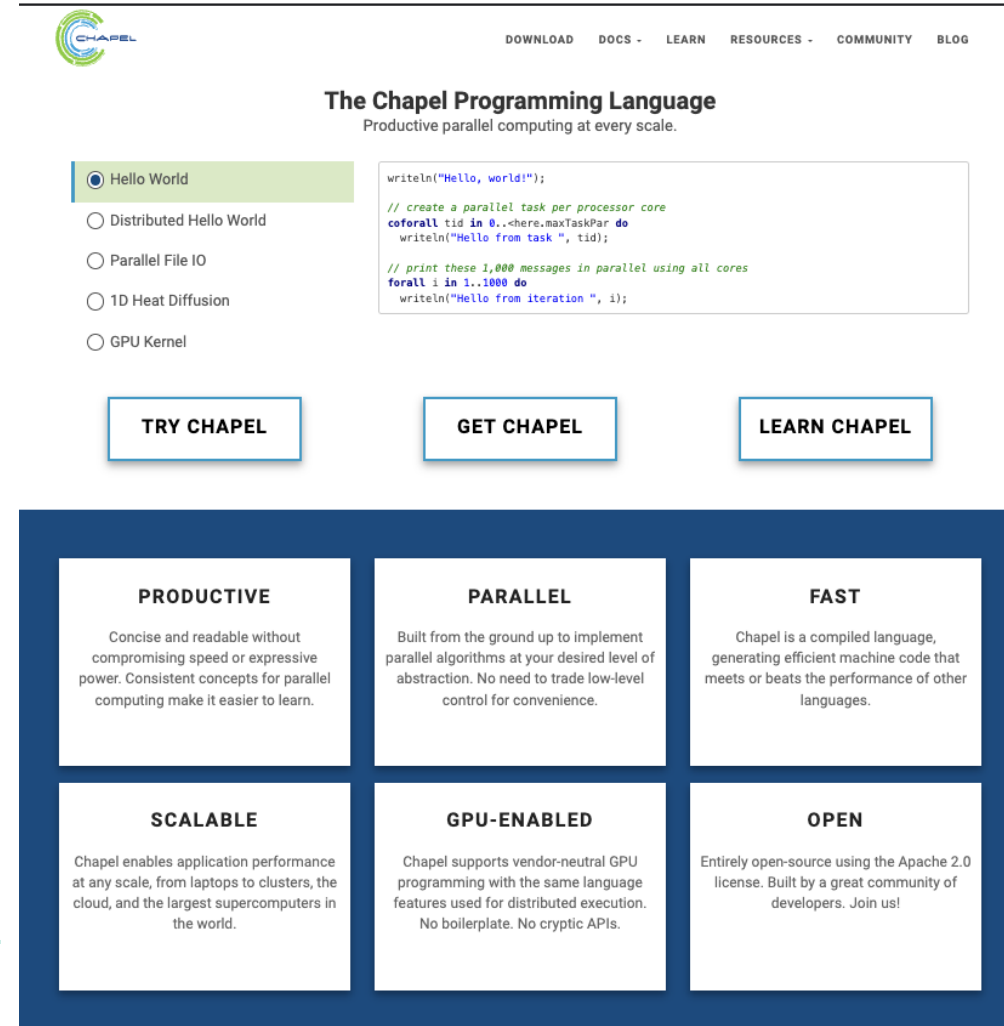
- (points to all other resources)

Social Media:

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

Community Discussion / Support:

- Discord: <https://discord.com/invite/xu2xg45yqH>
- Stack Overflow: <https://stackoverflow.com/questions/tagged/chapel>
- GitHub Issues: <https://github.com/chapel-lang/chapel/issues>



The screenshot shows the Chapel Programming Language homepage. At the top, there is a navigation bar with links: DOWNLOAD, DOCS, LEARN, RESOURCES, COMMUNITY, and BLOG. The main heading is "The Chapel Programming Language" with the tagline "Productive parallel computing at every scale." Below this, there is a section for "Hello World" with a radio button selected. To the right of this section is a code editor showing a "Hello World" program in Chapel. Below the "Hello World" section are three buttons: "TRY CHAPEL", "GET CHAPEL", and "LEARN CHAPEL". At the bottom, there is a grid of six boxes, each describing a feature of Chapel: PRODUCTIVE, PARALLEL, FAST, SCALABLE, GPU-ENABLED, and OPEN.

PRODUCTIVE
Concise and readable without compromising speed or expressive power. Consistent concepts for parallel computing make it easier to learn.

PARALLEL
Built from the ground up to implement parallel algorithms at your desired level of abstraction. No need to trade low-level control for convenience.

FAST
Chapel is a compiled language, generating efficient machine code that meets or beats the performance of other languages.

SCALABLE
Chapel enables application performance at any scale, from laptops to clusters, the cloud, and the largest supercomputers in the world.

GPU-ENABLED
Chapel supports vendor-neutral GPU programming with the same language features used for distributed execution. No boilerplate. No cryptic APIs.

OPEN
Entirely open-source using the Apache 2.0 license. Built by a great community of developers. Join us!