

# Chapel Realms: Language Support for Hybrid Computation<sup>†</sup>

Chapel Team, Cray Inc., chapel\_info@cray.com

## I. MOTIVATION

This paper describes the *realm*—a proposed extension to the Chapel parallel programming language for the purpose of executing a single Chapel program using multiple distinct target architectures. The assumption is that a Chapel user would want to do this in order to take advantage of the unique processing capabilities of each architecture, or simply to harness the increased computational power available when using multiple machines in concert. The immediate motivation for this work is to support heterogeneous node types within a single system such as the multithreading and SIO nodes of a Cray XMT, or the Marble and Granite nodes of a Cray Cascade system. We also envision realms as being useful for distributing a computation across multiple distinct systems that happen to share a network.

## II. BACKGROUND

This section provides a brief overview of how Chapel currently represents the target architecture’s resources in order to provide context for the new concepts introduced by this paper. For further details, please refer to the Chapel Language Specification.<sup>1</sup>

### A. Chapel Locales

Currently, the Chapel language defines the concept of a *locale* to represent a unit of the target system architecture that is useful for reasoning about locality. The purpose of the locale is to permit the programmer to control how a program executes relative to the target architecture’s resources. In particular, locales enable a programmer to specify and query where data is stored and where tasks execute. This ability to reason about locality and affinity is often crucial for obtaining scalable performance on large-scale systems given that memory is typically distributed between a machine’s nodes. In an application that has poor affinity between its tasks and the variables they access, the resulting network latencies can become a bottleneck and a barrier to achieving scalable performance.

Chapel’s locales are defined as follows. A locale has the ability to execute computations (tasks) and to store data (variables). Tasks can access variables stored in any locale, but variables stored within the task’s locale can be accessed more cheaply than those in other (remote) locales. Any two tasks running within the same locale should have fairly similar access times to a given variable stored on the system.

The definition of a locale is intentionally abstract in order to make it applicable to a wide variety of target architectures. The specific meaning of locale for a given target architecture is defined by the Chapel compiler used to build the program. This binding should be well-documented to permit users to reason about how their Chapel programs will execute on a given system. For example, our current Chapel compiler defines a locale for a commodity cluster or Cray XT4 as a single node of the architecture—a multicore processor or SMP node and its associated memory. For a machine like the Cray XMT which presents a completely flat view of shared memory to the user, all of the nodes being used to execute a user’s job are considered to be a single locale since there is no distinction in the programming model between “here” and “there,” nor any way at the user’s level to specify where a particular piece of data should be allocated or a task run.

The current implementation of Chapel assumes that locales are reasonably homogeneous. For example, a group of locales must use the same data representation and must be able to run a single binary executable. Aside from these rules, minor differences are permitted. For example, the number of processing cores and amount of memory may vary from one locale to the next. The target platform is specified by the user at compile time via a flag or environment variable.

At Chapel program execution time, the user can specify the number of locales on which the program should be run. For example, in our current compiler, a user wanting to execute a program *myProg* on 16 locales of the target platform would write:

```
prompt% ./myProg -nl 16
```

or:

```
prompt% ./myProg --numLocales=16
```

This request is used by the Chapel program to allocate the corresponding machine resources and to load the binary onto the appropriate nodes. After this bootstrapping process, the user code begins executing from its entry point (*main()*) using a single logical task running on locale 0.

<sup>†</sup>This work was funded under the Center for Adaptive Supercomputing Software—Multithreaded Architectures (CASS-MT) at the Dept. of Energy’s Pacific Northwest National Laboratory. Pacific Northwest National Laboratory is operated by Battelle Memorial Institute under Contract DE-ACO6-76RL01830. The material in this report is also based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0001.

<sup>1</sup>The current version of the *Chapel Language Specification* (0.780 at the time of this writing) is available at <http://chapel.cs.washington.edu>.

## B. Locales within the Chapel Language

Within the language itself, Chapel represents the locale concept using a type named `locale`. During the bootstrapping process, a unique locale value is created for each of the locales requested by the user. These values represent the machine resources on which the code is running, and each locale value is allocated within the memory of the locale that it represents. In the current language design, locale values may neither be created nor destroyed during program execution.

The Chapel programmer is provided with three built-in variables that represent the set of locales on which the program is executing. An integer variable `numLocales` represents the number of locales as specified by the user at execution time. A 1-dimensional array, `Locales`, stores the `numLocales` unique locale values that represent the machine resources being used. And finally, a 1-dimensional domain, `LocaleSpace`, represents the index set corresponding to the `Locales` array:  $0 \dots \text{numLocales} - 1$ . These variables can be thought of as being implemented using the following Chapel declarations:

```
config const numLocales: int = 1;

const LocaleSpace = [0..#numLocales];

const Locales: [LocaleSpace] locale = ...;
```

Locality-minded programmers can use these variables to control where their program data is stored and where their tasks execute. The main feature for doing so is the *on-clause*, which prefixes a Chapel statement to indicate that it should be executed on a specific locale. The *on-clause* takes a single expression as its argument. If the expression evaluates to a locale value, the corresponding statement will be executed on that locale. If the expression is a variable expression with storage associated with it, the statement will be executed by the locale on which the value is stored. If the expression is a literal expression then the *on-clause* is degenerate (*i.e.*, the statement will execute locally) since the value will be stored locally to the currently-executing task.

Here are some examples of the *on-clause* in use:

```
// Program execution starts on locale 0

var x: real;           // x is stored on locale 0

on Locales(1) {        // move execution to locale 1
  var y: real;         // y is stored on locale 1

  on x {               // move to where x is (locale 0)
    y += x;           // requires a remote get/put of y
  }                  // return to locale 1

  on 0 do              // degenerate since the integer
    write("hi");      // literal "0" is local to the
                    // current task by definition;
                    // thus, it runs on the current
                    // locale (locale 1 in this case)
}                    // return to locale 0
```

Note that while this program is sequential, in practice *on-clauses* are often used with task invocations to launch a number of parallel tasks across a set of locales. For example, a very

common idiom is to use a parallel loop to start a task per locale:

```
coforall loc in Locales do
  on loc do
  ...
```

Chapel users can query the placement of data and tasks using a small set of intuitive language concepts. A built-in method `.locale` can be applied to any variable and returns the locale on which that variable is stored. A built-in locale-private constant, *here*, can be used to refer to the locale on which the current task is executing.

Locale values support a few important methods as well. Applying the `.id` method to a locale value returns its index from 0 to `numLocales - 1`. The `.name` method returns a string corresponding to the identity of the locale on the target machine, similar to that which would be returned by invoking `hostname` from a shell running on the node. Other methods support the ability to query the amount of memory or number of processor cores associated with the locale.

Here is a simple example of these concepts in use:

```
// Program execution starts on locale 0

var x: real;           // x is stored on locale 0

writeln((x.locale.id,
         here.id));    // writes (0, 0)

on Locales(1) do      // migrate to locale 1
  writeln((x.locale.id,
          here.id));  // writes (0, 1)
```

Finally, locales are important to Chapel's concept of distributed domains and arrays in which a single logical data aggregate can be implemented using the disparate memories of multiple locales. Most distributions take a set of locales as an argument and distribute the domain indices and array elements between those locales. For example our *Block1D* distribution which partitions 1D domains/arrays across a set of locales takes a 1D array of locales as an argument, with a default value of `Locales`. In this way, users can specify a specific set of locales over which to distribute an array, or they can distribute it over all of the locales by default.

## C. Possible Extensions to Locales

There are a number of ways in which Chapel's current locale concept could be extended to make Chapel more general, flexible, or useful. We present a list of such extensions here:

- As proposed in the current work, the locale concept could be extended to support heterogeneous or hybrid target architectures. In this model, we envision a Chapel program being run across multiple machines, each with distinct capabilities. For example, the program may have a highly dynamic, unpredictable phase that would run best on an XMT; a highly regular, computationally intensive phase that would be well-suited for the XMT's SIO nodes; and a visualization package that would be most usefully run on a desktop workstation.
- A second means of introducing heterogeneity into the locale concept would be to expose distinct capabilities

within a processing node, such as the presence of coprocessors or accelerators in the form of (GP)GPUs, vector units, etc. In our current thinking, this would be done by making the locale type less of a black box, permitting the user to refer to architectural substructures within the locale.

- Similar to the previous item, even in a reasonably homogeneous node such as a multicore processor, one might want to control the placement of threads on cores or the placement of data within memory banks—particularly as multicore chips become larger and more hierarchical making intranode locality more of an issue. Again, we would imagine making this change by allowing the locale concept to be “opened up,” permitting the user to refer to hierarchy within the locale.
- The current restriction against creating and destroying locale values could be relaxed, permitting the set of locales to grow and shrink during program execution. This capability could permit a programmer to dynamically change the size and/or membership of their *Locales* array in order to adapt to changing program requirements or system utilization. It may also provide a means of reflecting failed nodes or varying system resources to the program.
- A layer of virtualization could be injected between the language-level locale values and the resources on which the program is running, for instance to provide resiliency by mapping a single locale value to multiple physical nodes and performing redundant storage and computation to tolerate machine failures.

While all of these directions are of interest to us as language designers and developers, this paper and our current scope of work only tackle the first of these items.

### III. REALMS

This section introduces the proposed realm concept for describing multiple target architectures. The purpose of a realm is to introduce a new language type and value that represents distinct target systems, permitting users to bind computations and data to the machines just as they currently use locales to do so for a single machine’s resources. We distinguish between realms and locales for a couple of reasons: First, because realms generally require the compiler to create multiple executables, one per target architecture. Second, because the cost of communicating between realms will typically be greater than communicating between locales due to increased latencies and the potential for data representation conversions. By exposing this distinction, we enable the compiler and user to reason about and optimize inter- vs. intra-realm communications.

A multi-realm Chapel program will commence execution of the user’s entry point using a single logical task executing on locale 0 of realm 0. Existing Chapel programs can be thought of as being single-realm executions in which variables associated with a realm are exposed to the user in the global scope. We expect that single-realm executions will continue

to be a common and important case, both for the user and for the compiler to optimize for. This is somewhat analogous to our current Chapel compiler’s support for specifying a single-locale execution for optimization purposes.

As a running example for this discussion, assume that the programmer wants to execute using three distinct realms: a 64-node Cray XMT (treated as one locale), its 16 SIO nodes (each a distinct locale), and a desktop Macintosh (also a single locale).

#### A. Realms within the Chapel Language

Since the compiler will need to generate distinct executables for different architectures, the user will need to specify the realms on which they want the Chapel program to execute at compile-time. This will be supported via a set of built-in parameters, similar to the constants supported in the current language definition to describe locales:

```
config param numRealms: int = 1;

param RealmSpace = [0..#numRealms];

config param realmTypes: [RealmSpace] string
    = getenv("CHPL_TARGET_PLATFORM");
```

The *numRealms* variable indicates the number of distinct machines on which the Chapel program will execute. If unspecified, *numRealms* will default to “1” and the compiler will execute on a single target architecture as it does today. The *RealmSpace* domain stores the index space for the target machines, just as *LocaleSpace* does for the locales in a traditional Chapel program. The *realmTypes* array stores a series of implementation-specific strings that uniquely represent the architectures being targeted. To implement the specified computation, the Chapel compiler will need to generate an executable for each distinct value in *realmTypes*.

Given these configuration parameters, a programmer wanting to invoke the Chapel compiler for the multi-realm case from our motivating example might use the following command line:

```
prompt% chpl -o myProg myProg.chpl --numRealms=3 \
    --realmTypes=("xmt", "xmt-sio", "darwin")
```

Note that the mechanisms for specifying configuration parameters can vary between Chapel implementations.

The user will be able to specify the number of locales per realm at execution time, much as they do for the *numLocales* variable today. This will be expressed using a configuration constant array of integers:

```
config const localesPerRealm: [RealmSpace] int;
```

Thus, to execute using the set of locales in our motivating example, the Chapel program would be launched as follows:

```
prompt% ./myProg --localesPerRealm=(1, 16, 1)
```

Just as Chapel currently supports a *locale* type and values for reasoning about the execution set of locales, it will also support a *realm* type and values, stored using a built-in array, *Realms*:

```

const Realms: [r in RealmSpace] realm
    = new realm(id=r, rtype=realmTypes(r));

```

Each realm will have its own private variables to describe its locale set, equivalent to those currently supported for single-realm executions—*numLocales*, *LocaleSpace*, and *Locales*. In this sense, the `realm` type can be thought of as being implemented using a record with the following structure:

```

record realm {
  const id: int;
  const rtype: string;

  const numLocales: int = localesPerRealm(id);
  const LocaleSpace = [0..#numLocales];
  const Locales: [LocaleSpace] locale = ...;
}

```

To support backwards compatibility for existing Chapel programs and to continue to make single-realm programming convenient for the user, when *numRealms* is found to have the value “1”, the compiler will introduce global variables *numLocales*, *LocaleSpace*, and *Locales* to describe the program’s target set of locales as a convenience.

Given these features, an `on`-clause for a multi-realm program might appear as follows:

```

on Realms(1).Locales(3) do foo();

```

For our running example, this statement would indicate that *foo()* should be executed on the 4th SIO node of the XMT. We will also extend `on`-clauses to take expressions of type `realm`, causing the statement to execute on locale 0 of that realm. Thus, the following statement would cause *bar()* to execute on the Macintosh in our running example:

```

on Realms(2) do bar();

```

We also need capabilities for realms equivalent to the *locale* and *here* concepts currently supported for locales. To this end, any variable expression can be queried with a built-in *realm* method to query the realm in which it is stored. Similarly, any task can access the built-in private constant *thisRealm* to query the realm in which it’s executing.

## B. Semantic Notes on Realms

Apart from the new concepts described in the previous section, we don’t anticipate changing Chapel’s semantics to deal with multiple realms. In particular, we will not impose semantic distinctions between realms to restrict operations to only act on data within their realm. This continues the Chapel theme of supporting a partitioned global namespace for the purposes of convenience, with the caution that programs which haphazardly access data across realm boundaries will be subject to the expected overheads of communicating between the distinct architectures. The specific overheads will obviously vary depending on how tightly coupled the target architectures are.

In the same vein, note that the locale values in a multi-realm program are all of a single locale type, permitting users to create collections of locales that span multiple compute resources. For example, a user can create a single array of

locales that describes all of the locales contained within all of the realms. In general, this might be set up as follows:

```

const totNumLocales = + reduce localesPerRealm;
var allLocs: [0..#totNumLocales] locale;

var lo = 0;
for r in Realms {
  const num = r.numLocales;
  allLocs[lo..#num] = r.Locales;
  lo += num;
}

```

Or, for the specific values in our motivating example, we could simply write:

```

var allLocs: [0..17] locale;

allLocs[0] = Realms[0].Locales[0]; // the XMT
allLocs[1..16] = Realms[1].Locales; // the SIOs
allLocs[17] = Realms[2].Locales[0]; // the Mac

```

Given such an array of locales, one could then do things like block distribute a domain between the locales of multiple realms:

```

var D: domain(1)
    distributed new Block1D(bbox=[1..m],
                           targetLocales=allLocs)
    = [1..m];

```

Whether or not doing such a thing is wise would depend on the differences in capabilities and latencies between realms.

## C. Multi-Realm Examples

The following Chapel program implements a simple multi-realm, multi-locale “Hello, world!” program:

```

coforall r in Realms do
  on r do
    coforall l in r.Locales do
      on l do
        writeln("Hello from locale#", here.id,
              " located in realm#", thisRealm.id,
              " named ", here.name);

```

In this code, a task is created per locale that prints out where it is executing using the built-in *here* and *thisRealm* constants. Equivalently, the statement in the inner loop could have been written:

```

writeln("Hello from locale#", l.id,
       " located in realm#", r.id,
       " named ", l.name);

```

Note that the statement “`on r do`” in the code above is not strictly necessary, but that using it causes each realm to spawn its own parallel tasks.

The next example demonstrates the use of the shared namespace across multiple realms:

```

// Program execution starts on realm 0, locale 0
// so checksum$ is stored on realm 0, locale 0
var checksum$: sync int = 0;

coforall r in Realms do
  coforall l in r.Locales do
    on l do
      checksum$ += r.id*1000 + l.id;

```

In this program, a task is created on each locale that increments a shared checksum variable by a value that is a function of its realm and locale ID. The checksum variable is a synchronized integer variable allocated on realm 0, locale 0 since that is where the program starts executing. This simple example would obviously result in a bottleneck and would be better implemented using reductions, but it demonstrates how tasks running on any realm or locale can access shared state declared in enclosing lexical scopes.

As a final example, consider the following code sketch as an illustration of using each realm to compute a distinct task suitable for its capabilities. We use a `cobegin` statement to launch off a number of parallel tasks, and have those tasks coordinate through shared state variables declared in a common enclosing scope.

```
var A: ... // declare shared state

cobegin {
  on Realms[0] do bigGraphComputation(A, ...);
  on Realms[1] do denseArrayComputation(A, ...);
  on Realms[2] do visualize(A, ...);
}
```

Alternatively, if the distinct phases of computation did not need to execute concurrently, the computation could be written to migrate sequentially from one realm to the next:

```
var A: ...

while (...) {
  on Realms[0] do bigGraphComputation(A, ...);
  on Realms[1] {
    var A1 = A; // make a local copy of A
    denseArrayComputation(A1, ...);
    A = A1; // copy result back
  }
  on Realms[2] {
    var A2 = A; // make a local copy of A
    visualize(A2, ...);
  }
}
```

These examples indicate just some of the ways that multiple realms can be used. Generally speaking, the realm is a completely composable concept within Chapel and may be used in a wide variety of ways.

#### D. Future Directions

A future direction that we would like to pursue for both multi-realm and single-realm programs is to establish the notion of an *execution context* to refer to the collection of locales on which the current task is executing. In traditional Chapel this has been a wishlist item, but it becomes more compelling in the presence of multi-realm execution for the purposes of code reuse. In particular, imagine that a developer has implemented a single-realm program that works well:

```
def myCode() {
  coforall loc in Locales do
    on loc do
      ...
}
```

Now the developer wants to incorporate this code to execute within one realm of a new multi-realm program that they are

writing. For example, perhaps the multi-realm program is a coupled model and the single-realm program is intended to be one of the components, executing within its own realm. The top-level control structure of the code can be written very cleanly:

```
cobegin {
  on Realms[0] do myCode();
  on Realms[1] do ...
}
```

but the code itself must be changed in order to support the multi-realm execution. In particular, the reference to the *Locales* variable has to be modified in order to refer to the locales of a particular realm:

```
def myCode() {
  coforall loc in thisRealm.Locales do
    on loc do
      ...
}
```

Changes like this are tedious but necessary under our current model. In order to better support code reuse between single-realm and multi-realm executions, imagine the introduction of a variable representing the execution context of a code segment as a collection of locales. For the purposes of this discussion, call the variable *currentLocales*. Any time that an on-clause's expression describes a realm or some other collection of locales, the value of *currentLocales* would be updated for the contained code to describe that set of locales.

Given such a feature, the original code could have been written:

```
def myCode() {
  coforall loc in currentLocales do
    on loc do
      ...
}
```

and could remain unchanged when the code was incorporated into a multi-realm program. Distributions like our *BlockID* distribution would be written to take *currentLocales* as the default target locale set rather than *Locales*.

This concept would also improve code reuse for single-realm executions since *Locales* could be carved into disjoint segments, each of which would have its own execution context. For example, imagine the high-level specification of an abstract coupled climate model:

```
cobegin {
  on Locales[0..4] do water();
  on Locales[5..10] do land();
  on Locales[11..15] do air();
}
```

In this code, each of the routines *water()*, *land()*, and *air()* would have its own execution context describing the set of locales specified by its enclosing on-clause. Thus, loops over *currentLocales* or distributions targeting *currentLocales* would span the subset of locales on which that component was executing rather than forcing the user to manage and target these collections of locales manually.

We consider this concept to be future work because we have not had sufficient time to vet it prior to publishing this report and because we believe that the currently proposed support is sufficient for supporting hybrid execution.

#### IV. SUMMARY

This paper proposes a new Chapel concept, the realm, to support a Chapel program's execution using hybrid compute resources. The goal of the realm is to continue the Chapel theme of supporting a global view of data and control flow across distributed memory architectures. By minimizing the changes to the language due to the introduction of realms, we believe we have met this goal while continuing to give the programmer appropriate abstractions for controlling where data is allocated and where tasks execute.