Chapel Tutorial Using Global HPCC Benchmarks: STREAM Triad, Random Access, and FFT*

(Revision 1.4 — June 2007 Release)

Bradford L. Chamberlain Steven J. Deitz
Mary Beth Hribar Wayne A. Wong
Chapel Team, Cray Inc.
chapel_info@cray.com

Abstract

This paper is a companion paper to *Global HPCC Benchmarks in Chapel: STREAM Triad, Random Access, and FFT*. Where that paper provides a high-level introduction to Chapel and the Chapel versions of the HPCC benchmarks, this paper walks through the codes in far greater detail, providing a tutorial introduction to Chapel using the benchmarks as a basis for discussion.

1 Introduction

Chapel is a new parallel programming language being developed by Cray Inc. as part of its participation in DARPA's High Productivity Computing Systems program (HPCS). The Chapel team is working to design, implement, and demonstrate a language that improves parallel programmability and code robustness while producing programs that have performance and portability that is comparable to or better than current practice.

In this paper, we present Chapel implementations of three of the global HPC Challenge (HPCC) benchmarks—STREAM Triad, Random Access (RA), and Fast Fourier Transform (FFT). We describe each implementation in-depth, both to explain how we approached the computation in Chapel, and to provide a tutorial introduction to Chapel concepts for new users.

The benchmark codes presented in this report compile and execute with our current Chapel compiler (version 0.5). We provide "*Compiler Status Notes*" to call out code excerpts in which our preferred code for a benchmark differs from the current version due to limitations in the current Chapel compiler.

2 Coding Conventions

In writing these codes, we used the HPC Challenge Class 2 Official Specification as our primary guide for defining the computations [4]. We studied and benefited from the HPCC reference implementations as well as the 2005 finalist codes, but typically chose to express the benchmarks in our own style rather than trying to mimic pre-existing implementations.

In particular, we chose names for our variables and subroutines that we found descriptive and appealing rather than trying to adhere to the naming conventions of previous implementations. The primary exception to this rule is for variables named by the written specification, such as m, n, and N_U. For these variables, we adopted the specification's names in order to clarify the ties between our code and the official description.

Several concerns directed our coding design (in roughly this order):

- faithfulness to the written specification
- ability to generate code with good performance

*This material is based upon work supported by the Defense Advanced Research Projects Agency under its Contract No. NBCH3039003.

- clarity and elegance of the solution, emphasizing readability over minimization of code size
- appropriate and demonstrative uses of Chapel features
- implementations that are generic with respect to types and problem parameters
- support for execution-time control over key program parameters like problem sizes
- ability to be tested in our nightly regression suite

Some of these motivations, particularly the last three, cause our programs to be slightly more general than required by the written specification. However, we believe that they also result in more interesting, realistic, and flexible application codes.

Structurally, we tried to keep the timed kernel of the computation in the program's `main()` procedure, moving other functionality such as initialization, verification, and I/O into separate routines. In the Random Access and FFT benchmarks, we also abstracted kernels of the computation into helper iterators and routines to improve abstraction and reuse.

Stylistically, we tend to use mixed-case names to express multi-word identifiers, rather than underscores. We typically use an initial lower-case letter when naming procedures and non-distributed variables, while domains, distributed arrays, and class instances start with an upper-case letter.

In our code listings, **boldface** text is used to indicate Chapel's reserved words and standard types, while "..." represents code that is elided in an excerpt for brevity.

In describing Chapel concepts, we occasionally provide template definitions for Chapel's code structure to illustrate the syntactic patterns. These templates use <*words-in-angle-brackets*> as logical placeholders for code that has not been specified; [text in square brackets] to indicate something that is optional; and {text | in curly brackets | separated by bars} to indicate a choice between multiple options. The language templates in this document are meant to be illustrative. They are intentionally incomplete and not intended as a replacement for the full Chapel language specification [2].

As mentioned previously, we approached these codes as we would for a large-scale parallel machine—thus they contain distributed data structures, parallel loops, and array-based parallelism. Since our current compiler only supports execution on a single locale and has only limited support for data parallelism, these constructs will necessarily fall into the degenerate cases of allocating data structures on a single locale and, in many cases, executing parallel loops and array statements using a single thread.

3 STREAM Triad in Chapel

The STREAM Triad benchmark asks the programmer to take two vectors of random 64-bit floating-point values, b and c, and to use them to compute $a = b + \alpha \cdot c$ for a given scalar value α. As with all of the HPCC benchmarks, the problem size for the vectors must be chosen such that they consume $1/4 - 1/2$ of the system memory. STREAM Triad is designed to stress local memory bandwidth since the vectors may be allocated in an aligned manner such that no communication is required to perform the computation.

This section describes our implementation of the STREAM Triad benchmark, which introduces several Chapel language concepts including modules, variable declarations, type definitions, type inference, looping constructs, subroutine definitions and invocations, domains, arrays, promotion, whole-array operations, array slicing, reductions, timings, randomization, and I/O.

3.1 Detailed STREAM Triad Walk-through

Appendix A contains the full code specifying our STREAM Triad implementation. In this section we walk through that code from top to bottom, introducing Chapel concepts as we go.

Defining and Using Modules All Chapel code is organized using *modules* which serve as code containers to help manage code complexity as programs grow in size. One module may "`use`" another, giving it access to that module's public global symbols. A module may be declared using a module declaration that specifies its name as follows:

```
module <moduleName> { ... }
```

Alternatively, for convenience in exploratory programming, if code is specified outside of a module declaration, the code's filename is used as the module name for the code that it contains. For example, all of the code in Appendix A is defined outside of a module scope, so if the code was stored in a file named `Stream.chpl`, it would define a module named *Stream*.

Lines 1–3 demonstrate how modules use one another:

```
use Time, Types, Random;

use HPCCProblemSize;
```

The first line uses three modules that are part of Chapel's standard library support: the *Time* module defines routines related to performing timings and reasoning about time; the *Types* module contains support for reasoning about Chapel's built-in types; and the *Random* module contains support for various random number generation routines.

The final line uses a module, *HPCCProblemSize*, that we wrote specifically for this study to compute and print the problem size that should be used by each benchmark. This code is common across the three benchmarks and was therefore placed in a module as a means of sharing it between the three programs, rather than replicating it and being forced to maintain multiple copies over time. The *HPCCProblemSize* module is listed in Appendix D and described in more detail in Section 6.

Variable Declarations Variable declarations in Chapel take the following basic form:

```
<variable-kind> <identifier> [: <definition>] [= <initializer>];
```

where *variable-kind* indicates the kind of variable being created, *identifier* specifies the variable's name, *definition* indicates the variable's "type", and *initializer* specifies its initial value. A variable's initializer may be omitted, in which case it will be initialized to a type-dependent value for safety (*e.g.*, "zero" for numerical types). Alternatively, a variable's definition may be omitted, in which case it will be inferred from its initializer.

Variable-kinds are specified in Chapel as follows:

```
[config] {param|const|var}
```

Working backwards, the `var` keyword indicates that a variable is truly "variable" and may be modified throughout its lifetime. The `const` keyword indicates that a variable is a constant, meaning that it *must* be initialized and that its value cannot change during its lifetime. Unlike many languages, Chapel's constant initializers need not be evaluatable at compile-time. The `param` keyword is used to define a *parameter*, which is a compile-time constant.[1] Parameter values are required in certain language contexts, such as when specifying a scalar type's bit-width or an array's rank. In other contexts, parameter values can be used to assert to the compiler that a variable's value is known and unchanging, for example to fold dead code away.

In this study, we typically declare variables to be as constant as possible, to make it clear to the reader and compiler which values will be modified and when. Changing all of our variable declarations to be `var` declarations would typically not affect the correctness of the programs contained in this report, and in most cases a mature compiler would be able to re-discover the symbols that are `const` or `param` by observing their uses.

Labeling a variable declaration with the optional `config` keyword allows its value to be specified on the command line of the compiler-generated executable (for `config const` and `config var` declarations), or on the command-line of the Chapel compiler itself (for `config param` declarations).

Variable declarations can also be specified in a variety of comma-separated ways, allowing multiple variables to share the same variable-kind, definition or initializer. These forms are fairly intuitive, so rather than define the syntax explicitly, we will demonstrate its use in our code as we go.

Line 6 defines the first variable of our Chapel code, *numVectors*:

```
param numVectors = 3;
```

This variable is used to represent the number of vectors that will be used by the program. Creating such a variable is unnecessary, but is good software engineering practice to eliminate "magic number" values from the code by using a symbolic name rather than embedding the value "3" throughout the source text.

In this declaration, the type of *numVectors* is elided, causing the compiler to infer it from the variable's initializer. Since the initializer "3" is an integer, the variable is inferred to be of Chapel's default integer type `int`, whose width is 32 bits. We define *numVectors* to be a parameter since its value is known at compile-time and is something that we won't want to change without greatly restructuring the code.

[1] The Chapel team remains somewhat uneasy about the use of the term "parameter" in this context due to its common usage to describe a subroutine's arguments; however, we have been unable to agree on a suitable replacement term and keyword. Please send any suggestions to chapel_info@cray.com.

Type Definitions Chapel supports the ability to create named type definitions using the `type` keyword. Line 7 demonstrates such a declaration, creating a named type *elemType* to represent the vector element type that we will use in this computation:

```
type elemType = real(64);
```

This statement defines the identifier *elemType* to be an alias for a `real(64)`—Chapel's 64-bit floating point type. The identifier *elemType* may be used to specify a variable's definition or anywhere else that a type is allowed. Like parameter variables, type definitions must be known at compile-time.[2]

STREAM Triad's Configuration Variables Lines 9–20 define the configuration variables for STREAM Triad, all of which are declared to be `const`:

```
config const m = computeProblemSize(elemType, numVectors),
             alpha = 3.0;

config const numTrials = 10,
             epsilon = 0.0;

config const useRandomSeed = true,
             seed = if useRandomSeed then SeedGenerator.clockMS else 314159265;

config const printParams = true,
             printArrays = false,
             printStats = true;
```

These declarations show how multiple variable declarations of the same variable-kind can share a single declaration statement by comma-separating them. Here, the grouping of the nine variables into four statements is arbitrary, chosen to group the variables roughly according to their roles—those that affect the computation, control the experiment, help with random number generation, and control I/O, respectively.

The first two lines declare configuration constants *m* and *alpha*, used to describe the problem size, m, and the scalar multiplier for Triad, α, respectively. The default value for *m* is initialized using a call to the routine `computeProblemSize()` defined in the *HPCCProblemSize* module. This routine takes as its arguments the element type being stored and the number of arrays to be allocated and returns the problem size that will fill the appropriate fraction of system memory as an `int` (see Section 6 for more details). Since *m*'s declaration does not specify an explicit type, it is inferred from `computeProblemSize()`'s return type, making *m*'s type `int`.

The default value for *alpha* is the value 3.0, chosen arbitrarily. Since *alpha*'s declaration contains no definition, and the literal "3.0" represents a real floating point value in Chapel, *alpha* is inferred to be of the default (64-bit) floating point type `real`.

The next two lines declare *numTrials* and *epsilon*—*numTrials* represents the number of times to run the Triad computation and has the default value of "10" as indicated in the specification; *epsilon* represents the threshold value of the infinity-norm computed during the verification stage in order for the test to pass. Since there should be no floating point precision differences between our timed and verification computations, we specify a default value of "0.0" for *epsilon*. These variables are inferred to be of type `int` and `real`, respectively.

The next two lines define variables used to control the pseudo-random initialization of the vectors. The variable *useRandomSeed* indicates whether the pseudo-random number generator should be seeded with a "random" value or one that is hardcoded into the program for verification purposes. It is initialized with the boolean literal `true` and is therefore inferred to be a variable of Chapel's boolean type, `bool`. The variable *seed* is used to store the seed value itself and is initialized with a conditional expression based on the value of *useRandomSeed*—if it is false, it uses the hardcoded literal value "314159265"; otherwise, it uses an object named *SeedGenerator* from the *Random* module that can generate seeds using a variety of methods. Here, the invocation of the 0-argument *clockMS* method indicates that the seed should be generated using the milliseconds value from the current time of day. This method returns a 64-bit integer (`int(64)`), causing the literal value "314159265" to be coerced to an `int(64)` and the type of *seed* to be inferred to be an `int(64)`.

The final three lines define three boolean configuration constants—*printParams*, *printArrays*, and *printStats* that control the printing of the program parameters, the arrays (for debugging purposes), and the timing/performance statistics, respectively.

[2] We have recently flirted with the idea of supporting a `config type` which would allow a type to be specified on the compiler's command-line, much as a `config param` allows parameter values to be specified. For instance, if *elemType* had been defined to be a `config type` in this example, our STREAM Triad implementation could be compiled for a variety of vector element types without modifying the source text at all.

As described previously, any `config const` can have its default value over-ridden on the command-line of the compiler-generated executable. Thus, the user can specify different values for any of the variables described in this section for each execution of the program. For instance, our nightly regression testing system uses the following command-line when testing this program:

```
Stream --m=8 --printArrays=true --printStats=false --useRandomSeed=false
```

This set of assignments sets the problem size to be 8 (so that it runs quickly), turns on the printing of the arrays (in order to check the values being stored in the arrays), turns off the printing of the statistics (since they are dependent on timings and therefore differ from run to run), and uses the hard-coded random seed (to ensure deterministic results).

Entry Point All Chapel programs define a single subroutine named `main()` that specifies the entry point for the program. A Chapel program's entry point is conceptually executed by a single logical thread, in contrast to the Single Program, Multiple Data (SPMD) models used by most parallel programming languages today. The `main()` subroutine for our program is defined on lines 23–41, and takes the following form:

```
def main() {
  printConfiguration();
  ...
}
```

The keyword `def` is used to define a new subroutine in Chapel. Here, we are defining a subroutine named "main" that takes no arguments and has an inferred return type (inferred to be "void" since it will be seen to contain no `return` statements).

The curly braces define the body of `main()` and can contain declarations and executable statements. Here we show the first statement which calls a 0-argument subroutine to print out the program's problem size and parameters (described below). A subroutine may be called in Chapel before it is defined (as this one is) without requiring a prototype as in many traditional languages.

Domain Declarations In Chapel, a *domain* is a first-class language concept used to represent an index set. Domains can be thought of as describing the size and shape of an array, but without any associated data values[3] In this report, all domains are *arithmetic*, which means they store indices that are integers or tuples of integers. Though not used in this paper, arithmetic domains may also store sparse or strided index sets. Chapel has other domain types that can be used to store indices of *anonymous types* or arbitrary value types.

The syntax for specifying a domain's definition is as follows:

```
domain [ ( <domain-args> ) ] [ distributed [ <dist-obj> ] [ on <target-locales> ] ]
```

The `domain` keyword indicates that a domain is being declared. The optional *domain-args* arguments parameterize the domain. For example, arithmetic domains are parameterized by their rank. If this information is not provided, the compiler will try to infer it from the domain variable's initializer.

By default, a domain's index set is stored locally in the memory of the locale on which the current thread is executing. However, domains may also be declared to be distributed across locales using an optional *distribution clause*, specified using the `distributed` keyword. The distribution clause takes optional *dist-obj* and *target-locales* expressions which specify how the domain's indices are to be distributed and to which locales.

The *dist-obj* specification is an instance of a *distribution class*—either from Chapel's standard distribution class library or written by the user. If it is omitted, the programmer is specifying that the domain should be distributed, but leaves the choice of a specific distribution class to the Chapel compiler and runtime. The *target-locales* specification is a collection (*e.g.*, an array) of locales to which the distribution should map the domain's indices. If omitted, the indices are distributed between all locales on which the program is running.

Our STREAM Triad implementation creates one named distribution to describe the index space for the vector computation—*ProblemSpace*. It is declared on line 26 as follows:

```
const ProblemSpace: domain(1) distributed(Block) = [1..m];
```

This declaration specifies that *ProblemSpace* is a constant domain, indicating that its index set will not change after it is initialized. The *domain-args* value of "1" indicates that it is a 1D arithmetic domain, meaning that its indices are simple integer values. The initializer is a 1D arithmetic domain literal indicating that *ProblemSpace* should represent the index set $\{1, 2, \ldots, m\}$

[3]Chapel's domains are a generalization of ZPL's *region* concept[1].

We want the vectors that will be defined by the *ProblemSpace* domain to be distributed across the memories of all the locales in our execution set, so we specify a distribution clause, leaving off the *target-locales* specification. We indicate that *ProblemSpace* should be distributed using the *Block* distribution from Chapel's standard library. This distribution maps consecutive blocks of either $\lfloor m/numLocales \rfloor$ or $\lceil m/numLocales \rceil$ indices to each locale.

A key property of a domain's distribution is that it does not affect the Chapel program's semantics, only the implementation of its domains and arrays, and therefore its performance. In this benchmark, we chose the *Block* distribution due to: (*i*) its simplicity, (*ii*) the fact that the problem is statically load-balanced by nature (assuming homogeneous locales), and (*iii*) the fact that it produces large contiguous index subsets, supporting efficient loop and array access idioms.

A final note is that regular arithmetic domains like *ProblemSpace* only require $\Theta(1)$ space to store their $\Theta(m)$ indices. $\Theta(m)$ storage is allocated only for arrays declared using this domain. For a *Block* distribution, this results in $\Theta(m/numLocales)$ memory per locale.

Array Declarations Array definitions are expressed in Chapel using the following pattern:

```
[ <domain-spec> ] <element-def>
```

The *domain-spec* is a named or anonymous domain which specifies the index set over which the array will be defined. The *element-def* specifies the element type to be stored for each index in the domain. The *element-def* may be another array, supporting arbitrary array compositions.

The vectors for the STREAM Triad computation are declared in line 27:

```
var A, B, C: [ProblemSpace] elemType;
```

This declaration statement uses comma-separated identifiers to declare multiple variables of the same type. *A*, *B*, and *C* are declared to be arrays whose indices are defined by the *ProblemSpace* domain and whose elements are of type *elemType* (our symbolic alias for `real(64)`).

As a result of this declaration, each locale will allocate memory for a sub-block of each of the three arrays, corresponding to its subset of *ProblemSpace*'s index set as defined by the *Block* distribution. Since the arrays are not explicitly initialized, each element will be initialized with its type's default value for safety ("0.0" for these `real(64)` elements). A compiler that can prove to itself that every array element will be assigned before it is accessed may optimize this default initialization away. Alternatively, a performance-conscious programmer may remove this safety belt by requesting that the array not be initialized, either in its declaration or on the compiler command-line.

The next statement, line 29, calls a subroutine `initVectors()` to initialize arrays *B* and *C*:

```
initVectors(B, C);
```

By default, arrays are passed to subroutines by reference in order to avoid expensive array copies and temporaries. This subroutine is defined and discussed later.

The following statement, line 31, declares another array to store the timings collected for each of the *numTrials* executions of the Triad computation:

```
var execTime: [1..numTrials] real;
```

In this declaration, rather than using a named domain to represent the indices $1, 2, \ldots, numTrials$, we simply give the domain's definition in-line (note that the redundant square brackets can be omitted). Our coding philosophy is to name domains only when they are distributed or used several times within a program. For this code, there is no need to distribute the *execTime* array, and the domain will only be used once more (to control the for loop described in the next section), so we refer to it anonymously.

For-Loop Statements For loops are expressed in Chapel using the following syntax:

```
{ for | forall | coforall } [ index-vars in ] iterator-expr [ do ] <body>
```

The *iterator-expr* term generates the values over which the loop will iterate. In practice, it can be a *range*, a domain, an array, an *iterator* (defined later), or a product of these things. Loops that start with the `for` keyword are sequential, causing their iterations to be executed one at a time. When users specify a loop with the `forall` keyword, they are asserting that all of the loop's iterations can be executed concurrently. The decision as to how many threads (or other parallel resources) should be used to implement the loop's iterations is left to the compiler, runtime, and/or iterator author by default. Loops specified using the `coforall` keyword are used to assert that a distinct parallel task should be created for each iteration of the loop.

The optional *index-vars* term declares index variables to store the values generated by the *iterator-expr*. These variables are local to the loop body and cease to exist after the loop terminates. The types of these index variables are typically inferred using the values generated by the *iterator-expr*, though their types may also be specified explicitly if desired.

The *body* of the for loop defines work to be performed for each iteration as in most languages. If the body is a compound statement, the `do` keyword may be omitted.

Our STREAM Triad implementation uses a for-loop on lines 33–37 to perform the *numTrials* experiments:

```
for trial in 1..numTrials {
  ...
}
```

We use a `for` loop because we want each trial to complete before the next one starts. This loop's *iterator-expr*—"`1..numTrials`"—defines a *range* in Chapel, similar to those used to express the initializer for the *ProblemSpace* domain. The variable *trial* is declared by this statement and takes on the values $1, 2, \ldots, numTrials$, one at a time. Because *numTrials* is a variable of type `int`, the range also describes `int` values and therefore *trial*'s type will be inferred to be `int`.

Timings Chapel's standard *Time* module contains support for time-related computation including a *Timer* class that has start, stop, and reset capabilities like a stopwatch. For these benchmarks, all timings can be computed trivially by taking the difference between two times on any reference clock. Thus, for this study we use a routine from the *Time* module called `getCurrentTime()` which returns the elapsed time since midnight in seconds as a `real` value (an optional argument can be passed in to request that the time be returned in other units such as microseconds, minutes, or hours).

The timing calls for STREAM Triad occur in lines 34 and 36:

```
const startTime = getCurrentTime();
...
execTime(trial) = getCurrentTime() - startTime;
```

The first line here checks the current time and stores it in a constant named *startTime* which is inferred to be of type `real`. The second line checks the current time again, subtracts *startTime* and stores the result in the $trial^{th}$ element of the *execTime* array.

This is the first random access into an array that we have seen so far. All array accesses are dynamically bounds-checked if the compiler cannot prove that the index values are in-bounds at compile-time. One interesting benefit of replacing the two instances of `1..numTrials` with a domain declaration is that it would allow the compiler to trivially prove that all accesses to *execTime* are in-bounds since the same domain would be used to define both the array and the loop bounds—we will see and describe examples of this in further detail in subsequent benchmarks. For a case this simple, even a modest compiler ought be able to eliminate the dynamic bounds check without any trouble. Even if it doesn't, this is a small enough loop that it would not greatly impact the program's overall performance (and does not affect the timed portion of the benchmark). As with array initializations, programmers who are more concerned with performance than with safety can turn off dynamic array bounds checking via compiler flags or pragmas. In Chapel we turn bounds checking on by default because we have found it to be the number one cause of user bugs that were blamed on the compiler in our previous work with ZPL.

Due to an interesting piece of esoterica that we'll skip past here, array accesses using scalar indices can be expressed in Chapel using either parentheses (as shown here) or square brackets (*e.g.*, `execTime[trial]`). In this study, we adopt the convention of using parentheses for scalar array indexing and square brackets for array slicing.

Triad Computation The Triad computation is expressed using a single statement in line 35:

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

This line uses Chapel's whole-array syntax to express computation on the elements of arrays *A*, *B*, and *C in toto*. Chapel allows scalar operators and subroutines to be *promoted* across the elements of data aggregates such as arrays. For example, the multiplication operator ($\star$) is technically only defined for Chapel's scalar types rather than the scalar and array operands used in this statement. The promotion of multiplication causes *alpha* to be multiplied by each element in array *C*, yielding a virtual array result of the same size and shape. Similarly, the addition operator (+) is promoted across the values in *B* and the result of $\alpha \cdot C$, adding them in an elementwise or *zippered* manner. In this case the operands trivially conform with one another since *B* and *C* are defined using the same domain. More generally, promotion of operators and subroutines in this zippered manner is legal as long as the arguments have the same

size and shape or are scalars. The final promotion occurs in the assignment of the virtual result array representing $B + \alpha \cdot C$ to A, which promotes the scalar assignment operator, copying elements in an elementwise manner. Note that although we refer to the results of the subexpressions in this statement as "arrays", the Chapel compiler will ultimately implement this statement without allocating any intermediate array temporaries.

Expressions using whole-array syntax are implicitly parallel in Chapel, meaning that the statement above is equivalent to the slightly more verbose:

```
forall i in ProblemSpace do
  A(i) = B(i) + alpha * C(i);
```

For such whole-array operations, each locale computes the operations that are local to it in parallel, potentially using additional intra-locale parallelism (such as multiple processors, cores, or hardware support for vectorization or multithreading) to perform its local work in parallel.

Because *A*, *B*, and *C* are all declared using the same domain, *ProblemSpace*, the compiler can trivially determine that no communication is required to execute this statement. Conceptually, the scalar *alpha* is stored in the memory of the locale on which the thread executing `main()` began. However, since it was declared to be `const`, it can be replicated in each of the locale's local memories. If the user were to override the default value on the executable's command-line, it would require a broadcast at program startup time to synchronize the various copies. If the user was concerned about this cost, they could either resist the temptation to override the default, or remove the `config` specification and declare *alpha* to be a `param` or `const`.

Wrapping up main() The `main()` subroutine is concluded by two more subroutine calls on lines 39–40:

```
const validAnswer = verifyResults(A, B, C);
printResults(validAnswer, execTime);
```

The first subroutine takes arrays *A*, *B*, and *C*, verifies that *A* contains the correct result using an alternative implementation of Triad, and returns a `bool` indicating success or failure. The second routine takes the result of the first routine and the array of execution times and prints out a summary of the results. Both of these subroutines are defined and described below.

I/O Now we turn to the definitions of the helper subroutines that were invoked by `main()`. The first of these is `printConfiguration()` which is defined in lines 44–49:

```
def printConfiguration() {
  if (printParams) {
    printProblemSize(elemType, numVectors, m);
    writeln("Number of trials = ", numTrials, "\n");
  }
}
```

The body of `printConfiguration()` is guarded by a conditional statement whose test checks the value of the `config const` *printParams*. Conditionals in Chapel are similar to those in other modern languages. The first statement of the conditional calls a helper routine defined in the *HPCCProblemSizes* module which prints out some information about the problem size, taking in *elemType*, *numVectors*, and *m* as arguments.

Console I/O is expressed in Chapel using variable-argument `write()` and `writeln()` routines defined by its standard libraries. Each routine prints out its arguments to the console by converting them to strings and writing them out in the order listed. The user can also specify formatting conventions to use when converting the arguments to strings (not used in these benchmarks). The `writeln()` routine prints a linefeed to the console after processing all of its arguments. Linefeeds can also be generated using the standard "`\n`" character as in this example. In multithreaded contexts, the `write()` and `writeln()` statements are implemented to execute atomically with respect to those of other threads, to ensure that output from multiple threads is not interleaved at finer than the statement level.

Executing the `writeln()` statement in this subroutine would result in the following being printed to the console:

```
Number of trials = 10
(blank line)
```

Though not used in these benchmarks, Chapel also supports console input via a corresponding `read()` routine as well as file I/O using versions of these routines that operate on files.

Subroutine Arguments The next routine, defined on lines 52–62, initializes vectors *B* and *C* using Chapel's library support for pseudo-random number generation. It is the first subroutine we have defined that takes arguments:

```
def initVectors(B, C) {
  ...
}
```

As with variable declarations, formal arguments in Chapel may either be explicitly typed or not. In these benchmarks, we tend to elide the types of formal arguments, either to make the subroutines generic across multiple types, or simply in favor of brevity. In such cases, the types of the formal arguments are obtained from the actual arguments passed in at the routine's callsites, potentially cloning the subroutine for different argument type signatures.

In many cases, providing formal argument types helps document a subroutine's expected interface, and Chapel has many concepts for doing so including explicit typing, *type arguments*, *type query variables*, and *where clauses*. This topic is largely beyond the scope of this report however, so we ask that the reader adopt an exploratory programming mindset for the time being and bear with our typeless arguments.

Subroutine arguments can be specified with *argument intents* indicating whether the argument is a constant for the duration of the subroutine, or whether it is to be copied in and/or out on entry/exit. A formal argument with no specified intent, as in this subroutine, is treated as a `const` argument for most argument types, meaning that it may not be modified for the duration of the subroutine. Array and domain arguments are the primary exception to this rule. An array argument's elements or a domain argument's index set can be modified within a subroutine, and those modifications will be directly reflected in the actual argument. As mentioned previously, this is to avoid the nasty surprise of having array copies inserted automatically by the compiler. In addition, class instances passed to arguments with blank intents have their *reference* treated as a constant for the subroutine's duration, allowing the object's members to be modified. While these type-specific interpretations of a blank argument intent may seem somewhat confusing at first, they are chosen to support the principle of least surprise.

Randomization The body of the `initVectors()` subroutine is defined on lines 53–61:

```
var randlist = RandomStream(seed);

randlist.fillRandom(B);
randlist.fillRandom(C);

if (printArrays) {
  writeln("B is: ", B, "\n");
  writeln("C is: ", C, "\n");
}
```

This routine starts by constructing an instance of the *RandomStream* class which is defined in Chapel's standard *Random* module. New objects are constructed in Chapel simply by naming the class to be instantiated and supplying its constructor arguments in an argument list. Here, we capture a reference to the new object in a variable named *randlist*. The *RandomStream* constructor takes our *seed* value as its constructor argument, though the seed may also optionally be omitted causing the constructor to generate one using the *SeedGenerator* class.

The *RandomStream* class implements a conceptual stream of random numbers that may be traversed via an iterator or used to *fill* arrays. Our current implementation of the *RandomStream* class implements a linear congruential generator. In future versions of the *Random* module, we will be supporting a broad palette of parallel pseudo-random number generation techniques.

The next two lines use the *randlist* object to fill arrays *B* and *C* with pseudo-random values. In a parallel implementation, filling each array would be performed in parallel, with each locale filling in the values of its local portion of the array.

The final statements in this subroutine print out the arrays if specified by the `config const` *printArrays*. As these lines show, arrays can be written out using the `writeln()` subroutine like other expressions, which causes their values to be printed to the console in a formatted, row-major order by default. Console I/O only permits sequential writes by its nature, but when large arrays are written to files, their elements are typically written cooperatively in parallel using multiple locales, random file accesses, and potentially a parallel prefix computation for text-based output.[4]

[4]The user can also express parallel file I/O explicitly in Chapel using standard parallel concepts such as a distributed array of files and forall loops.

Forall Expressions and Reductions The next subroutine, `verifyResults()` is defined on lines 65–71:

```
def verifyResults(A, B, C) {
  if (printArrays) then writeln("A is: ", A, "\n");

  const infNorm = max reduce [i in A.domain] abs(A(i) - (B(i) + alpha * C(i)));
  ...
}
```

This routine starts by printing array *A*, which has now been computed, much as it printed *B* and *C* in the previous subroutine. Note the use of the keyword `then` to define a conditional whose body is not a compound statement.

The next line recomputes the Triad computation, simultaneously computing the infinity norm between the two approaches using a *reduction*. This statement is somewhat long, so let's take it piece by piece:

Reductions in Chapel collapse a data aggregate along one or more of its dimensions using a specified operator, resulting in a single result (for a *full reduction*) or a subarray of results (for a *partial reduction*). Reductions are defined using an operator to collapse the elements, and this operator is typically commutative and associative in practice. Chapel supports a number of built-in reductions such as `+`, `*`, `max`, `min`, and logical and bitwise operations. Users may also define their own reduction operators [3] though that is beyond the scope of this report. Reductions over distributed arrays are typically computed by having each locale compute its contribution independently and then combining the results using tree-based communication. In our reduction expression we use a `max` reduction to compute the infinity norm.

The data aggregate being reduced is an expression that is prefixed by "`[i in A.domain]`". This is a *forall expression*, which is similar to the forall-loop defined earlier except that its body is an expression and syntactically it can occur at the expression level.[5] This forall expression loops over the indices described by *A*'s domain, referenced using the 0-argument method, `domain`. It refers to the indices using an iteration variable, *i*, that is local to this expression.

The forall expression's body computes $B_i + \alpha \cdot C_i$ and subtracts the result from A_i. It then computes the absolute value of the result using the `abs()` function that is supplied with Chapel's standard math libraries and overloaded for its scalar types.

Note that this statement could also have been written simply as:

```
const infNorm = max reduce abs(A - B + alpha * C);
```

We avoided this form simply because it did not seem "different enough" from the original Triad computation to be considered a good verification, even though it describes the same computation.

As a final note, forall expressions can also be used at the statement level, providing a third possible expression of the original Triad computation:

```
[i in ProblemSpace] A(i) = B(i) + alpha * C(i);
```

Subroutine Return Types The final statement of `verifyResults()` is the return statement on line 70:

```
return (infNorm <= epsilon);
```

This statement simply returns a boolean value indicating whether or not the infinity norm exceeds *epsilon*. Chapel allows subroutine return types to be elided just like most type specifications, in which case the return type is inferred from the expressions in the routine's `return` statements. In this case, the expression is of type `bool` and therefore the return type of the subroutine is `bool`. This could have been specified explicitly as follows:

```
def verifyResults(A, B, C): bool { ... }
```

Printing the Results and Array Slicing The final subroutine for the STREAM Triad benchmark computes and prints out the results, and is defined on lines 74–88:

[5] Note the syntactic correlation between the forall expression and the array type definition syntax, "`[D] elemType`", which can be read as "for all indices in domain *D*, store an element of type *elemType*."

```
def printResults(successful, execTimes) {
  writeln("Validation: ", if successful then "SUCCESS" else "FAILURE");
  if (printStats) {
    const totalTime = + reduce execTimes,
          avgTime = totalTime / numTrials,
          minTime = min reduce execTimes;
    writeln("Execution time:");
    writeln("  tot = ", totalTime);
    writeln("  avg = ", avgTime);
    writeln("  min = ", minTime);

    const GBPerSec = numVectors * numBytes(elemType) * (m/minTime) * 1.0e-9;
    writeln("Performance (GB/s) = ", GBPerSec);
  }
}
```

Happily, this routine uses almost no new concepts. It uses a conditional expression within a `writeln()` call to indicate whether or not the computation was successful; it uses a conditional to guard the printing of the statistics; it uses reductions over the *execTimes* array to compute various timing statistics; it prints those statistics out; it computes the Gigabytes per second performance metric and prints that out.

One small note on this routine is that some implementations of the STREAM benchmark omit the initial trial from their timing statistics in order to allow things to get warmed up. This can be expressed easily in Chapel by using *array slicing* notation to describe the data aggregate provided to the reduction. Array slicing is expressed in Chapel by indexing into an array using a domain expression, named or anonymous. Thus, we could compute the alternative timing statistics using:

```
const totalTime = + reduce execTimes[2..numTrials],
      avgTime = totalTime / (numTrials - 1),
      minTime = min reduce execTimes[2..numTrials];
```

Array slicing also suggests a fourth way to express the original Triad computation:

```
A[ProblemSpace] = B[ProblemSpace] + alpha * C[ProblemSpace];
```

3.2 STREAM Triad Highlights

This section summarizes the STREAM Triad walkthrough by pointing out a few salient details about our code:

- Note the heavy use of generics and type inference in our code. Explicit types are only used in four places: to define the symbolic name *elemType*; to declare the distributed domain *ProblemSpace*; to declare the vectors *A*, *B*, and *C*; and to define the array of timings, *execTime*. Moreover, by changing one line of code—the specification of *elemType*—the entire computation will work as written on vectors of various numeric element types of any size—integers, floating point values, and complexes.
- Keep in mind that while we chose to use a very non-typed style of coding, the language fully supports specifying the type of every variable, argument, and return type. We conceive of most programs as starting out in an exploratory, sketchy manner as we have demonstrated here, and then having additional type specifications and constraints added as they evolve into production-grade codes. We have talked about adding a flag to the compiler that would cause it to convert a user's non-typed Chapel code into a fully-typed program in order to aid with this process. As an intermediate step, one could also conceive of a compilation mode in which the programmer could query the type of a specific symbol via a command-line flag, or have the compiler print out a table of symbols and their inferred types as a sanity check.
- Note that Chapel's global view made the management of the distributed arrays *A*, *B*, and *C* trivial since the compiler, runtime, and author of the *Block* distribution have taken care of all the details of managing local bounds and data segments, converting global indices to local addresses, and synchronizing between operations. This allows programming on distributed arrays to appear more similar to sequential computation. It also allows multiple levels of parallelism to be used to implement this statement—multiple processors, multiple cores, multithreading, and vector instructions—in a platform-independent manner without any intervention from the user.

4 Random Access in Chapel

The Random Access benchmark computes pseudo-random updates to a large distributed table of 64-bit integer values, T. Each table location T_i is initialized to store its index i in order to support verification of the computation afterwards. The pseudo-random values that are generated are used both to specify the table location to be updated and the value to use for the update. The updates themselves are performed by xor-ing the random value into the table location. Because multiple updates may attempt to modify the same table location simultaneously, the benchmark permits a certain fraction of such conflicting updates to be lost. Random Access is designed to test a system's ability to randomly access memory at a fine granularity. The benchmark permits updates to be performed in batches of a certain size as a concession to architectures and programming models that do not support fine-grain communication.

This section describes our implementation of the Random Access benchmark which builds on the concepts introduced in the previous section by introducing default arguments, call-by-argument name, argument query variables, type casts, iterators, and atomic sections. Our implementation is broken into two modules to separate the core computation from the generation of random values. The code for the computation is given in Appendix B.1 and described in Section 4.1 while the random stream generation module is listed in Appendix B.2 and described in Section 4.2.

4.1 Detailed Random Access Walk-through

In this section we walk through the Random Access computation, using Chapel concepts introduced in the STREAM Triad benchmark and explaining new ones as we go.

Module Uses Our Random Access code begins in lines 1–3 by listing the modules that it needs to `use`:

```
use Time;

use HPCCProblemSize, RARandomStream;
```

As in the previous section, the *Time* module is used to perform experimental timings and the *HPCCProblemSize* module is the same common module used to compute an appropriate problem size.

The third module, *RARandomStream*, is specific to this benchmark and implements an iterator that generates the random stream of values used to update the table. This code was factored into its own module to demonstrate how Chapel's iterators and modules allow different random number generators to be swapped in with minimal changes to the benchmark code itself. We show the use of the iterator in the computation and verification loops below, and describe its definition in the next section.

The following section (4.2) walks through the *RARandomStream* module in detail, and its source code listing is contained in Appendix B.2. We show the use of the iterator in the computation and verification loops described below.

Parameters and Types Lines 6–8 define the global parameters and types used to define the benchmark:

```
param numTables = 1;
type elemType = randType,
     indexType = randType;
```

The *numTables* parameter plays a similar role to *numVectors* in the STREAM Triad code, representing the number of large data structures that we need to allocate. The type alias *elemType* is used to define the type of element stored in the table while *indexType* is used to define the type used to index into the table (thereby constraining the maximum table size). Both of these types are defined to be *randType*, a type alias declared in the *RARandomStream* module to represent the values returned by the stream (declared to be Chapel's 64-bit unsigned integer value, `uint(64)`). While different types can be specified for *elemType* and *indexType*, constraining the three types to be the same in this way is natural since the random numbers serve both as the values to accumulate into the table and as the basis for computing the indices to update.

Configuration Variables, Default Arguments, Call-by-Argument-Name, and Casts Lines 10–22 define the configuration variables used to drive the program execution, as well as a few global constants whose values are computed from the configuration variables:

```
config const n = computeProblemSize(elemType, numTables,
                                     returnLog2=true): indexType,
              N_U = 2**(n+2);

const m = 2**n,
      indexMask = m-1;

config const sequentialVerify = (numLocales < log2(m)),
              errorTolerance = 1.0e-2;

config const printParams = true,
              printArrays = false,
              printStats = true;
```

The first line declares a configuration variable, *n* that is used to represent the $\log_2$ of the problem size. This allows the user to specify a problem size on the command line while constraining it to be a power of 2.

We use the same `computeProblemSize()` routine that we used for the STREAM Triad benchmark, but pass it an additional argument that wasn't used in that code. Two things are worth noting about this argument. The first is the fact that we did not use it previously. As in many languages, formal arguments in Chapel may be given *default values* which allow them to be omitted from calls that do not wish to set them. In this case, the *returnLog2* argument has a default value of `false`, indicating that the problem size should typically be unconstrained and returned directly, as in the STREAM Triad code. As we will see in Section 6, setting this argument to `true` constrains the problem size to be a power of 2 and returns the $\log_2$ of its value. The second note for this argument is that it uses a *call-by-argument-name* style of argument passing, in which the formal argument with which the actual is to be matched is named in the call (in this case, *returnLog2*). This usage is not actually necessary since *returnLog2* is the third formal argument in `computeProblemSize()`. However, we believe that it makes the call more readable than if the boolean literal `true` was passed in directly.

The call to `computeProblemSize()` is followed by the expression "`: indexType`." This is Chapel's syntax for a *dynamic cast* which safely coerces an expression into the specified type. Note that as in our declaration syntax, the "`:`" character is used to indicate an explicit type specification (and this is its only purpose in Chapel). In this declaration, we cast *n* to be of type *indexType*—a 64-bit unsigned integer—since it will be used to compute the table size.

The next declaration creates *N_U* which represents the number of updates required by the benchmark (N_U). This is initialized using Chapel's exponentiation operator (`**`). We write this expression as an exponent rather than a bit-shift operation for clarity and due to the belief that even a modest compiler's strength reduction optimizations should convert exponentiation on powers of two into the appropriate bit-shift operators.

The next two lines declare global constants *m* and *indexMask*, used to represent the problem size and to convert random numbers into table indices, respectively. The subsequent two lines are used to control the benchmark's verification step. The first specifies whether the verification should be computed sequentially or in parallel while the second specifies the error tolerance allowed in the verification due to conflicting updates. We use a simple heuristic to determine whether or not to use a sequential verification step based on the number of locales being used to execute the program. The final three configuration variables are used to control the program's output, as in our STREAM Triad implementation.

RA's Domains and Arrays The program's `main()` routine is defined on lines 25–45 and begins by printing the problem configuration and declaring its distributed domains and arrays:

```
def main() {
  printConfiguration();

  const TableSpace: domain(1, indexType) distributed(Block) = [0..m);
  var T: [TableSpace] elemType;

  const UpdateSpace: domain(1, indexType) distributed(Block) = [0..N_U);
  ...
}
```

The domain *TableSpace* is used to define the table. It is similar to the *ProblemSpace* domain in the STREAM Triad benchmark in that it is a 1D arithmetic domain that is distributed in a *Block* manner. We chose the *Block* distribution since it supports fast linear initialization, fast random access computations, and is statically load-balanced. However,

any distribution with these properties would be appropriate for this benchmark given the unpredictable nature of its access pattern.

Two differences exist between these domains and those used in STREAM Triad. The first is that the domain specification is parameterized by *indexType* in addition to the rank parameter "1". This specifies that the domain should represent its indices using *indexType*, a `uint(64)`. By default, arithmetic domains store indices of Chapel's default 32-bit `int` type. The second difference is that the domain's index set is specified using the expression `[0..m)`. This is syntactic sugar in Chapel to represent the index set [0..m-1] due to the prevalence of this pattern in codes that use 0-based indexing.[6] In this instance, we use a 0-based index set because it simplifies the arithmetic required when indexing into the table.[7]

The next line declares the table array, *T*, defined using the *TableSpace* domain and *elemType*. The following line declares a second domain, *UpdateSpace*, which is used to represent the updates against the table. This domain differs from previous ones we've created in that it is distributed, yet not used to allocate arrays. Instead, *UpdateSpace* is created simply to describe a set of work and to distribute that work between the locales. As with *TableSpace*, *UpdateSpace* is a 1D arithmetic domain indexed using *indexType*, implemented using the *Block* distribution, and initialized using the half-open interval style of domain specification. In distributing *UpdateSpace*, we chose the *Block* distribution under the assumption that the target machine is homogeneous enough and the workload random enough that a statically blocked work distribution would be sufficient. If these assumptions are incorrect, a standard distribution that dynamically distributes blocks of work could be used instead.

Timings and Table Initialization Lines 33–41 contain the timed portion of the benchmark:

```
const startTime = getCurrentTime();

[i in TableSpace] T(i) = i;
...
const execTime = getCurrentTime() - startTime;
```

The timing itself is performed using the `getCurrentTime()` routine, as in STREAM Triad, and is stored in the scalar variable *execTime*.

The initialization of the table, *T*, is expressed using a forall expression that assigns index *i* to element T_i of the table. This statement serves as an example of a Chapel statement in which dynamic bounds checks can be proven away by the compiler trivially. In particular, each Chapel domain *D* defines an *index type*, `index(D)`, that represents the domain's index type (*e.g.*, an integer index) and is guaranteed to be contained within domain *D*. In a domain iteration like this one, the index variable *i* is inferred to be of type `index(TableSpace)`. When *i* is used to index into array *T*, it is semantically guaranteed to be in-bounds because *T* was defined using domain *TableSpace*. This particular example is admittedly simple, and even without domains a good compiler would be able to prove the bounds-check away. However, once a program begins to declare and store index variables, or pass them as arguments, the index types serve to preserve those semantics over larger dynamic code ranges, helping both the compiler and the reader understand the program's semantics.[8]

One final "cute" note about the initialization of *T* is that it could have been written more succinctly as:

```
T = TableSpace;
```

This statement uses promotion to assign the indices represented by *TableSpace* to the elements stored by *T* in an elementwise manner. We chose not to use this idiom because its meaning seems far less clear to the casual reader.

The Update Loop and Iterator Usage The update loop for Random Access occurs in lines 37–39:

```
coforall block in UpdateSpace.subBlocks do
  for r in RAStream(block) do
    T(r & indexMask) ^= r;
```

[6] This syntax was chosen to suggest the mathematical notation for half-open intervals $[a \ldots b)$. Note that making the low index open (*e.g.*, `(lo..hi]` or `(lo..hi)`) is not supported in Chapel due to ambiguities with simple expression parenthesization as well as the observation that these cases are not as common in domain programming.

[7] Chapel strongly believes that a scientific programming language should not impose 0- or 1-based indexing on programmers given that indices often have important semantic meaning to a problem's description and no single choice of lower bound best satisfies all needs. We avoid supporting a default lower bound for our index sets to maintain neutrality in the debate between 0-based and 1-based programming.

[8] Conversely, when all of your indices are represented using integers, it can be hard to reason about their semantic role in a program—for example, the varied roles of integers in a compressed sparse row data structure to represent sparse arrays. Chapel also has a *subdomain* concept, not used in these benchmarks, that can be used to establish relationships between related domains and their indices for more interesting access patterns.

This loop is an example of expressing parallelism in a simple, architecturally-neutral way. Our goal is to create parallel work such that each locale has an appropriate number of tasks, each performing a fraction of the locale's random updates. To this end, we create a parallel outer `coforall` loop to describe the parallelism across the machine resources and a serial inner loop to express the updates owned by that resource.

The outer loop invokes an iterator method, `subBlocks`, defined on domains. This iterator generates sub-blocks of that domain to match the machine's parallel resources. As an example, if a machine's locale was a multicore processor, `subBlocks` would divide the locale's portion of *UpdateSpace* into a sub-block per core. If the locale was a multithreaded processor, it would create a sub-block per hardware thread context. For a single-threaded, single-processor locale, the sub-block would be that locale's entire portion of *UpdateSpace*. The `subBlocks` iterator generates values that are subdomains of the domain on which it was invoked, so for this loop *block* represents a 1D arithmetic subdomain of *UpdateSpace*. Because we want to create an explicit parallel task per sub-block, we use a `coforall` loop to describe the parallelism in this loop. A `forall` loop would also assert that the loop could be executed in parallel, but would not guarantee that we would get a parallel task per iteration.

The inner loop uses a second iterator, `RAStream()` that we define in the *RARandomStream* module, described below. This iterator takes a range or 1D domain as its argument to indicate the indices of the pseudo-random values to generate. In this invocation, we provide these arguments using the *block* of indices generated by the `subBlocks` iterator in the outer loop.

The update itself is expressed in the body of the inner loop, using Chapel's bitwise-and (`&`) and -xor-assignment (`^=`) operators to compute the index and modify the table. Since nothing in our code prevents multiple threads from attempting to update the same element of *T* simultaneously, this implementation contains a race condition as permitted by the benchmark specification. During the verification computation, we will show how this race condition can be eliminated using *atomic sections*.[9]

This loop is an excellent demonstration of how iterators can be used to abstract complex looping structures away from loop bodies just as subroutines are used to abstract computations away from other code. In particular, note that `RAStream()` could be replaced by any iterator that generates indices for this loop, whether using a different randomization algorithm, computing the indices directly, or reading them from a database. Moreover, since the type of *r* is inferred, `RAStream()` could be modified to return an aggregate of random numbers using a range or array which would result in an implicitly parallel update to multiple values of *T* due to the resulting promotion of its indexing and xor-assignment operators. A loop body that uses promoted operators in this way enables the compiler to implement the parallelism using vector operations on a vector processor, multiple threads, or by bucketing remote values and scattering them in chunks as many MPI-based implementations of Random Access do. Note that while such architecturally-specific tuning decisions do require modifications to the code, all the modifications are isolated to the iterator's definition, leaving the expression of the computation intact.

Argument Query Variables We skip over the calls used to print the results in `main()` and to define the `printConfiguration()` routine, due to their similarity to the code in the STREAM Triad implementation. The next routine is `verifyResults()`, defined on lines 56–74. Its definition takes the following form:

```
def verifyResults(T: [?TDom], UpdateSpace) {
  ...
}
```

This is the first subroutine definition we have encountered in which an argument's type is expressed via the "`:`" operator. However, rather than specify the type precisely, we give only part of the argument's definition, indicating that it is an array (due to the square brackets) but with no element type specified. This requires that the argument *T* be passed an array argument, but permits any element type.

In addition, note that rather than specifying a legal domain identifier, the syntax "`?TDom`" is used. This expression, which we refer to as an *argument query variable*, is used to create a local identifier named *TDom* that will be bound to *T*'s domain. We will see how *TDom* is used later. Alternatively, the domain could have been explicitly queried using a method and bound to a local variable, but the query syntax provides a convenient and intuitive shorthand. Argument query variables also allow the new identifier to be reused in the formal argument list to express a constraint between arguments, or to define the routine's return type.

[9] At various times, it has been proposed that Chapel's *op*= assignment operators automatically be considered atomic since the expression being read/written only appears once in the statement. Current thinking is that these semantics are too subtle and may cause unwanted overhead when such atomicity is not required, so such operators can result in races in our current plan of record.

Verification Loops and Atomic Statements The verification computation is expressed in lines 59–66:

```
if (sequentialVerify) then
  for r in RAStream([0..N_U)) do
    T(r & indexMask) ^= r;
else
  forall i in UpdateSpace {
    const r = getNthRandom(i+1);
    atomic T(r & indexMask) ^= r;
  }
```

We provide two verification loops, selected by a configuration variable: The first performs the verification sequentially; the second performs it in parallel using a modified algorithm from the one we used to compute the updates, and also eliminates race conditions.

The sequential verification loop simply uses a serial for-loop statement, using the same `RAStream()` iterator as in the computation, but passing it an anonymous domain describing the entire update space, `[0..N_U)`. The loop body is as before, and is guaranteed to be safe due to the sequential execution of the loop.

The parallel verification loop specifies a `forall` iteration over all the indices in the *UpdateSpace* domain. Rather than using the `RAStream()` iterator, it uses a different subroutine from the *RARandomStream* module, `getNthRandom()` which computes the n^{th} random number directly, though in a more expensive fashion.[10] The table update itself is prefixed by the `atomic` keyword. This modifier causes a statement to appear to execute atomically from the point of view of all other threads. In practice it may be applied to a compound statement containing many read and write operations, but here we use it simply to ensure that no thread will read a stale value of *T* that another thread is about to overwrite. This eliminates the race condition in our original computation and allows us to perform the table updates safely, as requied by the verification phase.

Wrapping up Random Access The rest of the verification routine uses concepts that we have seen before: a + reduction is used to count the number of table elements that don't contain their index; I/O statements are used to optionally print out the table's values and number of errors; and a comparison against the error tolerance is performed and returned to determine whether or not the test was successful.

The final routine in this module is the `printResults()` routine which prints out the experimental results using familiar concepts.

4.2 Walk-through of the RARandomStream Module

As mentioned earlier, we implemented the random stream iterator in its own module in order to emphasize its modularity and independence from the updates being computed by the benchmark. This enables alternative implementations to be plugged in simply by specifying another file that supports the same module name and public interface. Or, as a minor modification, the module and iterator names in the original source text may be changed. In this section, we walk through the salient details of this helper module, listed in full in Appendix B.2.

An Explicit Module Declaration In all of our previous examples, we have defined modules that are not used by any other modules, and therefore have relied on the convenience of having the compiler generate a module name from the filename. For the *RARandomStream* module, we want to `use` it from other modules and would also like the option of creating several implementations in different files, so we scope the code and name the module as follows in lines 1–52:

```
module RARandomStream {
  ...
}
```

[10] Note that the argument $i + 1$ is used to make our random number generator match the reference implementation only for the benefit of comparison—the reference random number generator seems to compute two consecutive numbers when a new sequence is requested and we mimicked this in our iterator class, requiring us to offset i by 1 here. With slight modifications to the random number generator, this issue could be cleaned up, allowing us to pass i in directly, but we chose to follow the reference implementation in order to support direct comparison.

RARandomStream Globals Lines 2–6 define the global types and variables for this module:

```
param randWidth = 64;
type randType = uint(randWidth);

const bitDom = [0..randWidth),
      m2: [bitDom] randType = computeM2Vals(randWidth);
```

The first declaration declares a `param` named *randWidth* to represent the bit-width for the random numbers generated by this module. We name this value because it is used to initialize several of the other globals. The next statement defines the type alias *randType* to represent the type of the random numbers we will be generating—namely, Chapel's 64-bit unsigned integer type, `uint(64)`.

The next two declarations create a domain *bitDom* and an array *m2* of size *randWidth*, which are used as a lookup table to compute polynomials quickly. The declaration of *bitDom* differs from other domains that we've seen due to the fact that its type is inferred from its initializer. All of our previous domain declarations have been distributed, a property which cannot be inferred from a domain value. Here, our intention is to give every locale fast access to the lookup table, so we declare it to be `const` so that the compiler will replicate it. The *m2* array is initialized using another iterator, `computeM2Vals()`, defined at the end of the module.

Compiler Status Note: Here is the first instance in which our compiler does not handle the code we would ultimately like to write. It would be nice to declare this lookup table as a `param` to enable it to be computed at compile-time and potentially used for optimizations. However, our compiler currently does not support domain and array `param` values and declaring them as such causes the compilation to fail.

Chapel modules can specify which of their global symbols are private or public. If no specification is made, all global symbols are assumed to be public in order to support exploratory programming. As soon as any global symbol is specified as private or public, all other globals within the module are considered private until otherwise specified. The module `use` statement can also filter a module's global symbols controlling which are or are not imported, and renaming them to avoid conflicts if desired.

Iterators Lines 9–15 contain our first iterator declaration:

```
def RAStream(block) {
  var val = getNthRandom(block.low);
  for i in block {
    getNextRandom(val);
    yield val;
  }
}
```

Iterators are declared similarly to normal subroutines except that instead of returning a single value, they contain `yield` statements that generate values for the callsite. After a yield statement, the iterator logically continues running, continuing onward from that point. Conceptually, an iterator returns a sequence of values, though our implementation strategies never require the sequence to be manifested in memory.

As described before, this iterator takes one argument, *block*, which defines the sequence of indices that specify the random values that it needs to generate. The iterator is written in a very general way such that block can be either a range or a 1D domain (or, for that matter, any type that supports a low method and a default iterator method).

The body of the iterator appears much like a normal subroutine. It starts by getting an initial value using the `getNthRandom()` routine described earlier and defined below. It then enters a standard `for` loop, computing the next random value and yielding it back to the callsite.[11] When all the values have been yielded, the iterator returns, causing the loop which invoked it to terminate.

The getNthRandom() Subroutine Lines 18–33 define the `getNthRandom()` subroutine which can be used to randomly access an element in the stream. This code is essentially a port of the reference implementation, cleaning it up in a few places. It largely uses concepts we have seen previously, but we call out a few key lines here:

```
def getNthRandom(in n) {
  ...
  n %= period;
```

[11] The fact that the next random number is computed before yielding the value is the reason that we had to pass $i + 1$ into `getNthRandom()` in our verification loop as noted previously. By swapping the two statements in this loop, the off-by-one issue would disappear, though our implementation would then compute different updates than the reference version.

```
  ...
  var ran: randType = 0x2;
  for i in [0..log2(n)) by -1 {
    var val: randType = 0;
    ...
    if ((n >> i) & 1) then getNextRandom(ran);
  }
  ...
}
```

This routine is the first we have seen to use an argument intent. The argument *n* is declared with the intent "`in`" to indicate that a copy of its value should be made when passing it to this routine, allowing it to be modified locally. In particular, we replace *n* with its value modulo the generator's period using Chapel's modulus assignment operator (`%=`).[12]

This routine also uses more explicit typing of variables than we have seen previously to ensure that *ran* and *val* are represented using `uint(64)` values rather than being inferred to be 32-bit `int`s.

Additionally, the routine uses Chapel's standard math routine for computing $\log_2$, overloaded for integer values to use a fast implementation via bit manipulations. The loop from 0 to $\log_2 n - 1$ uses Chapel's `by` operator to iterate over the range backwards, from the high value down to the low value.

Finally, this routine uses Chapel's bit-shift operators (>> and <<).

The getNextRandom() Subroutine The `getNextRandom()` subroutine is defined on lines 36–41:

```
def getNextRandom(inout x) {
  param POLY:randType = 0x7;
  param hiRandBit = 0x1:randType << (randWidth-1);

  x = (x << 1) ^ (if (x & hiRandBit) then POLY else 0);
}
```

This routine is notable only in that it uses an `inout` argument intent to indicate that the argument passed to it should be copied in, and that any modifications to it should be copied back when the routine completes. It also contains our second compromise in the implementation:

Compiler Status Note: Due to a subtle semantic rule involving parameter values and implicit coercions, the variable *POLY* should not need to be explicitly typed but rather should automatically be coerced to a `uint(64)` due to context. However, due to limitations in the current implementation, its type must be specified explicitly.

The computeM2Vals() Iterator Lines 44–51 wrap up the *RARandomStream* module by defining an iterator named `computeM2Vals()` that is used to initialize the lookup table *m2*. Now that you are a Chapel expert, you should find it completely intuitive.

4.3 Random Access Highlights

This section summarizes the Random Access walkthrough by pointing out a few key features:

- The main productivity feature demonstrated by this code is the use of iterators and modules to completely abstract the random stream generator away from the loops that use those values. This permits alternative implementations of the random number stream to be swapped in and out simply by using the same module name and providing the same interface (or, alternatively, by simply changing the module name and iterator invocations in the application code rather than restructuring three distinct loop nests).
- A second feature is the use of the `subBlocks` iterator to create an appropriate amount of parallel work for the target architecture in a portable way. Using this feature of domains allows programmers to specify where they want parallel work to be created without cluttering their computational loops with all of the details of creating and managing it. Instead, those details are abstracted into the distribution classes used to define domains.
- Another feature is the ability to change the granularity of work performed in the body of the update loop simply by modifying the random number iterator to return ranges or vectors of indices rather than a single index at a time. If done properly, this can support vectorization of the loop body and/or bucketing of updates to amortize communication overheads.

[12] This bounding of *n* was modeled after the reference implementation though there is debate within our group about its necessity and utility.

- The use of atomic sections is a very clean means of specifying that updates performed in parallel must not conflict with each other in the verification loop. This concept is receiving a lot of attention in the mainstream programming community as multicore processors become more commonplace and people search for ways to utilize them productively.
- Finally, Chapel's use of global-view arrays and distributions once again simplifies the programmer's job by taking care of the distributed allocation and maintenance of the table, the element location and communication required to perform the updates, and the synchronization.

5 FFT in Chapel

The FFT benchmark requires the user to compute a 1D discrete fourier transform on a vector of pseudo-random values. The programmer is given a fair amount of latitude in choosing an FFT algorithm and approach as long as the outcome is correct. Depending on how it is written, the FFT benchmark could stress a number of system parameters including floating point operations, local memory bandwidth, cache size, and network.

In our implementation, we use a radix-4 DFT algorithm in order to take advantage of its beneficial Flops-to-MemOps ratio. A radix-4 FFT on a vector of length n can be thought of as containing $log_4 n$ phases where each stage computes $n/4$ *butterfly computations*. These butterflies replace a 4-element slice of the input vector with new values computed as a function of the elements and a vector of values known as *twiddle factors*. In phase i of the algorithm, the stride between the elements in each slice is 4^i, causing each slice to span 4^{i+1} elements. In each phase, several butterflies will use identical twiddle factors. We refer to these groups of butterflies as *banks*. If the problem size is not a power of 4, a final radix-2 cleanup stage is required that computes butterflies on pairs of elements.

This section describes our implementation of the FFT benchmark, which introduces the following new concepts: complex types and imaginary literals, let expressions, tuples, strided ranges and domains, zippered iteration, unbounded arithmetic ranges, and array views.

5.1 Detailed FFT Walk-through

Appendix C contains the full code specifying our FFT implementation. In this section we walk through that code, pointing out interesting Chapel concepts and computations as we go. Due to the relative size and straightforwardness of the FFT program, we cover the code in a bit less detail than in previous sections.

Module Uses and Globals FFT's module uses and global declarations are specified in lines 1–23 and are similar to what we've seen in the previous codes. We excerpt a few lines of interest here:

```
use BitOps, Random, Time;
...
param radix = 4;
...
type elemType = complex(128);
```

The first line shows that our FFT uses Chapel's standard *BitOps* module, which defines a rich set of bit manipulation library routines, including logical bitwise operations that do not have operator support, bit matrix operations, and parity checks.

The next line defines a `param` value, *radix* which indicates that this is a radix-4 implementation. The final line defines our element type to be Chapel's 128-bit complex type, `complex(128)`. Complex numbers are a built-in type in Chapel, permitting the compiler to reason about and optimize complex arithmetic rather than pushing it into a library as in many other languages. Chapel also supports a *pure imaginary* type, `imag`, to complement its `real` type. This type is useful to the compiler in helping it to optimize complex arithmetic and maintain numerical stability.

The rest of the global declarations `use` modules that we have seen before and declare familiar variables: problem sizes, verification parameters, randomization controls as in STREAM Triad, and I/O controls as in both of the previous benchmarks.

FFT's Domains and Arrays The structure of FFT's `main()` routine, defined on lines 26–47 is much as we have seen before: It prints the problem configuration, declares its domains and arrays, initializes the arrays, performs the timed computation, verifies the answer, and prints the results. The domain and array declarations are as follows:

```
const TwiddleDom: domain(1) distributed(Block) = [0..m/4);
var Twiddles: [TwiddleDom] elemType;

const ProblemDom: domain(1) distributed(Block) = [0..m);
var Z, z: [ProblemDom] elemType;
```

This benchmark defines two main domains—one for the twiddle factors, and a second for the problem itself. Both are defined using a *Block* distribution and 0-based indexing to simplify index computations. Later in this section, we will discuss the idea of redistributing the domain to reflect the highly strided accesses that take place in the FFT's later phases.

The FFT Computation Lines 39–41 constitute the timed FFT computation:

```
Z = conjg(z);
bitReverseShuffle(Z);
dfft(Z, Twiddles);
```

Unlike the previous benchmarks, we move the timed computation into subroutines due to its size and the fact that these routines are used in multiple places in the benchmark. The first line uses the standard math routine `conjg()` in a promoted manner to compute the complex conjugate of the input vector *z* and store the result into our working array *Z*. The second line calls a routine defined later in the file, `bitReverseShuffle()` to perform vector permutations. The third line computes the core FFT computation, defined below in `dfft()`.

Problem Initialization The Twiddle and data vectors are initialized in the routine `initVectors()`, defined on lines 55–65. This routine calls `computeTwiddles()` to compute the twiddle factors and then permutes them using the `bitReverseShuffle()` routine once again. The `computeTwiddles()` routine is defined on lines 68–81 as follows:

```
def computeTwiddles(Twiddles) {
  const numTwdls = Twiddles.numElements,
        delta = 2.0 * atan(1.0) / numTwdls;

  Twiddles(0) = 1.0;
  Twiddles(numTwdls/2) = let x = cos(delta * numTwdls/2)
                         in (x, x):elemType;
  forall i in [1..numTwdls/2) {
    const x = cos(delta*i),
          y = sin(delta*i);
    Twiddles(i)            = (x, y):elemType;
    Twiddles(numTwdls - i) = (y, x):elemType;
  }
}
```

This routine uses a couple of concepts that we haven't seen before. The first is the 0-argument `numElements` method, defined on arrays to return the number of elements they are storing. It is used here, and in other routines in the FFT, to make this routine general across vector arguments of various sizes.

The second is a *let expression* which can be used to create a variable that is local to an expression. In this code, the let expression is used to call the standard cosine function a single time and use the value twice in one expression. It is simply a convenience to avoid declaring a variable at the function scope or typing the call to cosine twice.

The final new concept is the use of a type cast to coerce a *tuple* of floating-point values into a complex value (*elemType*). Though we do not use tuples much in these benchmarks, Chapel has general support for tuple types, allowing them to be created, destructured, passed as arguments, and returned from subroutines. Tuples are also used to represent the index variables of arithmetic domains for ranks greater than 1. While complex numbers are not implemented in Chapel as tuples, we support casts from 2-tuples of numeric values into complex types as a convenient way to define complex values component-wise.

The bitReverseShuffle() Routines The `bitReverseShuffle()` routine is defined on lines 84–88 and performs a vector permutation by reversing the bits of each element's index. We express this routine as follows:

```
def bitReverseShuffle(Vect: [?Dom]) {
  const numBits = log2(Vect.numElements),
        Perm: [i in Dom] Vect.eltType = Vect(bitReverse(i, revBits=numBits));
  Vect = Perm;
}
```

The routine creates an array *Perm* that stores a permuted copy of the input array argument, *Vect*. This declaration uses an array declaration style that combines Chapel's standard array definition syntax with the forall expression notation in order to support initialization of each element as a function of its index (named *i* in this case). We use a helper routine, `bitReverse()` to compute the indices of the permutation and then use the resulting values to access the *Vect* array.

The next line copies the permuted array back into the input argument array. Since arrays are passed by reference by default, this change will be reflected back in the callsite's actual argument. The *Perm* array will be deallocated upon returning from the subroutine.

The helper subroutine `bitReverse()` is defined on lines 91–96:

```
def bitReverse(val: ?valType, revBits = 64) {
  param mask = 0x0102040810204080;
  const valReverse64 = bitMatMultOr(mask, bitMatMultOr(val:uint(64), mask)),
        valReverse = bitRotLeft(valReverse64, revBits);
  return valReverse: valType;
}
```

It uses bit matrix multiplication and rotation operators from the *BitOps* module to reverse the low *revBits* bits of an integer value. The bit matrix multiplication routines treat a 64-bit integer value as an 8×8 bit array. In this case, the value is multiplied against a bit matrix storing 1's in its antidiagonal to reverse the bits in the variable *val*. It then uses a bit rotation to bring the bits back into the low *revBits* locations. Casts are used on function entry and exit to convert the input value into an unsigned 64-bit integer and back to its original type on exit. Chapel's bit manipulation operators are intended to be implemented using special hardware instructions on platforms that support them, or optimized software routines otherwise.

FFT Loop Structure The FFT itself is computed using the `dfft()` routine defined on lines 99–133 and the helper routine `butterfly()` defined on lines 146–159. The main loop structure of `dfft()` is shown here:

```
for (str, span) in genDFTPhases(numElements, radix) {
  forall (bankStart, twidIndex) in (ADom by 2*span, 0..) {
    ...
    forall lo in bankStart + [0..str) do
      butterfly(wk1, wk2, wk3, A[[0..radix)*str + lo]);
    ...
    forall lo in bankStart + span + [0..str) do
      butterfly(wk1, wk2, wk3, A[[0..radix)*str + lo]);
  }
  ...
}
```

This structure starts with a sequential outer loop to enumerate the FFT's phases. This loop uses an iterator we wrote for this benchmark, `genDFTPhases()`, that generates a tuple for each phase defining the stride and span of the accesses required to compute the component butterflies (*str* and *span* respectively). This iterator is defined on lines 136–143 and is fairly straightforward, generating powers of four using a straightforward loop.

The next loop is used to describe the parallel banks of butterfly operations that use a common set of twiddle values. This forall loop uses a concept called *zippered iteration* to perform two iterations simultaneously in an elementwise fashion. This is expressed by specifying a tuple of iterator expressions and a tuple of index variables. In general, iterator expressions that are zippered together must have the same size and shape. The first expression in our zippered iteration is an expression that uses Chapel's `by` operator to iterate over the domain *ADom* in a strided manner. In particular, *bankStart* will take on the values *ADom.low*, *ADom.low*$+2 \cdot span$, *ADom.low* $+4 \cdot span$, ...

The second iterator expression in this loop is an *unbounded range* due to the fact that it specifies a low bound but no high bound. This causes the range to be the same length as the other iteration expression with which it is zippered. Here the unbounded range will contain the same number of elements as the strided iteration over the data

vector's domain, *ADom*. It is used to generate the iteration number as a 0-based index, stored using the variable *twidIndex*. Obviously, this range's high bound could be computed explicitly by the programmer, but the unbounded range notation reduces the chance of errors, eliminates the unnecessary math, and results in cleaner code.

The innermost loops express the butterfly computations that can be performed in parallel within each bank. These are expressed using calls to the helper `butterfly()` routine, passing twiddle values and an array slice of the vector *A*. This array slice is expressed by applying arithmetic operators to a range. Multiplying a range by a scalar causes its low and high bounds, as well as its stride, to be scaled by the scalar value. Adding a scalar to a range shifts its low and high bounds by the scalar. Thus the expression, `[0..radix)*str + lo` is equivalent to the range `lo..lo+(radix-1)*str by str`. In these array slices, we use range math in order to make the logic of the slicing operator clearer—in particular, to indicate that it contains *radix* elements strided by *str* and offset by *lo*. This form also reduces the redundancy of the *lo* and *str* references required by the longer form.

The rest of the `dfft()` routine computes twiddle values, loop bounds, and strides. There is also a cleanup phase in lines 122–132 that takes care of the final butterfly computations that are not handled by the main loop. This cleanup is either the set of radix-2 butterflies mentioned above or a final set of radix-4 butterflies that are specialized due to their twiddle factors (mentioned below).

The Butterfly Routine The routine to compute a single radix-4 butterfly is defined in lines 146–159:

```
def butterfly(wk1, wk2, wk3, inout A:[1..radix]) {
  var x0 = A(1) + A(2),
      x1 = A(1) - A(2),
      x2 = A(3) + A(4),
      x3rot = (A(3) - A(4))*1.0i;

  A(1) = x0 + x2;
  x0 -= x2;
  A(3) = wk2 * x0;
  x0 = x1 + x3rot;
  A(2) = wk1 * x0;
  x0 = x1 - x3rot;
  A(4) = wk3 * x0;
}
```

This routine performs the required arithmetic between the twiddle values and elements of the array slice. Three things about it are worth noting. The first is that this code demonstrates Chapel's support for imaginary literals, expressed by appending an "`i`" to an integer or floating point literal (imaginary literals are used elsewhere in the FFT code, but were elided in our code excerpts for this walkthrough). Like math on traditional numeric literals, operations like this multiplication by $1.0i$ will be strength-reduced by the Chapel compiler into less expensive operations than those required by complex or imaginary multiplication.

Our second note is that because the twiddle values have no declared types, if the user takes care to special-case calls in which a twiddle value has a 0.0 real or imaginary component, the compiler will create a specialized version of the routine and eliminate useless floating point operations against zero values. As an example, the cleanup code contains the following call on line 125:

```
butterfly(1.0, 1.0, 1.0, A[[0..radix)*str + lo]);
```

This will cause a version of the `butterfly()` routine to be created that takes floating point twiddle values, reducing the cost of the arithmetic operations as compared to the version that takes complex twiddle values. More aggressive users may also hoist initial iterations out of the FFT loops in order to specialize additional butterflies whose twiddle values lead to similar benefits.

The final note about the `butterfly()` routine is that the formal argument *A* is defined using the domain `[1..4]` while the array actuals being passed in are strided slices of a much larger array. This is called an *array view* and allows subroutines to be written that compute on subarrays using indices that are convenient to the routine, as in Fortran 90. In this case we define the argument to have `inout` intent to indicate that an array temporary should be created in the locale memory of the thread executing the routine. If a blank intent was used instead, the array elements would be accessed in-place.

Wrapping up FFT The remainder of the FFT code uses concepts that are quite familiar by now to verify the results and print them out.

5.2 FFT Highlights

Some salient details of the FFT code are as follows:

- Chapel's support for complex and imaginary literals supports a very clean expression of the math behind the FFT computation. Moreover, its support for generic types allows routines like the butterfly to be expressed in a way that is trivially specializable for twiddle values with 0.0 real or imaginary components.
- Support for strided domains, range arithmetic, and zippered iteration support a rich means of expressing the loops and array slices required for the computation.
- As always, Chapel's support for a global view of computation greatly simplifies the expression of this benchmark because the programmer can express the FFT algorithm without cluttering it with explicit decomposition of data structures and loops, communication, and synchronization. As we describe in the performance notes section of the companion overview paper to this one, FFT's access patterns raise some performance challenges for the compiler, yet we believe they are surmountable.

6 Common Module for Computing HPCC Problem Size

As described earlier, since the determination of problem sizes for the HPCC benchmarks is similar across the codes, we created a separate module, *HPCCProblemSize*, to compute and print problem sizes for use by all of the implementations. The code for this module is given in Appendix D. This section walks through the code at a high level introducing those concepts that have not been seen previously: type parameters, locale types and variables, and the halt routine.

6.1 HPCCProblemSize Module Walk-through

The *HPCCProblemSize* module defines two routines, one that computes the problem size and another that prints it out along with memory usage statistics.

Type Parameters The `computeProblemSize()` subroutine is defined on lines 6–27. Its interface is defined as follows:

```
def computeProblemSize(type elemType, numArrays, returnLog2 = false) {
  ...
}
```

This subroutine is the first we have defined that takes a `type` as an argument. In Chapel we allow subroutines and classes to be parameterized by types and parameters as in other languages that support generic programming. However, rather than using a specialized syntax to separate these arguments from the traditional value-based arguments, we have opted to specify them in-line with normal arguments in an effort to keep the language syntax cleaner. Needless to say, `type` and `param` arguments must be declared as such and their actual values must be computable at compile-time so that Chapel can maintain good performance at execution-time. By convention, type and parameter arguments appear first in formal argument lists. We hypothesize that this intermixing will trouble programming language experts more than it does the HPC user community.

In this routine, we use a type argument to specify the element type being stored by the benchmark. The other arguments specify the number of arrays to be allocated, and whether or not the routine should compute a power-of-2 problem size.

Locale Types and Variables In order to allow a program to refer to the hardware resources on which it is running, Chapel defines a *locale type* ("`locale`") that is used to represent the machine locales on which the program is running. The locale type is a fairly opaque type that cannot be used for much more than specifying where data should be located or computations should be run. In these benchmarks, we have never needed to refer to the locales explicitly since our algorithms have distributed their data over all of the locales and have used that distribution to drive the computation. Locales can also be used to query information about the machine's capabilities, as we will demonstrate in this routine.

When launching a Chapel program, the user specifies the number of locales on which it should execute on the command-line. The Chapel code has access to a global variable, *numLocales* that stores this value, as well as a 1D arithmetic array, *Locale*, that stores elements of `locale` type to represent the set of locales on which the program

is running. Users can reshape this array to create their own virtual representation of the machine resources to suit their algorithm. For example, programmers may choose to reshape the 1D array into a 3D array of locales in order to support distributions of 3D numerical arrays in their program.

In this module, we `use` Chapel's standard *Memory* module which provides routines for reasoning about memory. One of these routines defines a `physicalMemory()` method on the `locale` type which returns the amount of physical memory locally available to that locale. It also takes two optional arguments, indicating the unit of memory that the result should be returned in (defaulting to *Bytes* and the type that should be used to represent the return value (defaulting to `int(64)`). We use this routine at lines 7 and 22 in our code:

```
const totalMem = + reduce Locale.physicalMemory(unit = Bytes),
...
const smallestMem = min reduce Locale.physicalMemory(unit = Bytes);
```

Each of these statements calls the `physicalMemory()` method in a promoted manner over the *Locale* array. This will result in an array of phsyical memory counts which is then collapsed using a `+` reduction in the first statement and a `min` reduction in the second. The sum reduction is used to compute the total system memory, and from that a bunch of fairly straightforward math is done to find the problem size that meets the HPCC requirements. Then a sanity check is performed to ensure that an even share of that problem size will not exceed the minimum memory of a single locale (in case the machine memory is heterogeneous, for example).

The halt routine If this test fails, the standard routine `halt()` is called in line 24:

```
halt("System is too heterogeneous: blocked data won't fit into memory");
```

The `halt()` routine is similar to `writeln()` in that it can be used to print out a variable number of arguments. Then it halts the program, gracefully shutting down all threads. As such, `halt()` is the typical way of signaling an exceptional runtime condition in Chapel.[13]

If the test passes, the routine returns the result that it computed.

The printProblemSize() Subroutine The `printProblemSize()` routine is defined on lines 30–43, and is fairly unremarkable—it does some simple computations to compute memory usage and then prints out the results. Its one noteworthy feature is that it uses an argument query variable, `?psType`, to capture the type of the *problemSize* argument that is passed in (for our benchmarks it could be either a signed or unsigned value) and uses casts on the values returned by the `log2()` routines in order to ensure that the operations performed on them are legal; otherwise the compiler will complain.[14]

7 Summary and Future Work

In summary, we have created implementations of the HPCC STREAM Triad, Random Access, and FFT benchmarks that we believe represent the computations cleanly, succinctly, and in a manner that will support good performance as our compiler development effort proceeds. As stated in the introduction, all codes listed in the appendices compile and run with our current compiler.

Our future implementation priorities are to support multi-locale execution for distributed memory machines and to implement parallel forall-style iteration for single-locale ranges and domains in order to take advantage of multicore processors. We plan to minimize hardware and software assumptions during this phase in order to support portability of the implementation (*e.g.*, commodity Linux clusters are a primary target). We plan to release updated versions of the compiler to larger audiences as time goes on.

In concluding this article, it is worth noting that while the HPCC benchmarks have demonstrated many of Chapel's productivity features for global-view programming and software engineering, they remain rather restricted in terms of the parallel concepts that they exercise. In particular, none of these benchmarks required any significant task

[13] One of our most frequently asked questions is whether Chapel supports exceptions. While we are great fans of exceptions, when we were framing the language we did not have the resources or expertise to construct an exception model for parallel computation in which we could have confidence. This is something we would like to incorporate into version 2.0 of Chapel.

[14] One disadvantage of using a modern, type-safe language is that operations combining signed and unsigned values that you have been using for years in C without a second thought are suddenly flagged as being ambiguous or illegal by the compiler. It is often enough to make one want to throw unsigned types out of the language, but we tried that and it was the first feature that the HPCS user community thought was missing. Now that we have added them, we are anticipating the day those users request a compiler flag that enables unsafe C-style coercions between signed and unsigned values. In the meantime, we are looking to C# as our model for wisdom on this matter.

parallelism, thread synchronization, or nested parallelism. Because the computations were typically driven by a single distributed problem vector, there was no need for Chapel's features for explicit locality control. And even within the data parallel space, all of these benchmarks used only dense 1D vectors of data, leaving Chapel's support for multidimensional, strided, and sparse arrays; associative and opaque arrays; and array composition on the table. In future work, we intend to undertake similar studies for computations that exercise a broader range of Chapel's features.

Acknowledgments The authors would like to thank the 2005 HPC Challenge finalists who shared their implementations with us and fielded questions about them—Petr Konecny, Nathan Wichmann, Calin Cascaval, and particularly John Feo who helped us learn a lot about FFT in a short amount of time. We would also like to recognize the following people for their efforts and impact on the Chapel language and its implementation—David Callahan, Hans Zima, John Plevyak, David Iten, Shannon Hoffswell, Roxana Diaconescu, Mark James, Mackale Joyner, and Robert Bocchino.

References

[1] Bradford L. Chamberlain. *The Design and Implementation of a Region-Based Parallel Language*. PhD thesis, University of Washington, November 2001.

[2] Cray Inc., Seattle, WA. *Chapel Specification*. (http://chapel.cs.washington.edu).

[3] Steven J. Deitz, David Callahan, Bradford L. Chamberlain, and Lawrence Snyder. Global-view abstractions for user-defined reductions and scans. In *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on principles and practice of parallel programming*, pages 40–47. ACM Press, 2006.

[4] Jack Dongarra and Piotr Luszczek. HPC challenge awards: Class 2 specification. Available at: http://www.hpcchallenge.org, June 2005.

A STREAM Triad Chapel Code

```
use Time, Types, Random;

use HPCCProblemSize;


param numVectors = 3;
type elemType = real(64);

config const m = computeProblemSize(elemType, numVectors),
             alpha = 3.0;

config const numTrials = 10,
             epsilon = 0.0;

config const useRandomSeed = true,
             seed = if useRandomSeed then SeedGenerator.clockMS else 314159265;

config const printParams = true,
             printArrays = false,
             printStats = true;


def main() {
  printConfiguration();

  const ProblemSpace: domain(1) distributed(Block) = [1..m];
  var A, B, C: [ProblemSpace] elemType;

  initVectors(B, C);

  var execTime: [1..numTrials] real;

  for trial in 1..numTrials {
    const startTime = getCurrentTime();
    A = B + alpha * C;
    execTime(trial) = getCurrentTime() - startTime;
  }

  const validAnswer = verifyResults(A, B, C);
  printResults(validAnswer, execTime);
}


def printConfiguration() {
  if (printParams) {
    printProblemSize(elemType, numVectors, m);
    writeln("Number of trials = ", numTrials, "\n");
  }
}


def initVectors(B, C) {
  var randlist = RandomStream(seed);

  randlist.fillRandom(B);
  randlist.fillRandom(C);

  if (printArrays) {
    writeln("B is: ", B, "\n");
    writeln("C is: ", C, "\n");
  }
}
```

```
65 def verifyResults(A, B, C) {
66   if (printArrays) then writeln("A is: ", A, "\n");

68   const infNorm = max reduce [i in A.domain] abs(A(i) - (B(i) + alpha * C(i)));

70   return (infNorm <= epsilon);
71 }


74 def printResults(successful, execTimes) {
75   writeln("Validation: ", if successful then "SUCCESS" else "FAILURE");
76   if (printStats) {
77     const totalTime = + reduce execTimes,
78           avgTime = totalTime / numTrials,
79           minTime = min reduce execTimes;
80     writeln("Execution time:");
81     writeln("  tot = ", totalTime);
82     writeln("  avg = ", avgTime);
83     writeln("  min = ", minTime);

85     const GBPerSec = numVectors * numBytes(elemType) * (m/minTime) * 1.0e-9;
86     writeln("Performance (GB/s) = ", GBPerSec);
87   }
88 }
```

B Random Access Chapel Code

B.1 Random Access Computation

```
use Time;

use HPCCProblemSize, RARandomStream;


param numTables = 1;
type elemType = randType,
     indexType = randType;

config const n = computeProblemSize(elemType, numTables,
                                     returnLog2=true): indexType,
             N_U = 2**(n+2);

const m = 2**n,
      indexMask = m-1;

config const sequentialVerify = (numLocales < log2(m)),
             errorTolerance = 1.0e-2;

config const printParams = true,
             printArrays = false,
             printStats = true;


def main() {
  printConfiguration();

  const TableSpace: domain(1, indexType) distributed(Block) = [0..m);
  var T: [TableSpace] elemType;

  const UpdateSpace: domain(1, indexType) distributed(Block) = [0..N_U);

  const startTime = getCurrentTime();

  [i in TableSpace] T(i) = i;

  coforall block in UpdateSpace.subBlocks do
    for r in RAStream(block) do
      T(r & indexMask) ^= r;

  const execTime = getCurrentTime() - startTime;

  const validAnswer = verifyResults(T, UpdateSpace);
  printResults(validAnswer, execTime);
}


def printConfiguration() {
  if (printParams) {
    printProblemSize(elemType, numTables, m);
    writeln("Number of updates = ", N_U, "\n");
  }
}


def verifyResults(T: [?TDom], UpdateSpace) {
  if (printArrays) then writeln("After updates, T is: ", T, "\n");

  if (sequentialVerify) then
    for r in RAStream([0..N_U)) do
```

```
61        T(r & indexMask) ^= r;
62    else
63      forall i in UpdateSpace {
64        const r = getNthRandom(i+1);
65        atomic T(r & indexMask) ^= r;
66      }

68    if (printArrays) then writeln("After verification, T is: ", T, "\n");

70    const numErrors = + reduce [i in TDom] (T(i) != i);
71    if (printStats) then writeln("Number of errors is: ", numErrors, "\n");

73    return numErrors <= (errorTolerance * N_U);
74  }


77  def printResults(successful, execTime) {
78    writeln("Validation: ", if successful then "SUCCESS" else "FAILURE");
79    if (printStats) {
80      writeln("Execution time = ", execTime);
81      writeln("Performance (GUPS) = ", N_U / execTime * 1.0e-9);
82    }
83  }
```

B.2 Random Access Random Stream Generator

```
module RARandomStream {
  param randWidth = 64;
  type randType = uint(randWidth);

  const bitDom = [0..randWidth),
        m2: [bitDom] randType = computeM2Vals(randWidth);


  def RAStream(block) {
    var val = getNthRandom(block.low);
    for i in block {
      getNextRandom(val);
      yield val;
    }
  }


  def getNthRandom(in n) {
    param period = 0x7fffffffffffffff/7;

    n %= period;
    if (n == 0) then return 0x1;

    var ran: randType = 0x2;
    for i in [0..log2(n)) by -1 {
      var val: randType = 0;
      for j in bitDom do
        if ((ran >> j) & 1) then val ^= m2(j);
      ran = val;
      if ((n >> i) & 1) then getNextRandom(ran);
    }
    return ran;
  }


  def getNextRandom(inout x) {
    param POLY:randType = 0x7;
    param hiRandBit = 0x1:randType << (randWidth-1);

    x = (x << 1) ^ (if (x & hiRandBit) then POLY else 0);
  }


  def computeM2Vals(numVals) {
    var nextVal = 0x1: randType;
    for i in 1..numVals {
      yield nextVal;
      getNextRandom(nextVal);
      getNextRandom(nextVal);
    }
  }
}
```

C FFT Chapel Code

```
1  use BitOps, Random, Time;

3  use HPCCProblemSize;


6  param radix = 4;

8  param numVectors = 2;
9  type elemType = complex(128);


12 config const n = computeProblemSize(elemType, numVectors, returnLog2 = true);
13 const m = 2**n;

15 config const epsilon = 2.0 ** -51.0,
16              threshold = 16.0;

18 config const useRandomSeed = true,
19              seed = if useRandomSeed then SeedGenerator.clockMS else 314159265;

21 config const printParams = true,
22              printArrays = false,
23              printStats = true;


26 def main() {
27   printConfiguration();

29   const TwiddleDom: domain(1) distributed(Block) = [0..m/4);
30   var Twiddles: [TwiddleDom] elemType;

32   const ProblemDom: domain(1) distributed(Block) = [0..m);
33   var Z, z: [ProblemDom] elemType;

35   initVectors(Twiddles, z);

37   const startTime = getCurrentTime();

39   Z = conjg(z);
40   bitReverseShuffle(Z);
41   dfft(Z, Twiddles);

43   const execTime = getCurrentTime() - startTime;

45   const validAnswer = verifyResults(z, Z, Twiddles);
46   printResults(validAnswer, execTime);
47 }


50 def printConfiguration() {
51   if (printParams) then printProblemSize(elemType, numVectors, m);
52 }


55 def initVectors(Twiddles, z) {
56   computeTwiddles(Twiddles);
57   bitReverseShuffle(Twiddles);

59   fillRandom(z, seed);

61   if (printArrays) {
62     writeln("After initialization, Twiddles is: ", Twiddles, "\n");
```

```
63      writeln("z is: ", z, "\n");
64    }
65  }


68  def computeTwiddles(Twiddles) {
69    const numTwdls = Twiddles.numElements,
70          delta = 2.0 * atan(1.0) / numTwdls;

72    Twiddles(0) = 1.0;
73    Twiddles(numTwdls/2) = let x = cos(delta * numTwdls/2)
74                           in (x, x):elemType;
75    forall i in [1..numTwdls/2) {
76      const x = cos(delta*i),
77            y = sin(delta*i);
78      Twiddles(i)            = (x, y):elemType;
79      Twiddles(numTwdls - i) = (y, x):elemType;
80    }
81  }


84  def bitReverseShuffle(Vect: [?Dom]) {
85    const numBits = log2(Vect.numElements),
86          Perm: [i in Dom] Vect.eltType = Vect(bitReverse(i, revBits=numBits));
87    Vect = Perm;
88  }


91  def bitReverse(val: ?valType, revBits = 64) {
92    param mask = 0x0102040810204080;
93    const valReverse64 = bitMatMultOr(mask, bitMatMultOr(val:uint(64), mask)),
94          valReverse = bitRotLeft(valReverse64, revBits);
95    return valReverse: valType;
96  }


99  def dfft(A: [?ADom], W) {
100   const numElements = A.numElements;

102   for (str, span) in genDFTPhases(numElements, radix) {
103     forall (bankStart, twidIndex) in (ADom by 2*span, 0..) {
104       var wk2 = W(twidIndex),
105           wk1 = W(2*twidIndex),
106           wk3 = (wk1.re - 2 * wk2.im * wk1.im,
107                  2 * wk2.im * wk1.re - wk1.im):elemType;

109       forall lo in bankStart + [0..str) do
110         butterfly(wk1, wk2, wk3, A[[0..radix)*str + lo]);

112       wk1 = W(2*twidIndex+1);
113       wk3 = (wk1.re - 2 * wk2.re * wk1.im,
114              2 * wk2.re * wk1.re - wk1.im):elemType;
115       wk2 *= 1.0i;

117       forall lo in bankStart + span + [0..str) do
118         butterfly(wk1, wk2, wk3, A[[0..radix)*str + lo]);
119     }
120   }

122   const str = radix**log4(numElements-1);
123   if (str*radix == numElements) then
124     forall lo in [0..str) do
125       butterfly(1.0, 1.0, 1.0, A[[0..radix)*str + lo]);
126   else {
127     forall lo in [0..str) {
```

```
128       const a = A(lo),
129             b = A(lo+str);
130       A(lo)     = a + b;
131       A(lo+str) = a - b;
132     }
133 }


136 def genDFTPhases(numElements, radix) {
137   var stride = 1;
138   for 1..log4(numElements-1) {
139     const span = stride * radix;
140     yield (stride, span);
141     stride = span;
142   }
143 }


146 def butterfly(wk1, wk2, wk3, inout A:[1..radix]) {
147   var x0 = A(1) + A(2),
148       x1 = A(1) - A(2),
149       x2 = A(3) + A(4),
150       x3rot = (A(3) - A(4))*1.0i;

152   A(1) = x0 + x2;
153   x0 -= x2;
154   A(3) = wk2 * x0;
155   x0 = x1 + x3rot;
156   A(2) = wk1 * x0;
157   x0 = x1 - x3rot;
158   A(4) = wk3 * x0;
159 }


162 def log4(x) return logBasePow2(x, 2);


165 def verifyResults(z, Z, Twiddles) {
166   if (printArrays) then writeln("After FFT, Z is: ", Z, "\n");

168   Z = conjg(Z) / m;
169   bitReverseShuffle(Z);
170   dfft(Z, Twiddles);

172   if (printArrays) then writeln("After inverse FFT, Z is: ", Z, "\n");

174   var maxerr = max reduce sqrt((z.re - Z.re)**2 + (z.im - Z.im)**2);
175   maxerr /= (epsilon * n);
176   if (printStats) then writeln("error = ", maxerr);

178   return (maxerr < threshold);
179 }


182 def printResults(successful, execTime) {
183   writeln("Validation: ", if successful then "SUCCESS" else "FAILURE");
184   if (printStats) {
185     writeln("Execution time = ", execTime);
186     writeln("Performance (Gflop/s) = ", 5.0 * (m * n) / execTime / 1.0e-9);
187   }
188 }
```

D HPCC Problem Size Computation Code

```
module HPCCProblemSize {
  use Memory, Types;

  config const memRatio = 4;

  def computeProblemSize(type elemType, numArrays, returnLog2 = false) {
    const totalMem = + reduce Locale.physicalMemory(unit = Bytes),
          memoryTarget = totalMem / memRatio,
          bytesPerIndex = numArrays * numBytes(elemType);

    var numIndices = (memoryTarget / bytesPerIndex): int;

    var lgProblemSize = log2(numIndices);
    if (returnLog2) {
      numIndices = 2**lgProblemSize;
      if (numIndices * bytesPerIndex <= memoryTarget) {
        numIndices *= 2;
        lgProblemSize += 1;
      }
    }

    const smallestMem = min reduce Locale.physicalMemory(unit = Bytes);
    if ((numIndices * bytesPerIndex)/numLocales > smallestMem) then
      halt("System is too heterogeneous: blocked data won't fit into memory");

    return if returnLog2 then lgProblemSize else numIndices;
  }


  def printProblemSize(type elemType, numArrays, problemSize: ?psType) {
    const bytesPerArray = problemSize * numBytes(elemType),
          totalMemInGB = (numArrays * bytesPerArray:real) / (1024**3),
          lgProbSize = log2(problemSize):psType;

    write("Problem size = ", problemSize);
    if (2**lgProbSize == problemSize) {
      write(" (2**", lgProbSize, ")");
    }
    writeln();
    writeln("Bytes per array = ", bytesPerArray);
    writeln("Total memory required (GB) = ", totalMemInGB);
  }
}
```