# Chapel: Cascade High-Productivity Language
# An Overview of the Chapel Parallel Programming Model*

Steven J. Deitz Bradford L. Chamberlain Mary Beth Hribar

Cray Inc.
Seattle, WA 98104
{deitz,bradc}@cray.com

**Abstract**

Chapel, which stands for **C**ascade **H**igh-**P**roductivity **L**anguage, is a new parallel programming language being developed at Cray Inc. as part of Cascade, a project at Cray funded by the DARPA **H**igh-**P**roductivity **C**omputing **S**ystems (HPCS) program. This paper will overview Chapel's parallel programming model, discuss its data- and task-parallel language abstractions, and show how the abstractions can be composed, allowing the programmer to write simple programs that, if written with today's most popular parallel programming facilities, would be complex, difficult-to-maintain codes.

## 1 Introduction

Chapel, **C**ascade **H**igh-**P**roductivity **L**anguage, is a new parallel programming language designed with productivity in mind. It is being developed at Cray Inc. as part of Cascade, a project at Cray funded by the DARPA **H**igh-**P**roductivity **C**omputing **S**ystems (HPCS) program. The goal of this program is to produce an advanced high-performance computing system that is highly productive for its users. The motivation behind this program is simple. It has become too difficult to program today's supercomputers.

To cope with this difficulty, Chapel targets both algorithm experimentation/development and production deployment, and it tries to bridge the gap between the two kinds of programs. To this end, Chapel borrows many language aspects from modern programming practice, and integrates them with high-performance parallel language abstractions.

The target machine for Chapel is a parallel computing system with an arbitrary number of homogeneous processors. Chapel is an explicitly parallel programming language so the programmer is responsible for identifying the available concurrency in code. However, the system manages the details of implementation, such as communication and data allocation, thus making programmers more productive.

This paper provides the following information:

- An overview of the Chapel parallel programming model.
- An enumeration of the language's abstractions for data- and task-parallelism.
- A discussion of how the data- and task-parallel abstractions can be composed, thus allowing the programmer to write simple programs that, if written with today's most popular parallel programming facilities, would be complex, difficult-to-maintain codes.

For more information on Chapel, refer to the specification [2].

## 2 Parallel Execution

Chapel supports an advanced parallel programming model that generalizes the programming model of CSP-like languages and libraries, *e.g.*, MPI [3], fragmented global address space languages, *e.g.*, Co-Array Fortran [4] and UPC [1], and global-view languages, *e.g.*, HPF [4] and ZPL [5].

*This work was funded in part by the Defense Advanced Research Projects Agency under its Contract No. NBCH3039003.

Chapel abstracts a parallel machine into units of locality, called *locales*, and units of execution, called threads. By distinguishing between threads and locales, Chapel is able to support and compose both data- and task-parallel programming abstractions.

On most large-scale systems, it is faster to access local memory than it is to access remote memory. Chapel's concept of locales allow programmers to optimize accordingly. A locale is a notion that has both associated data and computation, *i.e.*, both data and threads can be associated with a given locale.

The locale is mapped to some aspect of the machine in an implementation-dependent manner. For a cluster, for example, the locale may refer to a single processor or chip. For a cluster of shared memory nodes each with multiple processors, the locale may refer to the nodes, but could also refer to each processor within the nodes.

When a program is executed in Chapel, there is initially one thread running on one locale. Task-parallel abstractions can be used to spawn additional threads running on different locales or the same locale. Data-parallel abstractions can be used to compute across multiple locales with one conceptual thread.

# 3 Data-Parallel Abstractions

Chapel derives its data-parallel abstractions from ZPL and HPF. Its main data object from which data-parallelism is derived is the array, although arrays in Chapel are more general than in the most programming languages.

## 3.1 Arithmetic arrays

Chapel supports arithmetic arrays that are similar to arrays in languages such as Fortran but that provide enhanced functionality. As will be seen, a generalized array is Chapel's key data-parallel abstraction.

### 3.1.1 Domains

Like ZPL before it, Chapel distinguishes between the indices over which an array is defined and the data within that array by allowing the programmer to abstract them separately. A *domain* is an index set with no associated data. An array is declared over such a domain. For example, the following code declares a 2D domain over an n × n index set, another 2D domain over the interior indices, and an array of floating-point values over all the indices:

```
var D : domain(2) = [1..n, 1..n],
    InD : domain(2) = [2..n-1, 2..n-1],
    A, B : [D] float;
```

Domains benefit the programmer by abstracting the indices into a named set. This generally results in clearer, more concise code and often decreases the likelihood of tedious indexing and resultant errors. In the parallel context, domains provide the further benefit of allowing a distribution to be associated with an index set rather than a particular array of data.

Domains support iteration over their index set. For example, the following code assigns the interior elements of `B` to the interior elements of `A`:

```
for i, j in InD do
  A(i, j) = B(i, j);
```

### 3.1.2 Array indexing

Chapel provides a general means of accessing elements within arithmetic arrays. As seen in the example above, arrays can be indexed by integers where the number of integers is equal to the rank of the array. This is equivalent to basic array indexing in Fortran 77.

Arrays can also be indexed by tuples of integers where the size of the tuple is equal to the rank of the array. For example, the assignment of the interior elements of `B` to the interior elements of `A` can also be written as follows:

```
for i in InD do
  A(i) = B(i);
```

Chapel allows arrays to be indexed by domains as well. For example, we can also write the above computation as follows:

```
A(InD) = B(InD);
```

Chapel also allows an array to be indexed by *arithmetic sequences*. An arithmetic sequence is simply a sequence of integers given by a lower bound, an upper bound, and a stride. We already saw an arithmetic sequence in our definition of the domains above. As an example, we can assign the last column of `B` to the first column of `A` as follows:

```
A(1..n, 1) = B(1..n, n);
```

Parentheses are used for *zipper products* whereas square brackets are used for cross products. So the major diagonal of `A` is accessed by indexing into `A` as in `A(1..n, 1..n)` whereas all of the elements of `A` are accessed via `A[1..n, 1..n]`, or of course, simply by `A`. As will be seen, this is a general mechanism in Chapel that applies to function invocation as well.

### 3.1.3 Whole array operations

Arrays can be assigned to one another without indexing into the arrays to access their elements. For example, we can assign the elements in `B` to the elements in `A` by writing `A = B`. Whole array operations are implemented with sequence semantics and the sequences must conform to one another in shape, *i.e.*, must have the same rank and extent in each dimension. Thus a 1D array cannot be assigned to a 2D array even if the number of elements in each of the arrays are the same.

Scalar functions and operators can be applied to whole arrays as well. In this case, the scalar function is applied to each of the elements in the array. So for example `A + B` evaluates to an array containing the sums of the elements in `A` and `B`. Again, `A` and `B` must conform to one another in shape.

Whole array operations are data-parallel operations by definition. Conceptually there is one thread of execution and the operations are executed in parallel. *Distributions* allow the arrays to be stored on multiple locales, much like in HPF and ZPL.

### 3.1.4 Other types of arrays

In addition to arithmetic arrays, Chapel also provides sparse arithmetic arrays, associative arrays (which support hash tables), opaque arrays (which support arbitrary graphs), and product arrays (which support multidimensional arrays that are composed of any combination of Chapel arrays). In addition, arrays are extensible and can be defined by the Chapel programmer, discussed in Section 3.4.

## 3.2 Distributions

A distribution maps the indices of a domain to locales. Arrays declared over domains that are not distributed are stored on a single locale as is any other variable. For domains that are distributed, the array is stored on the locales specified by the distribution.

Iteration over a distributed domain executes the computation over the domain in the associated locale. For example in the code

```
for i in D do
  -- computation
```

the computation is executed on the locale that `i` is mapped to by the distribution of `D`. This affinity can be overridden by using the `on` statement as in the code

```
for i in D do on D.locale(i+1)
  -- computation
```

Here the computation is executed on the locale to which `i+1` is mapped. Note that `i+1` must be in the domain. The method `locale` defined on domains returns the locale that an index in that domain is mapped to.

Programmers can write their own distributions. Indeed, there are no standard distributions in Chapel although there is a provided library of distributions. Thus user-defined distributions will be able to achieve similar performance to provided distributions.

## 3.3 The `forall` statement

The `forall` statement is similar to the `for` statement except that it can be executed in parallel. It thus extends data-parallel semantics to arbitrary statements. For example,

```
forall i, j in InD do
  A(i, j) = B(i, j-1) + B(i, j+1);
```

assigns the sum of the neighboring elements in the rows of `B` to the elements in the interior of `A`.

## 3.4 Extensible Arrays

This subsection assumes elementary familiarity with object-oriented programming concepts. Chapel supports classes that are similar to classes in Java and C++, supporting an advanced notion of inheritance and dispatch.

As mentioned previously, Chapel augments its standard arrays by allowing programmers to create their own implementations of Chapel's generalized notion of array. These extensions are enabled by Chapel's object-oriented programming abstractions. Arrays are, with syntactic sugar, simply classes that Chapel programmers can define.

### 3.4.1 The value class or `record`

Most classes in Chapel are reference classes so variables of a class type store a reference to an instance of that class or a subclass. Chapel also supports value classes or *records*. Variables of a record type do not store references; instead, they store values from the time they are declared. When two such variables are assigned, the value is assigned, and the variables do not alias.

For example, the arithmetic array, declared with the syntactic sugar discussed in Section 3.1, is implemented as a record. Its declarations is written as follows:

```
record ArithmeticArray : Array {
  ...
}
```

#### 3.4.2 The `this` function

Classes that implement arrays must specify an indexing function. The `this` function tells how the object should be treated when it is used as a function, *i.e.*, when the object is used as a function. It is equivalent to overloading parentheses in C++.

For example, to implement a square arithmetic array record, a programmer could define the `this` function in a record as follows:

```
record SquareArray {
  var n : int;
  ...
  function this(i : int, j : int) {
    return data(i + j * n);
  }
}
```

In the code above, it is assumed that the elements are stored in a one-dimensional vector. The indexing function simply computes a position in that vector for each element.

Tuples and variable-length argument lists allow the programmer to create a rank-independent indexing function. A mechanism for setter functions lets the indexing function be used on the left hand side of an assignment.

#### 3.4.3 Scalar function promotion

*Scalar function promotion*, used as a mechanism to enable whole array operations in Section 3.1.3, is generalized to classes that implement a `this` iterator. It is similar to the `this` function in that it is used when iterating over "this" array. For example, we can add a `this` iterator to our square array above allowing it to be used in whole array operations:

```
iterator this {
  for i, j in [0..n-1, 0..n-1] do
    yield this(i, j);
}
```

# 4 Task-Parallel Abstractions

Chapel provides a rich set of high-level task-parallel abstractions. These concepts enable the user to control which portions of the code are executed concurrently and which portions are executed serially. Synchronization is abstracted through reading and writing to special types of shared variables.

## 4.1 The `cobegin` statement

A `cobegin` statement contains a list of statements that are executed concurrently. Each statement is executed by its own thread. For example, in the code

```
cobegin {
  x = analyze();
  y = evolve();
}
```

the functions `analyze` and `evolve` are executed concurrently.

Control continues after all of the statements in the `cobegin` block have been executed.

## 4.2 The `begin` statement

The `begin` statement executes a statement in a new thread. For example, in the code

```
begin x = analyze();
...
```

a new thread is used to execute the `analyze` function. The main thread continues immediately, *i.e.*, the `analyze` function is executed in parallel with the balance of the code.

## 4.3 The `serial` statement

The `serial` statement is used to control whether a parallel statement should be executed concurrently or should be serialized. For example, the following recursive sort function would be called in parallel until the size of the data is small, in which case it would be called serially:

```
function sort(A, low, high) {
  serial (high-low < 100) cobegin {
    sort(A, low, low+(high-low)/2);
    sort(A, low+(high-low)/2+1, high);
  }
  merge(A, low, high);
}
```

The expression following the `serial` keyword is evaluated and then the statement is evaluated regardless of the expression's value. If the value evaluates to

`true`, however, any dynamically encountered `forall` or `cobegin` statement is executed serially.

### 4.4 The `single` variable

Single assignment variables are declared by adding the keyword `single` at the beginning of a variable declaration. For example,

```
single var x : int;
```

declares a single assignment variable of type integer. Such variables may only be assigned once dynamically. Any read of the variable before it is assigned causes the thread to suspend execution and wait for the variable to be assigned.

For example, in the code

```
single var x : int;
begin x = analyze();
y = evolve();
... x, y ...
```

the `analyze` function is executed by its own thread while the main thread executes the `evolve` function. In the last statement, the main thread waits until the `analyze` function has finished executing and `x` has been written. Note that this particular example could also be written with a `cobegin` statement.

### 4.5 The `sync` variable

Synchronization variables generalize the single assignment variable to permit multiple writes. A synchronization variable is declared using the `sync` keyword instead of the `single` keyword.

Conceptually a `sync` variable is in one of two states. It is either *empty* or *full*. Before it is assigned, it is considered empty. When it is empty, threads that attempt to read it are suspended until it is full. When a thread does read it, it transitions to empty. When it is full, threads that attempt to assign to it are suspended until it is empty. When a thread does assign to it, it transitions to full. When multiple threads are suspended, one is chosen non-deterministically.

Synchronization variables allow a sequence of values to be communicated between threads using a single shared variable. They also can be used as building blocks for more traditional synchronization primitives such as semaphores and monitors.

Synchronization variables support a read and write method that allow a thread to read the variable but not transition to an empty state and assign to the variable but not transition to a full state.

## 5 Nested Parallelism

The data- and task-parallel abstractions of Chapel can be arbitrarily composed. For example, in the code

```
cobegin {
  forall ij in D do
    analyze(A(ij), B(ij));
  forall ij in D do
    C(ij) = evolve(A(ij), B(ij));
}
```

two conceptual threads are spawned to perform two data-parallel computations that may each involve all of the locales. This is a fairly common idiom for a program that a scientist may want to write, but it is very difficult to write it in today's parallel programming languages. Work on integrating task- and data-parallel abstractions is not new, and the desire to write codes like the above is the driving motivation.

In Chapel, task-parallel abstractions can be used within the data-parallel context as well. The following code illustrates how this might be done:

```
forall ij in D do
  cobegin {
    compute_produce(A(ij));
    compute_consume(A(ij));
  }
```

Chapel provides the ability to write complicated parallel implementations of an algorithm easily, *i.e.*, without managing all the details of communication, etc. Moreover, constructs are supplied to control locality and degree of concurrency for deep control of performance.

## 6 Conclusion

Chapel is a highly general parallel programming language that unifies abstractions for data and task parallelism, allowing these abstractions to be arbitrarily composed. By distinguishing between threads and locales, Chapel is more flexible than previously proposed parallel programming facilities.

## References

[1] W. W. Carlson, J. M. Draper, D. E. Culler, K. Yelick, E. Brooks, and K. Warren. Introduction to UPC and language specification. Technical Report CCS-TR-99-157, Center for Computing Sciences, Bowie, MD, May 1999.

[2] Cray Inc. *Chapel Specification (Version 0.4)*, February 2005.

[3] Message Passing Interface Forum. MPI: A message passing interface standard. *International Journal of Supercomputing Applications*, 8(3/4):169–416, 1994.

[4] R. W. Numrich and J. K. Reid. Co-Array Fortran for parallel programming. Technical Report RAL-TR-1998-060, Rutherford Appleton Laboratory, Oxon, UK, August 1998.

[5] L. Snyder. *Programming Guide to ZPL*. MIT Press, Cambridge, MA, 1999.