Five Powerful Chapel Idioms[*]

Steven J. Deitz Bradford L. Chamberlain Sung-Eun Choi David Iten Lee Prokowich

Cray Inc.

Abstract

The Chapel parallel programming language, under development at Cray Inc., aims to deliver high performance to more programmers with less effort than current practices. In this paper, we look at five idioms that illustrate the following points:

1. Chapel makes it easy to write simple data-parallel computations.
2. First-class, user-defined data distributions enable plug-and-compute functionality for distributed arrays.
3. Composable abstractions support tasks that can be both asynchronous and remote.
4. Chapel's data- and task-parallel abstractions support nested parallelism.
5. Support for local and remote transactions make distributed, multi-threaded programming easier.

1 Introduction

Chapel is a new parallel programming language being developed at Cray Inc. It aims to deliver high performance to more programmers with less effort than current practices. Specifically, the design of Chapel strives to improve the programmability of large-scale, distributed-memory systems while matching or exceeding the performance and portability of MPI and OpenMP, today's leading technologies.

Chapel increases programmer productivity by supporting general and *global-view* parallel programming while providing the programmer with control of locality and supporting mainstream language features. In support of general parallel programming, Chapel supports both task and data parallelism as well as their arbitrary composition, and it targets both fine-grain and coarse-grain parallelism available in a diversity of systems.

Chapel supports global-view programming that makes it far easier to write a particular, but common, style of program on distributed-memory systems. The central abstraction supporting global-view parallelism is the concept of a *global array*. A global array is an entity that can be treated as a whole even though its elements are partitioned across a system's locales. Chapel's support for global-view parallel programming is discussed in more detail in Sections 2 and 3.

Chapel supports programmer control of locality by allowing the programmer to explicitly control the affinity of both tasks and data to locales. Chapel supports a *locale* type allowing a programmer to map data and tasks to the locales. A locale is defined in an architecture-dependent way so that accesses to the memory associated with remote locales are more expensive than accesses to the memory associated with the local locale. The number of locales may be specified at program startup, and an array of the locales on which the program is running is part of the standard context of every Chapel program.

The five sections of this paper present examples of five powerful Chapel idioms that make parallel programming easier. These idioms illustrate some of the key features provided in Chapel, but are not meant to provide a complete introduction to Chapel. For a more complete discussion of the Chapel language, the reader is referred to the language specification [5].

Section 2 presents an idiom for data-parallel computations. This idiom makes it easy for Chapel programmers to write the large class of programs that exhibit data parallelism.

Section 3 presents an idiom for data distribution. Using this idiom, Chapel programmers can easily experiment with different data distributions on large arrays. Moreover, Chapel's data distributions may be user-defined.

Section 4 presents an idiom for asynchronous remote tasks. Two Chapel abstractions, one for asynchronous tasks and one for remote tasks, can be composed to support both asynchronous and remote tasks.

Section 5 presents an idiom for nested task and data parallelism. By allowing programmers to arbitrarily compose the abstractions in Chapel that introduce parallelism, there is no limit to the degree of parallelism that an algorithm can express.

Section 6 presents an idiom for transactions, which can be

[*]This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0001. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Defense Advanced Research Projects Agency.

either local or remote. Using such transactions makes distributed, multi-threaded programming easier.

The Chapel codes listed in this paper are compatible with version 1.1 of the compiler [4] as documented in version 0.795 of the language specification [5]. The one exception is `atomic` transactions, described in Section 6, which are not yet implemented.

2 An Idiom for Data-Parallel Computations

With the right abstractions, data-parallel codes are as easy to write as *a b c*. Consider a data-parallel computation over the elements of the three arrays `A`, `B`, and `C`. Idiom 1 shows the Chapel code for one such simple data-parallel computation featuring a *zippered* forall loop:

Idiom 1. Data-parallel computing via zippered iteration

```
forall (a,b,c) in (A,B,C) do
  a = b + alpha * c;
```

In a zippered forall loop over multiple iterators, the iterations proceed simultaneously such that if the body of the loop sees the *i*th element from one iterator, it sees the *i*th element from every other iterator too.

This particular parallel loop iterates over the elements of arrays `A`, `B`, and `C` element-wise such that the body of the loop sees the respective elements `a`, `b`, and `c`. On each iteration, the sum of `b` and the product of the scalar constant `alpha` and `c` is written to `a`.

This particular computation can also be written more concisely by applying assignment and operations to whole arrays:

```
A = B + alpha * C;
```

The implementation is comparable. Sidebar 1 illustrates the compiler transformations to implement this whole-array computation. If a loop's body contains more elaborate computations, they may not be easily expressed in this shortened form, so we use the explicit *forall* loop to illustrate the more general idiom.

This particular idiom may look similar to parallel loops in other languages, but it is more flexible and more powerful than the similar-looking support available with today's technologies due to its applicability to structures more varied than C- or Fortran-style arrays. To wit, parallel iteration in Chapel is supported over distributed arrays, associative arrays, sparse arrays, unstructured arrays, index sets, and user-defined iterators or data structures:

- **Data parallelism over distributed arrays.** This Chapel code will work even if the arrays are distributed across the locales of a large-scale, distributed-memory computer. Unfortunately, the most popular methods for parallel programming today make this simple computation more difficult to write than necessary. For example, MPI provides no support for holistically computing on distributed data structures, requiring the programmer to manage the low-level details of partitioning and synchronization. In Chapel, the algorithmic decisions related to distribution are separated from the computation using the abstractions described in Section 3.

- **Data parallelism over arrays with different distributions.** Parallel iteration over arrays with different distributions is supported. For example, array `A` may be distributed with a block distribution, `B` with a cyclic distribution, and `C` with a recursive bisection distribution. Of course, the increased communication costs will result in a less efficient loop than if the arrays all had the same distribution, but this can be weighed against the cost of redistribution when considering overall performance.

- **Data parallelism over associative, sparse, or unstructured arrays.** Associative, sparse, and unstructured arrays in Chapel support many of the same operations as the more standard arrays, including parallel iteration. For example, array `B` could be sparse or all of the arrays could be associative. There is a restriction that the arrays have the same shape as defined by the rank of the array

Sidebar 1. A high-level view of the Chapel compiler transformation of a whole-array statement. This sidebar illustrates, in Chapel, the steps that the compiler takes to lower whole-array statements to data-parallel loops. Steps 1 and 2 *promote* the product and sum operators over arrays `C` and `B`, producing equivalent code to introducing forall expressions around these operators. Step 3 collapses the nested forall loops to form a single forall expression. In the compiler, the collapse is a result of lowering the iterators that are created by the compiler to implement forall expressions. Step 4 expands the whole-array assignment into a zippered forall statement. Step 5, like step 3, collapses the nested forall loops, producing the code of Idiom 1.

Initial Code	`A = B + alpha * C;`
*1. Promotion of **	`A = B + (forall c in C do alpha * c);`
2. Promotion of +	`A = (forall (b, f) in (B, (forall c in C do alpha * c)) do b + f);`
3. Collapse of foralls	`A = (forall (b, c) in (B, C) do b + alpha * c);`
4. Expansion of =	`forall (a, f) in (A, (forall (b, c) in (B, C) do b + alpha * c)) do a = f;`
5. Collapse of foralls	`forall (a, b, c) in (A, B, C) do a = b + alpha * c;`

and the extent in each dimension. Note that the shape of associative arrays is incompatible with non-associative arrays and, consequently, associative arrays cannot be zippered with non-associative arrays.

- **Data parallelism without data.** Data parallelism does not actually require any data. A *domain* in Chapel is a first-class index set over which arrays are declared. Domains support parallel iteration on their own. For large iteration spaces, this allows programmers to use the simplicity of data parallelism without necessarily allocating data proportional to the size of the iteration space. Such loops are more akin to those supported in OpenMP although, in Chapel, the index sets can be distributed across multiple locales.
- **Data parallelism over user-defined iterators.** Chapel supports user-defined parallel iterators that can be attached to classes. Thus, for this computation, `A`, `B`, and `C` could be trees or graphs where each node is a class instance, allocated either locally or remotely, provided that this data structure supports parallel iteration.

By providing support for data-parallel computations, Chapel makes it easy to write this important category of codes. At the same time, Chapel provides the abstractions a programmer needs to write more complicated codes that are efficient.

3 An Idiom for Data Distribution

Chapel's data distribution abstraction allows an array or domain to be distributed across the memories of multiple locales. The computation can remain the same regardless of the specific distribution. This separation of concerns enables a *plug-and-compute* functionality that allows the distribution of an array to change from block to cyclic, for example, without any other code rewrites.

The type of the distribution must be known at compilation time. The compiler-generated code can change dramatically when switching distributions since it is similar to low-level code like Fortran and MPI.

Idiom 2 shows a declaration of a non-distributed domain `D`, a non-distributed array `A`, a distributed domain `BD` defined with the same index space as `D`, and a distributed array `BA`.

Idiom 2. Specifying data distributions

```
const D = [1..n, 1..n];
var A: [D] real;
const BD = D dmapped Block(boundingBox=D);
var BA: [BD] real;
```

The *dmapped* keyword creates distributed domains. This particular example uses the standard `Block` distribution defined with a single argument called `boundingBox`. This argument is used to partition the space of all indices across the system by evenly blocking the indices within the bounding box and mapping indices outside of the bounding box to the same locale to which the nearest index in the bounding box is mapped.

Since array `BA` is defined over the Block-distributed domain, it is a Block-distributed array. The declaration of the array uses the same syntax that is used for non-distributed arrays. As discussed in Section 2, computations over such arrays can use the same idioms as non-distributed arrays.

Distributions in Chapel can be user-defined [3]. This distinguishes Chapel from languages like HPF and ZPL where distributions were built into the system.

One important design point related to Chapel's standard distributions is that they are written entirely in Chapel. This ensures that switching to user-defined distributions will not necessarily result in a performance loss.

In addition to distributions, domains and arrays may be declared over *layouts*. A layout is like a distribution except the indices in a domain and the elements in an array are not partitioned across the locales of a system, but rather reside on a single locale. The keyword *dmapped* applies to both distributions and layouts, collectively called *domain maps*. When the *dmapped* keyword is omitted, the domain is declared over an implicit layout called the default layout.

The plug-and-compute nature of distributions supports writing code that is easier to read, maintain, and change because the partitioning is associated with the declarations instead of the computation.

4 An Idiom for Asynchronous Remote Tasks

Popular tools for parallel programming do not provide good support for creating asynchronous remote tasks. Chapel provides better support by distinguishing between tasks, an abstraction of computation, and locales, an abstraction of a node.

The *begin* statement is given by the following syntax:

begin `statement`

This compound statement creates a separate task to execute *statement*. The parent task continues executing immediately with the next statement. For example, the code

```
begin writeln("hello, world");
writeln("goodbye");
```

results in parallel execution of the two calls to `writeln`. The output of this program will either be

```
hello, world
goodbye
```

or

```
goodbye
hello, world
```

Since `writeln` is atomic, the letters will never become interleaved.

The *on* statement is given by the following syntax:

> **on** *expression* **do** *statement*

The execution of *statement* is transferred to the locale that stores *expression*. Once *statement* completes, control resumes on the original locale with the next statement. For example, the code

```
on A[i] do
  A[i] = 1;
A[j] = 2;
```

results in serial execution where the `i`th element of array `A` is assigned the value `1` on the locale that stores `A[i]`, and then the `j`th element is assigned the value `2` on the locale that the original task was executing.

The following idiom shows how the orthogonal natures of *on* and *begin* allow them to be naturally composed to invoke remote, asynchronous tasks:

Idiom 3. Composing 'on' with 'begin'

```
on loc do begin f();
```

In this code, the call of function `f` is executed on the locale given by `loc`, and control continues immediately on the original locale with the next statement.

It is possible to write a Chapel program that creates a single task to run on every locale at program starup (or at any other time). Such a task group can then be written in the SPMD

Sidebar 2. An implementation of the classic MPI ring example by fragmenting main in Chapel to subsume the MPI local-view programming model. In the ring example, the processors are arranged in a clock-wise ring. The last processor reads an integer value from standard input and sends this value to the next processor, namely, processor zero. Then each processor receives this value from the previous processor and sends it to the next processor. The token thus makes its way around the ring starting with processor zero.

The code in main uses a *for* loop coupled with a *begin* statement to create one task per locale and an *on* statement to run each task on its own locale. On each locale, the function `SPMDMain` is called. This function is similar to the main function in MPI codes because each task calls into this function.

The rest of the code is quite similar to an MPI program. First, compute the rank and size of the process (based on the locale ID in Chapel). Then if this is the last process, read an integer from standard input and send it to process zero. Then all processes receive and send the token as it comes around. The omitted functions `CHPL_Send` and `CHPL_Recv`, simplifications of `MPI_Send` and `MPI_Recv`, take two arguments: the data to send or receive and the locale to which or from which it is sent or received. These functions can be implemented using *on* statements and synchronization variables.

```
def main() {
  for loc in Locales do on loc do begin
    SPMDMain();
}

def SPMDMain() {
  var rank = here.id;
  var size = numLocales;

  if rank == size-1 {
    var token = read(int);
    CHPL_Send(token, 0);
  }

  var token: int;
  CHPL_Recv(token, (rank-1+size)%size);
  writeln("Locale ", rank, " has token ", token);
  CHPL_Send(token, (rank+1)%size);
}
```

When using Chapel's programming model with more regard to its global view of computation, this code can be greatly simplified. If the purpose of the ring code is to serialize control around the ring, then the preferred way of writing this computation in Chapel involves the serial *for* loop and the *on* statement to serialize control around the ring of locales. The following high-level Chapel code is equivalent to the above computation:

```
def main() {
  var token = read(int);
  for loc in Locales do on loc do
    writeln("Locale ", here.id, " has token ", token);
}
```

style that is typical of MPI. Sidebar 2 illustrates how such a program can be written in Chapel.

Chapel provides simple support for asynchronous, remote tasks that allow a programmer to write task-parallel computations for codes that do not admit efficient data-parallel solutions.

5 An Idiom for Nested Parallelism

The task- and data-parallel abstractions discussed above can be arbitrarily composed in Chapel. A common way to want to nest parallelism is to create two or more data-parallel statements that can run in parallel. The *cobegin* statement makes this straightforward in Chapel.

The *cobegin* statement is given by the following syntax:

cobegin { `statement-list` }

This statement creates a concurrent task for each statement in *statement-list*. Unlike *begin*, the *cobegin* statement results in code with more structure. Control waits to continue past the cobegin statement until each child concurrent task completes.

Idiom 4 uses *cobegin* to build on Idiom 1 to create two tasks that execute data-parallel statements:

Idiom 4. Invoking two data-parallel tasks

```
cobegin {
  forall (a,b,c) in (A,B,C) do
    a = b + alpha * c;
  forall (d,e,f) in (D,E,F) do
    d = e + beta * f;
}
```

The two data-parallel statements can be executed concurrently. This holds even if the arrays are distributed across a system's locales.

Task parallelism can be nested inside data parallelism as well. Consider the following Chapel code:

```
forall a in A {
  if a == 0 then
    begin a = f(a);
  else
    a = g(a);
}
```

Assume calls to function `f` take a very long time whereas calls to `g` are quick. In the above code, a new task is created when computing `f`. This allows the data-parallel task created by the *forall* loop to continue with the next iteration it owns without waiting for the result of `f`.

Data parallelism is a simple way to structure a program when it is applicable, but sometimes task parallelism is needed to achieve high performance, capture the natural parallelism of an algorithm, or take advantage of compute resources, *e.g.*, multicore chips and GPUs. By allowing data parallelism to be used within the context of a task parallel computation, and vice versa, Chapel supports the simplicity of data-parallel programming within the context of a general parallel programming system.

6 An Idiom for Remote Transactions

Chapel will eventually provide support for transactions via its *atomic* statement. Unlike the other idioms discussed in this paper, this work is not yet implemented, though an active collaboration is underway. Chapel support for atomic statements has been discussed in the literature[1, 6].

The atomic statement is given by the following syntax:

atomic `statement`

This statement creates an atomic transaction from *statement*. It is executed with transactional semantics such that:

- The statement appears to execute entirely or not at all.
- The statement appears to have completed in a consistent order with respect to other atomic statements.
- No variable assignment is visible to any other atomic statement until the statement has completely executed.

Note that this definition of an atomic statement is sometimes called *weak atomicity* because the semantics are atomic only with respect to the atomic statements. *Strong atomicity*, on the other hand, is defined so that an atomic statement is atomic with respect to the rest of the program. Whether Chapel will support weak or strong atomicity is still to be determined.

As an example, atomic statements can be used to implement critical sections that would otherwise require locks. Chapel also provides synchronization variables, which support locks. Synchronization variables are variables that have state associated with their implementation alongside their value. This extra state is binary and we adopt the tradition of calling the two states *full* and *empty*. When a synchronization variable is written, the state becomes full. When a synchronization variable is read, the state becomes empty. If the state is full, a write to the synchronization variable will block. If the state is empty, a read from the synchronization variable will block.

Given a synchronization variable `si$` of type `int`, declared as follows:

```
var si$: sync int;
```

A critical section can be implemented with the following code:

```
si$ = true;
critical();
si$;
```

If multiple tasks execute the above code, only one will make a call to function `critical` at a time because the other tasks will block when writing to `si$`.

This critical section can be implemented with an atomic statement as follows:

```
atomic critical();
```

A major advantage to this implementation is that atomic statements are composable whereas locks are not. If there is another atomic statement in the function `critical`, there is no problem. On the other hand, for the synchronization variable implementation, if there is another write to `si$` within the function `critical`, the program will deadlock waiting for a read of `si$`.

One disadvantage to atomic statements is that the code that can be put into an atomic statement cannot be any arbitrary code. For example, it is unlikely that system calls can be made in atomic statements.

Following the pattern of Idiom 3, atomic transactions can be executed on remote locales. Idiom 5 shows how easy remote transactions are to write:

Idiom 5. Demarcating a remote transaction

```
on A[i] do atomic A[i] ^= i;
```

The atomic statement in the above code is executed on the locale associated with array element `A[i]`. Although the write to the array element in this transaction is local to the task executing the atomic transaction, the read of `i` may be remote, which would make this transaction remote. Chapel is expected to support remote reads and writes in atomic statements, making it more powerful than other systems.

7 Conclusion

This paper examined five Chapel idioms, providing a glimpse of the Chapel features that support productive parallel programming. The first and second idioms, for data-parallel computing and data distributions, allow Chapel programmers to write high-level codes easily. Such codes have the potential to achieve high performance with minimal programming effort, and we are starting to see promising results [2].

The third idiom, for asynchronous, remote tasks, enables more irregular or complicated parallel control flow in an HPC code. By decoupling the notion of tasks from the notion of locales, Chapel supports a programming model with a richer feature set than today's technologies.

The fourth idiom, for nested parallelism, shows how the first three idioms can be used in a single program. In addition to improving the expressiveness of the language and allowing more levels of parallelism in the hardware to be targeted, this idiom is useful for writing a large program where most of the computation can be written using simple, data-parallel abstractions, but part of the computation requires more elaborate parallelism.

The fifth and final idiom, for local and remote transactions, has the potential to make task-parallel programming substantially easier. Programs that make use of atomic statements are significantly easier to read and maintain than programs that use locks or other means of synchronization.

Acknowledgments

We would like to thank everybody who has worked on the development of Chapel in the past and all of the early users of Chapel for their patience and insights.

References

[1] Robert L. Bocchino, Vikram S. Adve, and Bradford L. Chamberlain. Software transactional memory for large scale clusters. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2008.

[2] Bradford L. Chamberlain, Sung-Eun Choi, Steven J. Deitz, and David Iten. HPC Challenge benchmarks in Chapel. (available at http://chapel.cray.com), November 2009.

[3] Bradford L. Chamberlain, Steven J. Deitz, David Iten, and Sung-Eun Choi. User-defined distributions and layouts in Chapel: Philosophy and framework. In *USENIX Workshop on Hot Topics in Parallelism*, 2010.

[4] Cray Inc., Seattle, WA. *Chapel compiler chpl.* (Available at http://chapel.cray.com/).

[5] Cray Inc., Seattle, WA. *Chapel language specification.* (Available at http://chapel.cray.com/).

[6] Srinivas Sridharan, Jeffrey Vetter, and Peter Kogge. Scalable software transactional memory for global address space architectures. Technical report, ORNL FT Technical Report Series, 2009.