Global HPCC Benchmarks in Chapel: STREAM Triad, Random Access, and FFT*
(Revision 1.1 — HPCC BOF, SC06)

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

Abstract

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. In this report, we provide Chapel implementations of the global HPC Challenge (HPCC) benchmarks for the STREAM Triad, Random Access, and FFT computations. We use our implementations to provide an introduction to Chapel, to illustrate its features, and to discuss its parallel and performance-oriented concepts. The codes listed in this paper compile and execute with the current version of the Chapel compiler. We do not provide performance results in this report due to the limited ability of the current version of the Chapel compiler to target parallel systems. Nevertheless, our solutions illustrate how combining global-view parallel programming with modern language features can support clean, performance-minded expression of parallel computations. During the coming year we will be continuing our compiler development with the intent of generating publishable performance results for our codes in time for next year's competition.

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 report, 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 0.3.4070). We provide "*Compiler Status Notes*" to call out code excerpts in which our preferred code for a benchmark differs from our submitted version due to limitations in the current Chapel compiler. There are seven such notes, five of them related to limitations in evaluating compile-time expressions, and the other two pertaining to generality in defining iterators.

Our development effort in Chapel has focused on providing a single-processor prototype of the language's features in order to support experimentation with Chapel and generate feedback on its specification. While our language design and compiler architecture have both been developed with distributed-memory execution and performance optimization opportunities in mind, minimal development effort has been invested toward these activities to date. As a result, this report does not contain performance results since they do not reflect our team's emphasis. Distributed-memory execution and performance optimizations will become our focus in early 2007, and we expect to have publishable performance results for our codes in time for next year's competition.

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

While our lack of performance results makes our submission untenable for winning the class 2 competition, we believe that this paper's description of Chapel, implementation of the benchmarks, and discussion of parallel execution issues will be of interest and value to the community. Moreover, though we do not report on performance, we have done our best to write versions of the benchmarks that should result in good performance next year, rather than optimizing strictly for elegance, clarity, and size without any regard for real-world execution issues. To this end, our report contains a discussion of the top performance-related issues for each benchmark and our strategy for achieving good performance in the future.

The rest of this report is organized as follows: In Section 2 we give an overview of our results. In Section 3 we provide an introduction to Chapel's motivating themes. In Section 4 we describe the coding conventions that we adopted in writing these benchmarks. Section 5 provides a high-level overview of our approach for implementing the three benchmarks, calling out key Chapel features in passing. Sections 6, 7, and 8 then consider our implementations for STREAM Triad, Random Access, and FFT in greater detail, describing our approach to implementing the codes as well as providing a tutorial introduction to the Chapel features that they utilize. Each of these sections builds on its predecessors, so concepts that are introduced for earlier benchmarks are treated as though familiar in later discussions. Each section also concludes with a brief summary of the highlights of our implementation. Section 9 describes our common module code, used for all three benchmarks to compute the default problem size as a function of total system memory. Next, section 10 discusses performance-related issues in the benchmarks and gives some insight into Chapel's benefits and challenges. Finally in Section 11, we summarize and provide a brief status report for the Chapel project. Our complete source listings are provided in the report's appendices.

2 Overview of Results

The following table categorizes and counts the number of lines of code utilized by our HPCC implementations:

Benchmark Code:	**STREAM**	**Random**		
line count type	**Triad**	**Access**	**FFT**	**Common**
Kernel computation	1	4 + 19 = 23	49	0
Kernel declarations	9	14 + 21 = 35	25	20
Total kernel	*10*	*18 + 40 = 58*	*74*	*20*
Initialization	9	1 + 0 = 1	21	0
Verification	6	15 + 0 = 15	13	0
Results and Output	29	18 + 0 = 18	16	12
Total Benchmark	***54***	***52 + 40 = 92***	***124***	***32***
Debug and Test	9	6 + 0 = 6	8	3
Blank	27	26 + 12 = 38	50	9
Total Program	*90*	*84 + 52 = 136*	*182*	*44*

The line counts for each benchmark are represented in a column of the table. The fourth data column represents the common *HPCCProblemSize* module that is shared by the benchmarks to compute and print the problem size. For the Random Access benchmark, each entry is expressed as a sum—the first value represents the benchmark module itself, the second represents the module used to compute the random number stream, and the final value is the sum for both modules.

The rows of the table are used to group the lines of code into various categories and running totals. The first two rows indicate the number of lines required to express the kernel of the computation and its supporting declarations, respectively. For example, in the STREAM Triad benchmark, writing the computation takes a single line of code, while its supporting variable and subroutine declarations require 9 lines of code. The next row presents the sum of these values to indicate the total number of lines required to express the kernel computation. The *Initialization*, *Verification*, and *Results and Output* rows indicate the number of lines devoted to: initializing the problem's data set; verifying that the results are correct; and computing and outputting results for timing and performance, respectively. These three rows are then combined with the previous subtotal in the next row to indicate the number of source lines used to implement the benchmark and output its results. The next two lines indicate: the number of lines added so that we could debug and test the programs as part of our nightly regression testing system; and the number of blank lines. These values are added to the previous subtotal to give the total number of lines in the programs, and are provided to serve as a checksum against the line number labels that appear in the appendices.

The next table compares the total *Source Lines of Code (SLOC)* for our Chapel codes with the standard HPCC reference implementations. The Chapel results are obtained by summing the *Total Benchmark* figure from the table

above for each code with that of the common module. The reference results are the sum of the *Framework* and *Parallel* numbers reported in the table from the HPCC website's FAQ.[1]

Benchmark Code:	STREAM Triad	Random Access	FFT
Reference	433	1668	1406
Chapel	86	124	156
Ratio	*5.03*	*13.45*	*9.01*

This table shows that our Chapel codes are approximately 5–13× smaller than the reference implementations. While shorter codes are not necessarily better or easier to understand, we believe that our Chapel implementations are not only succinct, but also clear representations of the benchmarks that will perform well as our compiler matures. The rest of this paper examines the codes qualitatively to complement the quantitative results in this section.

3 Chapel's Motivating Themes

One of Chapel's primary themes is to support general parallel programming using high-level abstractions. Chapel does this through its support for a *global-view programming model* that raises the level of abstraction for both data and control flow, as compared to parallel programming models currently used in production.

Global-view data structures are arrays and other data aggregates whose size and indices are expressed globally in spite of the fact that their implementations may be distributed across the memories of multiple nodes or *locales*.[2] This contrasts with most parallel languages used in practice, which require users to partition distributed data aggregates into per-processor chunks, either manually or using language abstractions. HPF and ZPL are two other recent parallel languages that support global-view data structures [15, 6], though in a more restricted form than Chapel.

A global view of control means that a user's program commences execution with a single logical thread of control and that additional parallelism is introduced through the use of specific language concepts. All parallelism in Chapel is implemented via multithreading, though these threads are created via high-level language concepts and managed by the compiler and runtime, rather than through explicit fork/join-style programming. An impact of this approach is that Chapel can express parallelism that is more general than the Single Program, Multiple Data (SPMD) model that today's most common parallel programming approaches use as the basis for their programming and execution models. Examples include Co-Array Fortran, Unified Parallel C (UPC), Titanium, HPF, ZPL, SHMEM, and typical uses of MPI [18, 11, 20, 15, 6, 2, 19, 12]. Our multithreaded execution model is perhaps most similar to that which is supported by the Cilk language or the Cray MTA's runtime libraries [13, 1]. Moreover, Chapel's general support for parallelism does not preclude the user from coding in an SPMD style if they wish.

Supporting general parallel programming also means targeting a broad range of parallel architectures. Chapel is designed to target a wide spectrum of HPC hardware including clusters of commodity processors and SMPs; vector, multithreading, and multicore processors; custom vendor architectures; distributed, shared, and shared address space memory architectures; and networks of any topology. Our portability goal is to have any legal Chapel program run correctly on all of these architectures, and for Chapel programs that express parallelism in an architecturally-neutral way to perform reasonably on all of them. Naturally, Chapel programmers can tune their codes to more closely match a particular machine's characteristics, though doing so may cause the program to be a poorer match for other architectures. In this report we present codes written in an architecturally-neutral manner and note places where they could be tuned to better match specific architectural characteristics including memory models and network capabilities.

A second theme in Chapel is to allow the user to optionally and incrementally specify where data and computation should be placed on the physical machine. We consider this control over program locality to be essential for achieving scalable performance on large machine sizes given current architectural trends. Such control contrasts with shared-memory programming models like OpenMP [7] which present the user with a flat memory model. It also contrasts with SPMD-based programming models in which such details are explicitly specified by the programmer on a process-by-process basis via the multiple cooperating program instances.

A third theme in Chapel is support for object-oriented programming (OOP), which has been instrumental in raising productivity in the mainstream programming community. Chapel supports traditional reference-based classes as well

[1] http://www.hpcchallenge.org/faq/index.html

[2] A *locale* in Chapel is a unit of the target architecture that supports computation and data storage. Locales are defined for an architecture such that a locale's threads will all have similar access times to any specific memory address. For commodity clusters, each of their (single-core) processors, multicore processors, or SMP nodes would be considered a locale.

as value classes. The programmer is not required to use an object-oriented style in their code, so that traditional Fortran and C programmers need not adopt a new programming paradigm in order to use Chapel effectively.[3]

Chapel's fourth theme is support for generic programming and polymorphism, allowing code to be written in a style that is generic across types, making it applicable to variables of multiple types, sizes, and precisions. The goal of these features is to support exploratory programming as in popular interpreted and scripting languages, and to support code reuse by allowing algorithms to be expressed without explicitly replicating them for each possible type. This flexibility at the source level is implemented by having the compiler create versions of the code for each required type signature rather than by relying on dynamic typing which would incur unacceptable runtime overheads.

Chapel's first two themes are designed to provide support for general, performance-oriented parallel programming through high-level abstractions. The second two themes are supported to help narrow the gulf between parallel programming languages and mainstream programming and scripting languages. The benchmarks in this paper illustrate Chapel's support for global-view programming, for locality control, and for generic programming. Due to the relatively straightforward style of parallel computation utilized in these benchmarks, this report does not demonstrate many of Chapel's features for more general parallelism and locality control. We also chose not to use OOP in our benchmark codes since we did not believe that the introduction of classes would greatly improve the benchmarks' clarity, generality, organization, or performance.

For a more complete introduction to Chapel, the reader is referred to our project's website[4], overview publications, and draft language specification [8, 5, 3].

4 Coding Conventions

In writing these codes, we used the Class 2 Official Specification as our primary guide for defining the computations. We studied and benefitted from the HPCC reference implementations as well as last year's 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
- 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
- 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 and realistic 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 procedures. In the Random Access and FFT benchmarks, we also abstracted the kernel of the computation into helper routines and iterators as appropriate.

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

[3]Note that many of Chapel's standard library capabilities are implemented using objects, so these may require Fortran and C programmers to utilize a method-invocation style of syntax. However, their use does not necessitate broader adoption of OOP methodologies.

[4]http://chapel.cs.washington.edu

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 several options. The language templates in this document are meant to be illustrative. They are intentionally incomplete and no replacement for the full Chapel language specification [8].

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, these constructs will necessarily fall into the degenerate cases of allocating data structures on a single locale and, for the single-processor locales we use, executing parallel loops and array statements using a single thread.

5 Benchmark Overviews

In this section, we give a brief overview of our three benchmark implementations, noting key Chapel features that they exercise in passing. The subsequent three sections walk through our implementations in greater detail, providing a thorough description of our approach while giving a tutorial introduction to Chapel.

5.1 STREAM Triad Overview

Our approach to the STREAM Triad benchmark is summarized by the following lines of code:

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

A = B + alpha * C;
```

This code excerpt presupposes the definition of two named values, *m* defining the problem size and *alpha* defining the scalar multiplication value for the Triad computation (α). It also refers to a named type, *elemType*, that represents the element type to be stored in the vectors.

The first line declares a constant named *ProblemSpace* that is defined to be a *domain*—a first-class representation of an index space, potentially distributed across the memories of multiple locales. In this instance, the index space is declared to be 1-dimensional and to be distributed across the machine's locales using the "Block" distribution. *ProblemSpace* is defined to describe the indices $\{1, 2, \ldots, m\}$.

The next line uses the *ProblemSpace* domain to declare three arrays—*A*, *B*, and *C*—used to represent the vectors, a, b, and c from the written specification. The domain's index set defines the size and shape of these arrays, and its distribution specifies the arrays' distributed implementation across the locale memories. The identifier *elemType* specifies the type of each array element (defined to be a 64-bit floating-point value in our implementation).

The final line expresses the computation itself, using whole-array syntax to specify the elementwise multiplications, additions, and assignments needed to perform the Triad computation. Whole-array operations like this one are implicitly parallel and each locale will perform the operations for the array elements that it owns, as defined by *ProblemSpace*'s distribution (since that was the domain used to define all three arrays).

5.2 Random Access Overview

Our approach to the Random Access benchmark is summarized by the following lines of code:

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

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

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

forall block in subBlocks(UpdateSpace) do
  for r in RAStream(block.numIndices, block.low) do
    T(r & indexMask) ^= r;
```

This code excerpt assumes the definition of three named values: m which defines the problem size, N_U which represents the number of updates (N_U), and *indexMask* which stores the bitmask used to index into the table. It also refers to two named types, *indexType* and *elemType*, used to represent the types that should be used to store the indices and elements of table, T, respectively.

The first two lines define a domain and array used to represent the distributed table, T, that is randomly accessed by the benchmark. The next line defines a second domain, *UpdateSpace* that represents the distribution of the table update work across the locale set. Note that *UpdateSpace* is not used to allocate any arrays, only to distribute the computation space.

The computation starts on the fourth line of code which initializes the table using a *forall expression*, representing the parallel loop: "for all indices i in $TableSpace$, assign T_i the value i." The final lines describe the random access computation using a nested loop defined by two iterators. The outer loop uses an iterator defined by the *Block* distribution to generate sub-blocks of work that can be performed in parallel. The inner loop uses the iterator `RAStream()`, defined elsewhere in our code, to generate the random stream of values for a given block of work. The body of the loop uses those values to update the table.

The specification of the random stream using an iterator allows different number generation techniques to be swapped in without modifying the computation itself. Additional parallelism can be introduced in the algorithm by rewriting the `RAStream()` iterator to return chunks of work rather than singleton indices, causing the table update operations to be *promoted* and result in implicit parallelism (as in the STREAM Triad benchmark).

5.3 FFT Overview

We chose to implement a radix-4 FFT in order to take advantage of its improved Flops-to-MemOps ratio. Our main loop for FFT is as follows:

```
for i in [2..log2(numElements)) by 2 {
  const m = radix*span,
        m2 = 2*m;

  forall (k,k1) in (ADom by m2, 0..) {
    var wk2 = W(k1),
        wk1 = W(2*k1),
        wk3 = (wk1.re - 2 * wk2.im * wk1.im,
               2 * wk2.im * wk1.re - wk1.im):complex;

    forall j in [k..k+span) do
      butterfly(wk1, wk2, wk3, A[j..j+3*span by span]);

    wk1 = W(2*k1+1);
    wk3 = (wk1.re - 2 * wk2.re * wk1.im,
           2 * wk2.re * wk1.re - wk1.im):complex;
    wk2 *= 1.0i;

    forall j in [k+m..k+m+span) do
      butterfly(wk1, wk2, wk3, A[j..j+3*span by span]);
  }
  span *= radix;
}
```

The outer loop iterates over the FFT phases using Chapel's support for *strided domains and sequences*. The next loop uses *zippered iteration* to traverse two iteration spaces simultaneously, the second of which is defined using the *indefinite arithmetic sequence* "`0..`" to indicate that it should start counting at zero and iterate as many times as the iterator with which it is zippered. The iterator indices, *k* and *k1* are expressed using a tuple-style variable declaration. The inner loops describe the parallelizable calls to the `butterfly()` routine, which has the following interface:

```
def butterfly(wk1, wk2, wk3, inout A:[1..radix]) { ... }
```

This routine takes three twiddle values of unspecified type and uses them to update values within a 4-element array, *A*. The argument types are unspecified, and are typically represented using Chapel's 128-bit complex type. However, by omitting these argument types, programmers can optimize calls that they know to use twiddle factors with zero components by passing in values of Chapel's real or pure imaginary types. Such calls would cause the compiler to clone the butterfly routine for these types, thereby eliminating the extraneous floating point operations against zero values that would be incurred by a routine written specifically for complex values.

The calls to `butterfly()` are also of interest because they pass a strided slice of *A* to the formal vector argument that was defined using the anonymous domain "[1..4]". This results in the creation of an *array view*, allowing the original array elements to be accessed within the routine using the local indices $1 \ldots 4$.

6 STREAM Triad in Chapel

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.

6.1 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 1–5 demonstrate how modules use one another:

```
use Time;
use Types;
use Random;

use HPCCProblemSize;
```

The first three lines use 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 9.

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 known at compile-time. The `param` keyword is used to define a *parameter*, which is a compile-time constant.[5] Parameter

[5]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.

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.

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 8 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.

Type Definitions Chapel supports the ability to created named type definitions using the `type` keyword. Line 9 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.[6]

STREAM Triad's Configuration Variables Lines 11–22 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

[6] 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.

`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 9 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.

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. As described in Section 3, this entry point is executed by a single logical thread. The `main()` subroutine for our program is defined on lines 25–43, 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 As mentioned in Section 5.1, Chapel's *domains* are a first-class language concept used to represent an index set.[7] 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 arbitrary or anonymous types.

[7] Chapel's domains are a generalization of ZPL's *region* concept[4].

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 28 as follows:

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

This declaration specifies that *ProblemSpace* is a constant domain. 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 $\{1, 2, \ldots, m\}$

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 29:

```
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 31, 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 33, 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 } [ 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 *sequence*, 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) will be used to implement the loop's iterations is left to the compiler and/or runtime by default.

The optional *index-vars* term declares one or more variables to store each value 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 35–39 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 an arithmetic sequence 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 arithmetic sequence 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 36 and 38:

```
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 cannot 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 37:

```
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 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. While the scalar *alpha* is conceptually stored in the memory of the locale on which the thread executing `main()` began, since it was declared to be `const`, it will 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 41–42:

```
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 46–51:

```
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 54–64, 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 were chosen to support the principle of least surprise.

Randomization The body of the `initVectors()` subroutine is defined on lines 55–63:

```
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.[8]

Forall Expressions and Reductions The next subroutine, `verifyResults()` is defined on lines 67–73:

```
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 [10] 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.[9] 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.

[8] 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.

[9] 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*."

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 72:

```
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 76–90:

```
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];
```

6.2 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.

7 Random Access in Chapel

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 7.1 while the random stream generation module is listed in Appendix B.2 and described in Section 7.2.

7.1 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 1–4 by listing the modules that it needs to `use`:

```
use Time;

use HPCCProblemSize;
use 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 (7.2) 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 7–9 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 11–23 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 = false,
              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 9, 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. 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 26–46 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.[10] In this instance, we use a 0-based index set because it simplifies the arithmetic required when indexing into the table.[11]

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 34–42 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 ranges, helping both the compiler and the reader understand the program's semantics.[12]

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 38–40:

[10] 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.

[11] 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.

[12] Put another way, 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.

```
forall block in subBlocks(UpdateSpace) do
  for r in RAStream(block.numIndices, block.low) do
    T(r & indexMask) ^= r;
```

This loop is an example of expressing parallelism in a simple, architecturally-neutral way, as alluded to in Section 3. Our goal is to create parallel work such that each locale has an appropriate number of threads, each performing a fraction of the locale's random updates. To this end, we create a parallel outer 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 uses an iterator, `subBlocks()` that takes a domain as its argument and generates sub-blocks of that domain which 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 its input argument, so for this loop, *block* represents a 1D arithmetic subdomain of *UpdateSpace*.

Compiler Status Note: This statement reflects the first line of code that we had to modify due to limitations in our compiler implementation. The loop ought to appear as follows:

```
forall block in UpdateSpace.subBlocks() do
```

where `subBlocks()` is an iterator defined for a domain by its distribution class. Unfortunately, our compiler currently does not support iterators as class methods. Thus, rather than making `subBlocks()` a method of our *Block* domain class, we had to make it a standalone iterator that takes the domain as its argument. This will be fixed in future versions of the compiler.

The inner loop uses a second iterator, `RAStream()` that we define in the *RARandomStream* module below. This iterator takes two arguments indicating the number of pseudo-random values to generate and the index of the first value. In this invocation, we provide these arguments using the standard domain methods `numIndices` and `low` to query the portion of *UpdateSpace* that a given thread is responsible for as described by the *block* subdomain.

The update itself is expressed in the body of the inner loop, using Chapel's bitwise-and (`&`) and -xor (`^`) 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*.[13]

This loop is an excellent demonstration of how iterators can be used to abstract complex looping structures away from loop bodies just as subroutines can be 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 sequence 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. This aggregate loop body could be implemented by the compiler using vector operations on a vector processor, 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 57–75. 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

[13] 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, so such operators could result in races in our current plan of record.

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 of 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 represents 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.

Verification Loops and Atomic Statements The verification computation is expressed in lines 60–67:

```
if (sequentialVerify) then
  for r in RAStream(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 the full number of updates, *N_U*. As we will see in the iterator's definition, the starting index argument has a default value of 0, allowing it to be elided here. 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.[14] 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.

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.

7.2 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.

[14] Note that the argument $i + 1$ is used to make our random number generator match the reference implementation precisely 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 for purposes of comparison.

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 {
  ...
}
```

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

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

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—Chapel's 64-bit unsigned integer type, `uint(64)`.

Compiler Status Note: Here is the second instance in which our current compiler did not support the code we would ultimately like to write. It would be preferable to define this type using *randWidth* as follows:

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

However, because *randType* is used by other modules and due to the current phase ordering of our parameter folding pass, this currently breaks the compilation. Future versions of the compiler will fix this issue.

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 third 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 precomputed at compile-time and potentially used for optimizations. However, we currently do not support domain and array `param` values, so 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:

```
iterator RAStream(numvals, start:randType = 0): randType {
  var val = getNthRandom(start);
  for i in 1..numvals {
    getNextRandom(val);
    yield val;
  }
}
```

Iterators are similar 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 from that point. Conceptually, an iterator returns a sequence of values, though most implementation strategies never require the sequence to be manifested in memory.

As described before, this iterator takes two arguments: *numvals* defines the number of values to generate and *start* specifies the index of the initial value. As described earlier, *start* has a default value of 0. It is also explicitly declared

to be a *randType*, because current Chapel semantics specify that arguments with default values infer their types from that value, which would cause *start* to only be a 32-bit `int`.[15]

Compiler Status Note: Our current compiler implementation cannot yet infer the return type of an iterator, so we specify it explicitly for this and subsequent iterators even though omitting it would be closer to the coding style we have adopted for this study.

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.[16] 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;
  ...
  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 operator (`%`).[17]

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. And finally, it 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 fifth 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.

[15] We are currently exploring this issue to determine whether default values can be interpreted as a constraint on the argument type rather than a precise specification of the type.

[16] 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.

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

7.3 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 `subblock()` iterator to create an appropriate amount of parallel work for the target architecture in a portable way. Using this feature of distributions allows the programmer 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.
- 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 sequences 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.
- 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.

8 FFT in Chapel

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

Our coverage of the FFT benchmark differs somewhat from that of the previous benchmarks due to the fact that the FFT algorithm can be approached in so many different ways and the impact of these approaches can be highly architecture-dependent. As a result, we focus primarily on creating a clean, parallel radix-4 specification of the computation. In our performance discussions (Section 10), we discuss a few variations on our basic approach that one might utilize to tune the code for shared- and distributed-memory architectures.

8.1 Detailed FFT Walk-through

Appendix C.1 contains the full code specifying our generic 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 1–25 and are similar to what we've seen in the previous codes. We excerpt a few lines of interest here:

```
use BitOps;
...
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. It remains to be seen whether or not users will make much direct use of this type, but in any case it is useful to the compiler and its ability to optimize complex numbers 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 28–49 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 41–43 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 and store the result into our working array. 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 57–67. 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 70–83 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):complex;
  forall i in [1..numTwdls/2) {
    const x = cos(delta*i),
          y = sin(delta*i);
    Twiddles(i)            = (x, y):complex;
    Twiddles(numTwdls - i) = (y, x):complex;
  }
}
```

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.

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. Though we do not use them much in these benchmarks, Chapel has general support for tuples, 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 as tuples, we support casts from 2-tuples of numeric values into complex types as a convenient way to define complex values component-wise.

Compiler Status Note: This code also contains our next implementation note. To be consistent with the rest of the benchmark, we would really like to cast these tuples using the *elemType* type alias that was created earlier. However, our current compiler only supports casts from tuples to complex values for the default "`complex`" type specifically. In this case this is not a problem because the default complex size is 128 bits, the same as *elemType*. This is a temporary workaround, and one that appears throughout this benchmark. It will be fixed in an upcoming version of the compiler.

The bitReverseShuffle() Routines The `bitReverseShuffle()` routine is defined on lines 86–91 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] index(Dom) = bitReverse(i, revBits = numBits),
        Temp = Vect(Perm);
  Vect = Temp;
}
```

The routine creates a permutation array, *Perm* that stores elements of type `index(Dom)`. As described in the Random Access walkthrough, this is a type that is constrained to represent indices from the domain *Dom* (declared here using an argument query variable). This declaration uses an array declaration style that combines the array definition syntax with the forall expression syntax 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 values of the permutation vector. In general, assignment to a variable that is an `index` type requires a dynamic bounds check unless the compiler can prove it away.

The next line declares a temporary array *Temp*, assigning it the permuted elements of *Vect* by promoting *Vect*'s scalar indexing method. The temporary array is then copied back into *Vect* and deallocated upon exiting the subroutine.

The helper subroutine `bitReverse()` is defined on lines 94–99:

```
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 102–140 and the helper routine `butterfly()` defined on lines 143–156. The main loop structure of `dfft()` is shown here:

```
for i in [2..log2(numElements)) by 2 {
  ...
  forall (k,k1) in (ADom by m2, 0..) {
    ...
    forall j in [k..k+span) do
      butterfly(wk1, wk2, wk3, A[j..j+3*span by span]);
    ...
    forall j in [k+m..k+m+span) do
```

```
        butterfly(wk1, wk2, wk3, A[j..j+3*span by span]);
      }
      ...
    }
```

This structure starts with a sequential outer loop over the FFT's phases. This loop uses Chapel's `by` operator to create an anonymous strided arithmetic domain. The `by` operator can be applied to sequences and domains to create strided subsequences and domains. Using a negative stride causes the indices to be traversed from high to low.

The next loop is parallel and uses a concept called *zippered iteration* to traverse two iterators simultaneously in an elementwise fashion. This is expressed by specifying a tuple of iterator expressions and a tuple of index variables. All iterator expressions that are zippered together must have the same size and shape. In this example, the second iterator expression is an *indefinite arithmetic sequence* due to the fact that it specifies a low bound but no high bound. This causes the sequence to be the same length as the other iterators with which it is zippered. Here the indefinite sequence is being zippered with a strided iteration over the data vector's domain, *ADom*, and it is being used to store the iteration number as a 0-based index using the variable *k1*. Obviously, the sequence's high bound could be computed explicitly by the programmer, but this notation reduces the chance of errors, eliminates the unnecessary math, and results in cleaner code.

The innermost loops express the butterfly computations which can be performed in parallel. These are expressed using calls to the helper `butterfly()` routine, passing twiddle values and an array slice of the vector *A*. This slice is expressed using a strided domain literal.

The rest of the `dfft()` routine computes twiddle values, loop bounds, and strides. There is also a cleanup phase in lines 130–139 that takes care of any final butterfly computations that are not handled by the main loop.

The Butterfly Routine The routine to compute a single radix-4 butterfly is defined in lines 143–156:

```
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:

```
butterfly(1.0, 1.0, 1.0, A[j..j+3*span by span]);
```

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 took 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 result in similar benefits (see Appendix C.2 for an example).

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.

8.2 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 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 will see in Section 10, FFT's access patterns raise some performance challenges for the compiler, yet we believe they are surmountable.

9 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.

9.1 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 6–28. Its interface is defined as follows:

```
def computeProblemSize(type elemType, numArrays, returnLog2 = false,
                       memRatio = 4) {
  ...
}
```

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 should appear first in the 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, whether or not the routine should compute a power-of-2 problem size, and the ratio of the total system memory required by the benchmark, set to the default of 4 (note that none of the benchmarks that we implemented require any ratio other than 4, but we decided to make this an argument for generality—in retrospect we probably should have made it a `config const` of the *HPCCProblemSize* module).

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 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. From it we use the `physicalMemory()` routine which takes a `locale` argument and returns the amount of local physical memory available to that locale. It also takes a second optional argument indicating the unit of memory that the result should be returned in. We use this routine at lines 8 and 23 in our code:

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

Each of these statements calls `physicalMemory()` in a promoted manner, passing in the entire *Locale* array rather than a single `locale` value. The resulting aggregate 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 (if the machine memory is heterogeneous, for example).

The halt routine If this test fails, the standard routine `halt()` is called in line 25:

```
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.[18]

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

The printProblemSize() Subroutine The `printProblemSize()` routine is defined on lines 31–44, and is fairly unremarkable—it does some simple computations to compute memory usage and then prints out the results. Perhaps 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 `numBytes()` and `log2()` routines in order to make sure that the operations performed on them are legal; otherwise the compiler will complain.[19] The first of these casts also represents the final change we had to make to the code from our ideal implementation. . .

Compiler Status Note: Because parameter values have special coercion properties in the language that permit them to interact with values of otherwise illegal types and sizes, we have the desire to define *param routines* that can be evaluated at compile time to create a `param` value (currently, parameter values can only be expressed using literals, built-in operators, and other parameter values). For example, given that `type` arguments in Chapel must be statically known, the `numBytes()` routine could be defined as a `param` subroutine that the compiler would (and

[18] 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.

[19] 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.

should) evaluate. In such a case, the casting of its return type to *psType* would be unnecessary because Chapel's coercion rules would allow it to be multiplied by *problemSize* directly.

10 Performance Discussion

In this section we discuss performance issues for each of the three benchmarks and indicate Chapel's anticipated implementation advantages and challenges as we begin to target large-scale architectures.

10.1 STREAM Triad Performance Notes

Once we have a distributed-memory implementation of Chapel and some additional scalar optimizations, we expect our STREAM Triad implementation to perform as well as the target architecture allows— constrained only by the rate of a locale's local memory accesses. The timed Triad computation is embarrassingly parallel, and its expression in Chapel makes this obvious through the use of whole-array operations on arrays declared using the same domain. This demonstrates one of the many advantages of domains, which is that they often provide the compiler with valuable information about the relative alignment of arrays in distributed memory [4].

Our main implementation challenge in obtaining good performance relates to the definition of the *Block* distribution itself. While our team has extensive experience implementing block distributions whose performance competes with or outperforms hand-coded benchmarks [4, 9], a major goal for Chapel is to write all of its standard distributions using the same mechanism that programmers would use to author their own distributions. This contrasts with prior language work in which compilers typically have special knowledge of standard distributions. Our challenge is to design a distribution interface that allows the compiler to implement computations using standard distributions without compromising performance. If we fail to meet this challenge, we can always fall back on the approach of embedding knowledge of standard distributions into the compiler; however, this would be unfortunate because it suggests that users may face a performance penalty if they need to use a distribution that Chapel does not provide.

Our final performance-related note for STREAM Triad relates to the fact that Chapel uses a multithreaded execution model to implement its program's semantics. Users who are accustomed to an SPMD execution model may worry that for embarrassingly parallel applications like STREAM Triad, the overhead of supporting a general multithreaded execution environment may be overkill since simpler SPMD execution is sufficient. While we believe that multithreaded execution is required in the next generation of parallel languages to support general parallel programming, we also recognize the efficiency advantages of the SPMD model due to its simplicity. For this reason, we expect to optimize our implementation for phases of computation in which statically scheduled threads suffice, either in the compiler-generated code or in its runtime support system.

Using STREAM Triad as a specific example, when the compiler analyzes the source code, it can trivially see that there is neither task parallelism nor explicit parallelism specified in the code. All parallelism comes from performing implicitly parallel operations on distributed arrays—whole-array operations, pseudo-random fills, and reductions. Given such information, the compiler may choose to generate code that schedules a single thread per core across the distributed machine, using them in an SPMD fashion. Recall that Chapel's global-view model of control specifies that the entry point is executed by a single *logical* thread, allowing an SPMD execution strategy to be used as long as the illusion of a single startup thread is maintained and the program's semantics allow it. Similarly, the compiler may use an SPMD execution strategy for one phase of the user's program and more general techniques for other phases if it can prove it's legal.

Alternatively, Chapel's runtime support for multithreading may be tuned so that when a single thread is running per core, the overheads required by more general multithreading capabilities are minimized. In the limit, this may allow multithreaded Chapel codes executing in an SPMD style to be competitive with explicit SPMD execution models. Even in this scenario it should be expected that compiler-generated information about the threading and communication requirements of a computation phase would help Chapel's runtime libraries shut down unnecessary sources of runtime overhead, such as services for creating remote threads or servicing one-sided communication requests on a commodity cluster.

10.2 Random Access Performance Notes

For most platforms, the limiting factor for Random Access performance is the hardware rather than the software. We anticipate this to be the case for our parallel Chapel implementation as well. Without significant optimization, the default implementation that we provide will utilize the full parallelism of a machine's processors, but will make

updates to the global table one element at a time, requiring a lot of fine-grain communication. For architectures that are tuned for message-passing workloads this is a worst-case scenario, while for others, having many small messages in-flight is not an issue. As we noted in the walkthrough, the iterators used to describe the computation can be modified to create chunks of work of varying sizes to amortize these overheads or to use vector scatter technologies (if available on the architecture). In addition, a programmer who is committed to tuning the computation for a specific architecture could restructure the loops, add new levels of parallelism, or batch updates manually in order to make the computation match the architecture's characteristics. For example, on an architecture with single-threaded processors and poor fine-grain communication, the programmer might choose to rewrite the parallel loop to oversubscribe each locale since the communication overheads are likely to dominate the costs of software multithreading.

Our second performance-related note for Random Access relates to iterators. Iterators are one of several language concepts in Chapel that define semantics without specifying a specific implementation approach. This provides the user with a nice abstraction while allowing the compiler to consider a palette of implementation options and select one that is well-suited for the specific case. For example, an iterator's definition and uses can be analyzed to select an implementation technique that is appropriate for the code. While iterator bodies can contain complicated control structures and multiple yield statements in the general case, and while they can be invoked in complicated ways such as zippered iteration, many common cases simply use iterators as a clean and modular way of expressing a loop. The `RAStream()` iterator in our Random Access code is one such example, defining a straightforward loop structure and invoking it in a straightforward way. In such cases, the compiler can simply inline the iterator's definition in place of the loop statement and then inline the loop's body into the iterator's yield statement. This results in no runtime overhead compared to writing the loop out explicitly, while supporting loop-level modularity and the ability for the iterator to be invoked in more complicated ways in other loops. For an overview of some of our early and more general implementation approaches for iterators, please refer to our published work [14].

Another Chapel concept that defines semantics without specifying an implementation mechanism is the atomic section. Atomic sections that contain arbitrary code can be implemented using a number of sophisticated techniques, including the bodies of research known as *Software* and *Hardware Transactional Memory* in which groups of memory operations appear to execute atomically from the point of view of other threads. Implementing Software Transactional Memory effectively on a distributed memory machine remains an open problem, and one that we are actively investigating. For simple atomic statements like the one contained in this code, however, simpler implementation techniques can be utilized. For example, by guarding the verification loop containing the atomic updates with the proper synchronization, the compiler can use locks to ensure that no two threads try to access the same table element simultaneously. In the common case where there are no conflicts, the overhead of these locks should be negligible compared to the overhead of the fine-grain remote accesses. Moreover, on architectures with support for *atomic memory operations (AMOs)* or *compare-and-swap (CAS)* instructions, the Chapel compiler can utilize these capabilities to further reduce overheads.

The final performance issue we discuss here relates to inlining. In the *RARandomStream* module, we chose to factor the computation of the stream's next random number into a subroutine rather than replicating the obscure expression throughout the module. Doing so may result in a slight performance hit to make the function call, so we can either hope that the compiler will inline the routine automatically due to its size and simplicity, or we can add a directive that will force the compiler to inline the routine. In the current version of our compiler, this is done via:

```
pragma "inline"
def getNextRandom( ... ) { ... }
```

However, for the long-term we have not yet determined whether inlining will be specified via a pragma or a keyword, nor have we settled on a final pragma syntax. For these reasons, we chose not to include this performance-oriented directive in our proposed solution.

10.3 FFT Performance Notes

As stated in the FFT walkthrough, it is very challenging to express a global 1D FFT that is clear while also being well-tuned for general parallel architectures—there are simply too many degrees of freedom and too many architectural characteristics that can impact performance. In our approach, we have implemented the computation cleanly and expressed both outer- and inner-loop concurrency in order to maximize the parallelism available to the compiler. An aggressive compiler might recognize that the trip counts of these loops vary with the FFT's phases and create specialized versions of `dfft()`'s loops to optimize for outer- and inner-loop parallelism, or a blend of the two. However, even in such a case, the target architecture's memory model and network can still play a key role in how the performance-minded user might perform additional optimizations to maximize efficiency.

At one end of the spectrum, even on a parallel machine with a flat shared memory like the Cray MTA, the clean expression of the computation as given here is unlikely to perform optimally. In Appendix C.2, we provide an alternate Chapel implementation of FFT that is based on last year's HPCC submission for the MTA2 [16]. In that implementation, several specialized versions of the inner loops were created in order to maximize outer-loop parallelism, take advantage of twiddle values with zero components and tune for the MTA's characteristics. While we might hope that our more straightforward implementation of the benchmark would come close to achieving the performance of this specialized version, it will always be the case that programmers willing to expend additional effort to tune for a given platform are likely to produce better performance. Optimizations such as those expressed in this implementation would be likely to improve performance for flat shared-memory contexts such as an SMP node, multicore processor, or the MTA.

On the other end of the spectrum is the distributed-memory case. Here, our implementation faces the problem that in later phases of the computation, the array slices become increasingly spread out, accessing values stored in the memories of one or more remote locales. As in the Random Access benchmark, with no optimization this is likely to result in fine-grain messages that exercise the machine's ability to handle such communications efficiently. Recent work by John Mellor-Crummey's team on the Rice D-HPF compiler strives to statically analyze loops similar to the ones in our solution so that the programmer can express the computation naturally while having the compiler generate an efficient FFT implementation [17]. In their work, they analyze loop bounds and array access patterns in order to create specialized versions of the loops for butterflies that are completely local or distributed. In the latter case, communication is overlapped with computation in order to bring remote values into a locale's memory in a way that hides the communication latency. We hope to duplicate their success in the Chapel compiler and believe that Chapel's support for domains and index types should support such analysis cleanly.

Alternatively, the Chapel programmer can explicitly manage the array's distribution in order to minimize communication. For example, by *redistributing* the domain representing the problem space from a *Block* to a *Cyclic* distribution at the appropriate stage in the computation, accesses to values that would have been remote in the latter phases can be localized (given an appropriate number of processors). This approach effectively implements the common technique of computing the 1D FFT using a 2D representation that is blocked in one dimension, and then transposing it partway through the computation to localize accesses.

Of course, the 2D approach to 1D FFTs can also be coded explicitly in Chapel by declaring 2D domains and arrays that are distributed in one dimension and local in the other. After the local butterflies have been exhausted, the array can be transposed or the domain can be redistributed in order to localize the butterflies in the second dimension. We have coded several of these variations in Chapel, but omit them from this report due to its length—such an exploration merits a report of its own.

We conclude this discussion of FFT performance notes by repeating our mantra: The FFT is a rich computation with many possible approaches that are sensitive to architectural details. As we target large-scale machines, our hope is that as we can get acceptable portable performance from our baseline implementation without requiring the algorithm to be cluttered beyond recognition. We also hope to demonstrate the ability for a user to optimize performance further for a particular architecture by performing the various rewrites described in this section.

11 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 (version 0.3.4070). The Chapel compiler is written in C++ and utilizes a Chapel-to-C compilation strategy for portability, allowing us to target any UNIX-like environment that supports a standard C compiler. Our compiler currently builds, compiles, and generates executables for Linux, Linux64, Cygwin, SunOS, and Mac OS X platforms. We run a nightly regression test suite of over 1300 tests on Linux, Linux64, Cygwin, and SunOS workstations.

Our Chapel compiler development advances on a daily basis. Our near-term development strategy has two phases. During the first phase, our target is a Unix-like environment with single-locale execution in order to demonstrate most of Chapel's features in a shared-memory context. This will enable experimentation with the Chapel language by Cray developers and external reviewers. In early 2007 we will begin the second phase of development by adding full parallel execution and support for targeting distributed-memory machines. We plan to minimize hardware and software assumptions during this phase in order to support portability of the implementation (*e.g.*, commodity Linux clusters will be a likely target). We plan to release this version of the compiler in 2007.

In conclusion, it is worth noting that while these 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 idioms that they illustrate. In particular, none of these benchmarks required any significant task 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 write similar tutorial studies for sample computations that exercise a broader range of Chapel's features.

Acknowledgments The authors would like to thank the 2005 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, Shannon Hoffswell, Roxana Diaconescu, Mark James, Mackale Joyner, and Robert Bocchino.

References

[1] Gail Alverson, Simon Kahan, Richard Korry, Cathy McCann, and Burton Smith. Scheduling on the Tera MTA. In D. G. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, volume 949 of *LNCS*, pages 19–44. Springer Verlag, April 1995.

[2] Ray Barriuso and Allan Knies. SHMEM user's guide for C. Technical report, Cray Research Inc., June 1994.

[3] David Callahan, Bradford L. Chamberlain, and Hans Zima. The Cascade High Productivity Language. In *Ninth International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS'04)*, pages 52–60. April 2004.

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

[5] Bradford L. Chamberlain, David Callahan, and Hans P. Zima. Parallel programmability and the Chapel language. *to appear in an upcoming special issue of the International Journal of High Performance Computing Applications on High Productivity Programming Languages and Models*, 2007.

[6] Bradford L. Chamberlain, Sung-Eun Choi, Steven J. Deitz, and Lawrence Snyder. The high-level parallel language ZPL improves productivity and performance. In *IEEE International Workshop on Productivity and Performance in High-End Computing*, 2004.

[7] Rohit Chandra, Ramesh Menon, Leo Dagum, David Kohr, Dror Maydan, and Jeff McDonald. *Parallel Programming in OpenMP*. Morgan Kaufmann, October 2000.

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

[9] Steven J. Deitz. *High-Level Programming Language Abstractions for Advanced and Dynamic Parallel Computations*. PhD thesis, University of Washington, 2005.

[10] 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.

[11] Tarek El-Ghazawi, William Carlson, Thomas Sterling, and Katherine Yelick. *UPC: Distributed Shared-Memory Programming*. Wiley-Interscience, June 2005.

[12] William Gropp, Steven Huss-Lederman, Andrew Lumsdaine, Ewing Lusk, Bill Nitzberg, William Saphir, and Marc Snir. *MPI: The Complete Reference, volume 2*. Scientific and Engineering Computation. MIT Press, September 1998.

[13] Supercomputing Technologies Group. *Cilk 5.3.2 Reference Manual*. MIT Laboratory for Computer Science, November 2001.

[14] Mackale Joyner, Steven J. Deitz, and Bradford L. Chamberlain. Iterators in Chapel. In *Eleventh International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS'06)*. April 2006.

[15] Charles H. Koelbel, David B. Loveman, Robert S. Schreiber, Guy L. Steele, Jr., and Mary E. Zosel. *The High Performance Fortran Handbook*. Scientific and Engineering Computation. MIT Press, September 1996.

[16] Petr Konecny, Simon Kahan, and John Feo. SC05 HPCChallenge class 2 award—Cray MTA2. November 2005.

[17] John Mellor-Crummey. Personal communication.

[18] Robert W. Numerich and John Reid. Co-array Fortran for parallel programming. *SIGPLAN Fortran Forum*, 17(2):1–31, 1998.

[19] Marc Snir, Steve Otto, Steve Huss-Lederman, David Walker, and Jack Dongarra. *MPI: The Complete Reference, volume 1*. Scientific and Engineering Computation. MIT Press, 2nd edition, September 1998.

[20] Kathy Yelick, Luigi Semenzato, Geoff Pike, Carleton Miyamato, Ben Liblit, Arvind Krishnamurthy, Paul Hilfinger, Susan Graham, David Gay, Phil Colella, and Alex Aiken. Titanium: A high-performance Java dialect. *Concurrency: Practice and Experience*, 10(11–13):825–836, September 1998.

A STREAM Triad Chapel Code

```
use Time;
use Types;
use 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");
```

```
63    }
64  }


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

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

72    return (infNorm <= epsilon);
73  }


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

87      const GBPerSec = numVectors * numBytes(elemType) * (m/minTime) * 1.0e-9;
88      writeln("Performance (GB/s) = ", GBPerSec);
89    }
90  }
```

B Random Access Chapel Code

B.1 Random Access Computation

```
use Time;

use HPCCProblemSize;
use 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 = false,
             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;

  forall block in subBlocks(UpdateSpace) do
    for r in RAStream(block.numIndices, block.low) 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
```

```
61      for r in RAStream(N_U) do
62        T(r & indexMask) ^= r;
63    else
64      forall i in UpdateSpace {
65        const r = getNthRandom(i+1);
66        atomic T(r & indexMask) ^= r;
67      }

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

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

74    return numErrors <= (errorTolerance * N_U);
75  }


78  def printResults(successful, execTime) {
79    writeln("Validation: ", if successful then "SUCCESS" else "FAILURE");
80    if (printStats) {
81      writeln("Execution time = ", execTime);
82      writeln("Performance (GUPS) = ", N_U / execTime * 1.0e-9);
83    }
84  }
```

B.2 Random Access Random Stream Generator

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

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


  iterator RAStream(numvals, start:randType = 0): randType {
    var val = getNthRandom(start);
    for i in 1..numvals {
      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);
  }


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

C FFT Chapel Code

C.1 1D radix-4 FFT Chapel Code

```
use BitOps;
use Random;
use Time;

use HPCCProblemSize;


param radix = 4;

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


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

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

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

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


def main() {
  printConfiguration();

  const 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;

  initVectors(Twiddles, z);

  const startTime = getCurrentTime();

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

  const execTime = getCurrentTime() - startTime;

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


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


def initVectors(Twiddles, z) {
  computeTwiddles(Twiddles);
  bitReverseShuffle(Twiddles);
```

```
61    fillRandom(z, seed);
63    if (printArrays) {
64      writeln("After initialization, Twiddles is: ", Twiddles, "\n");
65      writeln("z is: ", z, "\n");
66    }
67  }
70  def computeTwiddles(Twiddles) {
71    const numTwdls = Twiddles.numElements,
72          delta = 2.0 * atan(1.0) / numTwdls;
74    Twiddles(0) = 1.0;
75    Twiddles(numTwdls/2) = let x = cos(delta * numTwdls/2)
76                           in (x, x):complex;
77    forall i in [1..numTwdls/2) {
78      const x = cos(delta*i),
79            y = sin(delta*i);
80      Twiddles(i)            = (x, y):complex;
81      Twiddles(numTwdls - i) = (y, x):complex;
82    }
83  }
86  def bitReverseShuffle(Vect: [?Dom]) {
87    const numBits = log2(Vect.numElements),
88          Perm: [i in Dom] index(Dom) = bitReverse(i, revBits = numBits),
89          Temp = Vect(Perm);
90    Vect = Temp;
91  }
94  def bitReverse(val: ?valType, revBits = 64) {
95    param mask = 0x0102040810204080;
96    const valReverse64 = bitMatMultOr(mask, bitMatMultOr(val:uint(64), mask)),
97          valReverse = bitRotLeft(valReverse64, revBits);
98    return valReverse: valType;
99  }
102 def dfft(A: [?ADom], W) {
103   const numElements = A.numElements;
104   var span = 1;
106   for i in [2..log2(numElements)) by 2 {
107     const m = radix*span,
108           m2 = 2*m;
110     forall (k,k1) in (ADom by m2, 0..) {
111       var wk2 = W(k1),
112           wk1 = W(2*k1),
113           wk3 = (wk1.re - 2 * wk2.im * wk1.im,
114                  2 * wk2.im * wk1.re - wk1.im):complex;
116       forall j in [k..k+span) do
117         butterfly(wk1, wk2, wk3, A[j..j+3*span by span]);
119       wk1 = W(2*k1+1);
120       wk3 = (wk1.re - 2 * wk2.re * wk1.im,
121              2 * wk2.re * wk1.re - wk1.im):complex;
122       wk2 *= 1.0i;
124       forall j in [k+m..k+m+span) do
125         butterfly(wk1, wk2, wk3, A[j..j+3*span by span]);
```

```
126     }
127     span *= radix;
128   }

130   if (span*radix == numElements) then
131     forall j in [0..span) do
132       butterfly(1.0, 1.0, 1.0, A[j..j+3*span by span]);
133   else
134     forall j in [0..span) {
135       const a = A(j),
136             b = A(j+span);
137       A(j)      = a + b;
138       A(j+span) = a - b;
139     }
140 }


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

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


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

162   Z = conjg(Z) / m;
163   bitReverseShuffle(Z);
164   dfft(Z, Twiddles);

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

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

172   return (maxerr < threshold);
173 }


176 def printResults(successful, execTime) {
177   writeln("Validation: ", if successful then "SUCCESS" else "FAILURE");
178   if (printStats) {
179     writeln("Execution time = ", execTime);
180     writeln("Performance (Gflop/s) = ", 5.0 * (m * n) / execTime / 1.0e-9);
181   }
182 }
```

C.2 Chapel Version of HPCC'05 1D radix-4 FFT for the MTA

In this section, we provide an alternate Chapel implementation of FFT, based on last year's HPC Challenge implementation for the Cray MTA [16]. This version of the code is provided to suggest some of the transformations that an aggressive performance-minded programmer might utilize to best take advantage of architectural characteristics. In this code, iterations are hoisted out of loops in order to special-case known twiddle factors, and the outer and inner loops are interchanged for certain phases to maximize parallelism.

```
1  use BitOps;
2  use Random;
3  use Time;

5  use HPCCProblemSize;


8  param radix = 4;

10 param numVectors = 2;
11 type elemType = complex(128);


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

17 config const epsilon = 2.0 ** -51.0,
18              threshold = 16.0;

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

23 config const printParams = true,
24              printArrays = false,
25              printStats = true;


28 def main() {
29   printConfiguration();

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

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

37   initVectors(Twiddles, z);

39   const startTime = getCurrentTime();

41   Z = conjg(z);
42   bitReverseShuffle(Z);
43   dfft(Z, Twiddles);

45   const execTime = getCurrentTime() - startTime;

47   const validAnswer = verifyResults(z, Z, Twiddles);
48   printResults(validAnswer, execTime);
49 }


52 def dfft(Z, Twiddles) {
53   cft1st(Z, Twiddles);

55   var span = radix;

57   for i in [4..n) by 2 {
```

```
58      if (i <= n/2) then
59        cftmd1(span, Z, Twiddles);
60      else
61        cftmd2(span, Z, Twiddles);
62      span *= radix;
63    }

65    if (radix*span == Z.numElements) then
66      forall j in [0..span) do
67        butterfly(1.0, 1.0, 1.0, Z[j..j+3*span by span]);
68    else
69      forall j in [0..span) {
70        const a = Z(j),
71              b = Z(j+span);
72        Z(j)      = a + b;
73        Z(j+span) = a - b;
74      }
75  }


78  def printConfiguration() {
79    if (printParams) then printProblemSize(elemType, numVectors, m);
80  }


83  def initVectors(Twiddles, z) {
84    computeTwiddles(Twiddles);
85    bitReverseShuffle(Twiddles);

87    fillRandom(z, seed);

89    if (printArrays) {
90      writeln("After initialization, Twiddles is: ", Twiddles, "\n");
91      writeln("z is: ", z, "\n");
92    }
93  }


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

100   Twiddles(0) = 1.0;
101   Twiddles(numTwdls/2) = let x = cos(delta * numTwdls/2)
102                          in (x, x):complex;
103   forall i in [1..numTwdls/2) {
104     const x = cos(delta*i),
105           y = sin(delta*i);
106     Twiddles(i)            = (x, y):complex;
107     Twiddles(numTwdls - i) = (y, x):complex;
108   }
109 }


112 def bitReverseShuffle(Vect: [?Dom]) {
113   const numBits = log2(Vect.numElements),
114         Perm: [i in Dom] index(Dom) = bitReverse(i, revBits = numBits),
115         Temp = Vect(Perm);
116   Vect = Temp;
117 }


120 def bitReverse(val: ?valType, revBits = 64) {
121   param mask = 0x0102040810204080;
122   const valReverse64 = bitMatMultOr(mask, bitMatMultOr(val:uint(64), mask)),
```

```
123         valReverse = bitRotLeft(valReverse64, revBits);
124   return valReverse: valType;
125 }


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

131   Z = conjg(Z) / m;
132   bitReverseShuffle(Z);
133   dfft(Z, Twiddles);

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

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

141   return (maxerr < threshold);
142 }


145 def printResults(successful, execTime) {
146   writeln("Validation: ", if successful then "SUCCESS" else "FAILURE");
147   if (printStats) {
148     writeln("Execution time = ", execTime);
149     writeln("Performance (Gflop/s) = ", 5.0 * (m * n) / execTime / 1.0e-9);
150   }
151 }


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

160   A(1) = x0 + x2;
161   x0 -= x2;
162   A(3) = wk2 * x0;
163   x0 = x1 + x3rot;
164   A(2) = wk1 * x0;
165   x0 = x1 - x3rot;
166   A(4) = wk3 * x0;
167 }


170 def cft1st(A, W) {
171   var x0 = A(0) + A(1),
172       x1 = A(0) - A(1),
173       x2 = A(2) + A(3),
174       x3rot = (A(2) - A(3))*1.0i;
175   const wk1r = W(1).re;

177   A(0) = x0 + x2;
178   A(2) = x0 - x2;
179   A(1) = x1 + x3rot;
180   A(3) = x1 - x3rot;

182   x0 = A(4) + A(5);
183   x1 = A(4) - A(5);
184   x2 = A(6) + A(7);
185   const x3 = A(6) - A(7);
186   A(4) = x0 + x2;
187   A(6) = (x2.im - x0.im, x0.re - x2.re):complex;
```

```
188   x0 = x1 + x3*1.0i;
189   A(5) = wk1r * (x0.re - x0.im, x0.re + x0.im):complex;
190   x0 = (x3.im + x1.re, x3.re - x1.im):complex;
191   A(7) = wk1r * (x0.im - x0.re, x0.im + x0.re):complex;

193   forall (j,k1) in ([8..A.numElements) by 8, 1..) {
194     var wk2 = W(k1),
195         wk1 = W(2*k1),
196         wk3 = (wk1.re - 2* wk2.im * wk1.im,
197                2 * wk2.im * wk1.re - wk1.im):complex;

199     butterfly(wk1, wk2, wk3, A[j..j+3]);

201     wk1 = W(2*k1+1);
202     wk3 = (wk1.re - 2*wk2.re * wk1.im,
203            2*wk2.re * wk1.re - wk1.im):complex;
204     wk2 = wk2*1.0i;
205     butterfly(wk1, wk2, wk3, A[j+4..j+7]);
206   }
207 }


210 def cftmd0(span, A, W) {
211   const wk1r = W(1).re,
212         m = radix*span;

214   forall j in [0..span) do
215     butterfly(1.0, 1.0, 1.0, A[j..j+3*span by span]);

217   forall j in [m..m+span) do
218     butterfly((wk1r, wk1r):complex, 1.0i, (-wk1r, wk1r):complex,
219               A[j..j+3*span by span]);
220 }


223 def cftmd1(span, A, W) {
224   const m = radix*span,
225         m2 = 2*m;

227   cftmd0(span, A, W);
228   forall (k,k1) in ([m2..A.numElements) by m2, 1..) {
229     var wk2 = W(k1),
230         wk1 = W(2*k1),
231         wk3 = interpIm(wk1, wk2);
232     for j in [k..k+span) do
233       butterfly(wk1, wk2, wk3, A[j..j+3*span by span]);

235     wk1 = W(2*k1+1);
236     wk3 = interpRe(wk1, wk2);

238     for j in [k+m..k+m+span) do
239       butterfly(wk1, wk2*1.0i, wk3, A[j..j+3*span by span]);
240   }
241 }


244 def cftmd2(span, A, W) {
245   const m = radix*span,
246         m2 = 2*m,
247         numElems = A.numElements;

249   cftmd0(span, A, W);
250   if (m2 >= numElems) return;
251   if (m2 >= numElems / 8) {
252     cftmd21(span, A, W);
```

```
253     return;
254   }

256   forall j in [0..span) {
257     forall (k,k1) in ([m2..numElems) by m2, 1..) {
258       const wk2 = W(k1),
259             wk1 = W(k1 + k1),
260             wk3 = interpIm(wk1, wk2);
261       butterfly(wk1, wk2, wk3, A[j+k..j+k+3*span by span]);
262     }

264     forall (k,k1) in ([m2..numElems) by m2, 1..) {
265       const wk2 = W(k1),
266             wk1 = W(2*k1 + 1),
267             wk3 = interpRe(wk1, wk2);
268       wk2 = wk2*1.0i;

270       butterfly(wk1, wk2, wk3, A[j+k+m..j+k+m+3*span by span]);
271     }
272   }
273 }


276 def cftmd21(span, A, W) {
277   const m = radix*span,
278         m2 = 2*m;

280   for (k,k1) in ([m2..A.numElements) by m2, 1..) {
281     var wk2 = W(k1),
282         wk1 = W(2*k1),
283         wk3 = interpIm(wk1, wk2);

285     forall j in [k..k+span) do
286       butterfly(wk1, wk2, wk3, A[j..j+3*span by span]);

288     wk1 = W(2*k1 + 1);
289     wk3 = interpRe(wk1, wk2);
290     wk2 = wk2*1.0i;

292     forall j in [k+m..k+m+span) do
293       butterfly(wk1, wk2, wk3, A[j..j+3*span by span]);
294   }
295 }


298 def interpIm(a, b)
299   return (a.re - 2*b.im*a.im, 2*b.im*a.re - a.im):complex;


302 def interpRe(a, b)
303   return (a.re - 2*b.re*a.im, 2*b.re*a.re - a.im):complex;
```

D HPCC Problem Size Computation Code

```
module HPCCProblemSize {
  use Memory;
  use Types;


  def computeProblemSize(type elemType, numArrays, returnLog2 = false,
                         memRatio = 4) {
    const totalMem = + reduce physicalMemory(Locale, unit = Bytes),
          memoryTarget = totalMem / memRatio,
          bytesPerIndex = numArrays * numBytes(elemType);

    var numIndices = memoryTarget / bytesPerIndex;

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

    const smallestMem = min reduce physicalMemory(Locale, 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): psType,
          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);
  }
}
```