Supporting Array Programming in X10

David Grove
IBM Research
groved@us.ibm.com

Josh Milthorpe
Australian National University
josh.milthorpe@anu.edu.au

Olivier Tardieu
IBM Research
tardieu@us.ibm.com

Abstract

Effective support for array-based programming has long been one of the central design concerns of the X10 programming language. After significant research and exploration, X10 has adopted an approach based on providing arrays via user definable and extensible class libraries. This paper surveys the range of array abstractions available to the programmer in X10 2.4 and describes the key language features and language implementation techniques necessary to make efficient and productive implementations of these abstractions possible.

Categories and Subject Descriptors D.3.2 [*Language Classifications*]: [Object-oriented languages]; D.3.2 [*Language Classifications*]: [Concurrent, distributed, and parallel languages]; E.1.3 [*Data Structures*]: Arrays

Keywords X10, arrays, language design, frameworks

1. Introduction

From the very beginning of the X10 project in 2004, array programming has been a central concern. An early X10 language paper [5] describes five major design decisions that led to X10, one of which was to

Include a rich array sub-language that supports dense and sparse distributed multi-dimensional arrays.

This decision was motivated by the desire to productively support high performance computing domains that typically include significant amounts of array processing.

In the ensuing decade, a variety of approaches to supporting array-based programming have been explored within X10. Initially, the project focused on defining a rich array sub-language in which a single compiler-supported array abstraction would support the entire range of possible usage scenarios. Although expressive, notationally concise, and flexible, achieving acceptable levels of performance for a single unified array sub-language was an elusive goal. In particular, making "one size fits all" implementation choices tended to yield array implementations that were simultaneously poorly performing for very common simple usage scenarios and not quite flexible enough to support the most complex scenarios. As a result of this experience, the project focus gradually shifted to identifying the core language mechanisms and implementation support necessary to allow an extensible collection of fit-to-purpose array abstractions to be defined entirely at the X10 language level as class libraries.

In this paper we will describe the current state of array-based programming in X10 and what we believe are the fundamental language and implementation building blocks needed to make a class library based approach to array programming effective. The core of the paper is a tour through the spectrum of array abstractions available to be used with X10 2.4. All of the frameworks and libraries described in this paper are available as either part of the X10 core class library or as additional open source modules available from `http://x10-lang.org`. We encourage the interested reader to download the system and explore these ideas further via the code!

We begin by briefly introducing the X10 programming language and enumerating the language features and implementation capabilities that we have found essential to our approach. In Section 3 we describe the three array abstractions included in the X10 2.4 core class libraries: `x10.lang.Rail`, `x10.array`, and `x10.regionarray`, and motivate why we believe all three are necessary. To illustrate the flexibility of the class library approach, Section 4 describes the X10 Global Matrix Library, an implementation of dense and sparse distributed matrices that provides a high-level sequential API to parallel distributed matrix operations. Additional scenarios of application-specific array abstractions are detailed in Section 5. Finally Section 6 discusses related work and Section 7 describes some possible avenues of future work.

2. X10 Language Features

2.1 X10 Overview

The X10 programming language [5, 15] has been developed as a simple, clean, but powerful and practical programming model for scale out computation. Its underlying programming model, the APGAS (Asynchronous Partitioned Global Address Space) programming model [13], is organized around the two notions of *places* and *asynchrony*. A place is an abstraction of shared, mutable data and worker threads operating on the data, typically realized as an operating system process. A single APGAS computation may consist of hundreds or potentially tens of thousands of places. Asynchrony is provided through a single block-structured control construct, `async S`. If `S` is a statement, then `async S` is a statement that executes `S` in a separate thread of control (*activity* or *task*). Dually, `finish S` executes `S`, and waits for all tasks spawned (recursively) during the execution of `S` to terminate, before continuing. Memory locations in one place can contain references (*global refs*) to locations at other places. To use a global ref, the `at (p) S` statement must be used. It permits the current task to change its place of execution to `p`, execute `S` at `p` and return, leaving behind tasks that may have been spawned during the execution of `S`. The termination of these tasks is detected by the `finish` within which the `at`

ARRAY '14, June 12-13 2014, Edinburgh, United Kingdom.
Copyright © 2014 ACM 978-1-4503-2937-8/14/06...$15.00.
http://dx.doi.org/10.1145/2627373.2627380

statement is executing. The values of variables used in `S` but defined outside `S` are serialized, transmitted to `p`, and de-serialized to reconstruct a binding environment in which `S` is executed. Constructs are provided for unconditional (`atomic S`) and conditional (`when (c) S`) atomic execution.

The sequential core language of X10 is a strongly-typed, garbage-collected, class-based, object-oriented programming language with single-class multiple-interface inheritance. To support the construction of large software systems and reusable frameworks, it includes a generic type system, support for closure literals and function types, user-definable operators, Java-style non-resumptive exceptions, and advanced type system features such as constraints.

X10 is implemented with two backends – on the *managed* backend, X10 compiles into Java and runs on (a cluster of) JVMs, on the *native* backend; X10 compiles into C++ and generates a native binary for execution on scale-out systems. As discussed in later sections of the paper, the desire to make code efficient on both backends has influenced the design of some array frameworks in X10, most notably `Rail` and the Global Matrix Library.

More information on X10 can be found online at `http://x10-lang.org` including the language specification [14], programmer's guide [16], and a collection of tutorials and sample programs.

2.2 Useful Language Features

Constructing any class library in X10 will naturally utilize core object-oriented language features such as classes, interfaces, inheritance and packages to control member visibility. Container types such as arrays will also utilize X10's generic type system. Finally, function types enable the succinct definition and use of higher order operations such as element iteration, map, and reduce.

We have made extensive use of two additional language features when building array libraries in X10: user-definable operators and constrained types.

X10's strong support for user-definable operators enables library-based array classes to feel as if they were intrinsic to the language. For example, in the code fragment `a(i,j) = b(i,j,k)` the operator `()` is being used to read the element from `b` and the operator `()=` is being used to store the value in `a`. The implementation of these operators is fully user-provided by writing X10 code in the form of operator definitions in the classes of `a` and `b` respectively.

Array libraries in X10 make extensive use of X10's constrained type system to statically enforce API usage invariants and to encode sophisticated implementation invariants. The most complex usages are in the `x10.regionarray` package (see 3.4), but all of our array libraries use constrained types to enforce that the number of dimensions used in indexing operations (`()` and `()=`) matches the `rank` (dimensionality) of the target array. An essential aspect of the usability of the constrained type system is compiler support for type inference of local variables and method return types and optional generation of dynamic checking code for constraints that are not statically satisfied.

2.3 Useful Implementation Capabilities

We have found three non-standard implementation capabilities of the X10 toolchain quite useful in building high performance array libraries: `@NativeRep` classes, annotations for indicating hot/cold paths, and constant propagation of reads from property fields based on static type information.

The `@NativeRep` mechanism of the X10 toolchain makes it possible to define new intrinsic types without modifying the X10 compiler. In brief, a `@NativeRep` class consists of two parts: an X10 class that specifies the API of the new intrinsic type in the form of a collection of `native` methods and an implementation class written in Java (managed X10) or C++ (native X10) that provides the actual operations. The X10 compiler provides an annotation framework that allows these pieces to be tied together, thus allowing the intrinsic types to be fully defined purely in source code (no compiler modification necessary).

To maximize performance, the array library implementer needs to be able to inform the compiler of hot/cold/exceptional path information for key operations. In X10, this is achieved through the annotation mechanism. In particular, the `Inline` and `NoInline` method annotations can be used to force the inlining of hot paths and maintain the outlining of cold paths. The `NoReturn` annotation indicates that a method will unconditionally throw an exception. By combining `NoInline` and `NoReturn` a cold path method that raises an `ArrayIndexOutOfBounds` exception can be properly recognized by the X10 compiler and this information passed along to the C++ compiler to enable it to make optimal code placement and register allocation decisions.

Finally, X10's constrained types provide a way to use static type information to drive compile-time code specialization. The static type of an expression may contain constraints restricting the possible values of some of the fields of the type. One idiomatic way this is used in the `x10.regionarray` library is to conditionalize code paths by reads of these fields. If the code is then inlined into a calling context where the constraints on the type of the containing object imply that the field is a compile-time constant, then the field read is replaced by a constant value and standard constant propagation and folding applied to simplify the code (for example by removing a branch of an if/then/else).

3. Core Arrays

This section describes the three base array abstractions that are included in the X10 standard library and thus available to all X10 2.4 programs. As much as possible, all three of these abstractions provide a similar look and feel by following a set of API conventions. After describing these commonalities, we will discuss each abstraction in turn covering its design goals and key aspects of its implementation.

3.1 Common APIs

All the core array abstractions provide element indexing operations via operators `()` and `()=`. The multi-dimensional arrays all define a property `rank` that represents the dimensionality of the array instance and the overloaded `()` and `()=` operators have method guards (type constraints) that ensure that indexing operations are always applied with arguments that are consistent with the target array's dimensionality.

To enable programmers to conveniently exploit large memory systems, the size of an array is a `long` (64-bit integer) and all array indexing operations use 64-bit quantities as well. This is especially important for distributed arrays on clustered systems, where the array's data, and thus the global index space, may be spread across thousands of compute nodes. Without 64-bit array sizes and indices, it is hard for a distributed array to effectively exploit the available aggregate cluster memory.

The various array implementations all provide high-level bulk operations such as `map` and `reduce` with implicitly parallel semantics. These operations take a function instance that represents the operation to be performed and applies it exactly once to each element in an unspecified order (thus allowing the implementations to be internally dynamically parallelized).

Finally, arrays all provide iterations over both their values and their index spaces, enabling succinct (and efficient) loop comprehensions to be used for many common idioms.

3.2 x10.lang.Rail: An Intrinsic Local Array

Design Goals The objective of the `Rail` class is to provide a simple, high-performance implementation of indexed storage that can be used as the foundation for building more sophisticated arrays. `Rail` should have minimal metadata space overhead and `Rail` indexing operations must result in optimal generated machine code sequences. To support memory safety, it must be possible to optionally[1] dynamically validate that all `Rail` accesses use valid indices.

To satisfy these goals, we decided that `Rail` should provide a one-dimensional, zero-based, densely-indexed array analogous to the built-in arrays in Java. The `Rail` class is generic in its element type; relying on X10's instantiation based semantics for generics to avoid unnecessary boxing of data elements.

Implementation The `Rail` classes uses the `@NativeRep` mechanism described in Section 2.3 to allow handcrafted implementations to be provided for Native and Managed X10. The X10 language also supports a dedicated syntax for `Rail` literals: `[1,2,3]` is a `Rail` literal of size 3 containing the numbers 1, 2 and 3.

For Native X10, `Rail` is implemented as a C++ generic class, using the standard idiom of a struct whose last member is a size 1 array to allow a single heap allocation to include both the metadata (C++ virtual function pointer and array size) and the data (C++ generic array of elements). As a result, the space overhead for metadata is held to 2 machine words and no extra indirections are needed. Indexing operations are mapped directly to C++ level indexing operations on the backing C++ array. The resulting C++ code is easily understood by the platform C++ compilers and all the standard loop optimizations such as induction variable elimination and vectorization can be applied.

For Managed X10, `Rail` is implemented as a Java class that wraps a backing Java array. To minimize boxing overheads, if the element type of the `Rail` matches one of Java's built-in primitive types (e.g. `double`) then a primitive array (e.g. `double[]`) is used as the backing Java array. This backing array is stored in a field of nominal static type `Object` and therefore must be downcast on every access. To mitigate the dynamic overhead of this design, especially in loops, the X10 compiler includes an optimization pass that caches the backing Java array in a local variable of the more specific type at the outermost scope in which it was accessed. This optimization greatly reduces the dynamic number of downcasts for many programs, but some room for improvement remains in our implementation. For impedance matching between Java's `int` (32-bit) sized arrays and X10's 64-bit array size, we check the requested size in Managed X10's `Rail` constructor and raise an exception if the size is not legal in Java.

3.3 x10.array: Basic Arrays

Design Goals The classes of the `x10.array` package are intended to provide a high-performance implementation of a very important common case of array usage in numeric codes: densely indexed, rectangular, multi-dimensional arrays. The primary performance metric for typical numerical kernels is dynamic machine instructions executed per element access; a successful design will be competitive with the best generated code for equivalent Fortran arrays. Minimizing metadata space overheads, cold-path code space and optimizing element locality are important secondary performance metrics.

The package provides both local and distributed arrays and supporting concepts such as iteration spaces and utilities for blocking iteration spaces. To improve programmer productivity, elements of distributed arrays are accessed using a global index space, which is internally converted into an offset into the local storage in the place of access. By default, all indexing operations include bounds checks (and place checks for distributed arrays); however these checks can be explicitly disabled with a compile-time flag.

Implementation All classes in this package are implemented as normal X10 source code (no usage of `@NativeRep`). The backing storage for the data is provided by `Rail`.

To achieve the performance goals outlined above, the entire hot path of the indexing operation must be fully inlined and contain no function calls that return normally. Our experience has been that even a single non-exceptional function call in the hot path of the array indexing operation will result in an unacceptable loss of performance for our intended use cases. Therefore we have adopted a design pattern for both local and distributed arrays where an abstract class, `Array` and `DistArray` respectively, provides bulk operations that can operate purely in terms of the backing `Rail` and a collection of specialized final subclasses (e.g. `Array_2` for a two-dimensional, zero-indexed, row major array) provide the indexing operations for each dimensionality and distribution. Application code refers to the type of the specialized subclass, enabling full optimization of all operations.[2] Having specialized subclasses also naturally minimizes space overheads because each subclass can define exactly the instance fields it needs to best encode its metadata. A typical array instance in this package is a small header object containing metadata and a pointer to a single backing `Rail` object that contains the densely stored data.

Although this approach results in a proliferation of specialized classes, each class is small and straightforward to implement. The entire package currently only contains 900 source lines of code; SLOC counts for selected classes are shown in table 1.

Table 1: Source Lines of Code for selected X10 array classes.

Class	SLOC Count
Array	65
Array_2	87
DistArray	98
DistArray_BlockBlock_2	129

We expect the lines of code will increase as additional specializations of `Array` and `DistArray` are implemented in this package and as external application-specific arrays. However due to the simpler APIs we expect the total lines of code to remain well below that of the `x10.regionarray` package.

3.4 x10.regionarray: Flexible Arrays

Design Goals The `x10.regionarray` package is the most ambitious of the array frameworks provided by the X10 standard library. It is descended from the early X10 array sub-languages and supports an extremely flexible and extensible set of abstractions. A region is a set of k-dimensional points. Supported regions vary from simple dense rectangular spaces to complex polyhedral spaces constructed via a rich region algebra. Local arrays are defined as a function from points in a region to data values. Multi-place arrays are defined via an additional level of mapping: a distribution that maps each point in a region to a `Place`. Any region can be combined with any distribution to define a distributed array, resulting in a highly flexible data distribution mechanism.

[1] X10 allows checks to be unsafely eliminated via compiler flags.

[2] The combination of local type inference and X10's type definition mechanism can greatly reduce the number of explicit references to the specialized subclasses, thus making it less burdensome to switch array implementation subclasses as the program evolves.

Core Implementation A standard object-oriented design was used to implement these abstractions. Abstract `Region` and `Dist` classes define a rich set of API methods and provide default implementations of many functions. All region and distribution instances are created via factory methods on `Region` and `Dist`, allowing the various implementation level subclasses to be freely changed and specialized without requiring any changes in client code. The `Array` and `DistArray` class are parameterized by `Region` and `Dist` instances and internally most operations are implemented via calls on the `Region` and `Dist` APIs. Additional subclasses of `Region` and `Dist` can be defined externally to the `x10.regionarray` package and used to create `Array` and `DistArray` instances.

Throughout the API and the implementation, extensive use was made of X10's constrained types to express usage invariants and to allow optimized indexing methods specialized to specific ranks (dimensionality) to be properly applied. Internally, constrained types were also used to specialize code paths for common cases such as dense rectangular regions, which enabled more efficient index calculation.

In total, the package contains 3,340 lines of code with approximately 500 lines in `Array`, 300 lines in `DistArray`, 1000 lines in the implementation of distributions and 1540 in the implementation of regions (due to the complexity of general polyhedral regions).

Extensibility Our experience is that the design did prove to be fairly extensible and flexible. Many usage scenarios involving subregions, stencils, and sophisticated distributions could be succinctly expressed using the provided APIs and classes. When necessary, the package could also be extended externally. For example early in the development of ANUChem [11] an application-specific region was used in the Fast Multipole Method to define `Arrays` over the unusually-shaped regions of multipole expansions.

Discussion Although expressive and flexible, and despite significant investment in development and optimization, the `x10.regionarray` framework was never actually usable in performance-sensitive code. The design suffered from several weaknesses that ultimately led to the decision to provide the `x10.array` package as a higher performance alternative for many common cases.

The fundamental issue is that high-performance array operations absolutely require that the indexing calculations be performed by simple inlined code sequences that can be effectively understood and optimized by the platform compilers. Careful design and compiler optimizations based on exploiting constrained types did enable this to be achieved in certain restricted scenarios. However, these optimizations did not apply to the more general scenarios (non-rectangular regions) and even with local type inference and aggressive method inlining were vulnerable to loss of static type information across method boundaries due to the complexity of specifying the necessary constraints. Additional work on interprocedural optimization and whole program specialization could expand the set of scenarios where acceptable performance could be achieved. But such whole program optimization is not easily compatible with the usage of X10 to build libraries and frameworks or with the Java-based Managed X10 toolchain. An important secondary issue was the higher space overheads of more complex metadata objects and of caching derived values to optimize indexing operations.

4. Global Matrix Library

4.1 Design Goals

The Global Matrix Library (GML) was developed to support distributed linear algebra in X10. It provides a variety of single-place and distributed matrix formats for dense and sparse matrices, and implements linear algebra operations for these formats. High-level operations operate on entire matrices and are intended to support a programmer writing in a sequential style while fully exploiting available parallelism. Parallelism within a node is exploited through the use of multi-threaded implementations of the Basic Linear Algebra Subroutines (BLAS) [3] and Linear Algebra PACKage (LAPACK) [2] routines. Between nodes, collective communications are used to support coarse-grained parallelism.

Table 2: Matrix classes in X10 Global Matrix Library.

	Storage	
Locality	**Dense**	**Sparse**
Local	`DenseMatrix`	`SparseCSC/SparseCSR`
Duplicated	`DupDenseMatrix`	`DupSparseMatrix`
	`DupBlockMatrix`	
Distributed	`DistDenseMatrix`	`DistSparseMatrix`
	`DistBlockMatrix`	

As well as being used directly within application code, another important design goal for GML is to serve as a compilation target for high-level array languages, to enable highly-parallel execution for matrix/vector operations on large data sets. For example the SystemML approach compiles machine learning algorithms written in the high-level Declarative Machine Learning language (DML) to produce MapReduce jobs for execution in Hadoop [6]; in a similar fashion, DML could compile to X10 code with GML operations for clustered execution.

4.2 Implementation

All matrix and vector representations implemented in GML derive from the base classes `x10.matrix.Matrix` and `x10.matrix.Vector`. Table 2 shows a selection of matrix classes that inherit from `x10.matrix.Matrix`. Additional local representations are also implemented for triangular and symmetric matrices (not shown). Matrices support a set of standard operations (for example element-wise addition and multiplication, matrix multiplication, norm, trace), which are polymorphic with regard to representation.

Operations for single-place dense matrices are implemented as `@NativeRep` wrappers around the BLAS and LAPACK routines; more complex operations are implemented in X10 using the simple operations as building blocks.

4.3 Usage of GML

The sequential style of GML is demonstrated through two example linear algebra kernels to compute Non-negative Matrix Factorization and PageRank.

Non-negative Matrix Factorization (NMF) [6, 9] is used in fields such as topic modeling and computer vision to infer structure within a large matrix $\mathbf{V}$ by finding an approximate factorization $\mathbf{V} \approx \mathbf{WH}$ such that the dimensions of the factor matrices are significantly smaller than $\mathbf{V}$. Figure 1 shows pseudocode and the corresponding X10 code for NMF implemented using GML. Matrices `V` and `W` are block-distributed across all places, while matrix `H` is duplicated across all places.

Figure 2 shows weak scaling of NMF on 1 to 384 cores of a Power 775 system. One X10 place was run per core, increasing number of rows in the $\mathbf{V}$ matrix to maintain 312500 non-zeros per core. The IBM ESSL library was used for the BLAS operations.

The PageRank algorithm [12] operates on a transition matrix $\mathbf{G}$ representing the structure of a network of linked documents, to find the dominant eigenvector $\mathbf{p}$ as a measure of the centrality or importance of each document. Figure 3 shows pseudocode and corresponding X10 code for an implementation of PageRank using

```
for $(1 \ldots n)$ do
    $\mathbf{H} = \mathbf{H} \cdot (\mathbf{W}^T\mathbf{V} \;/\; \mathbf{W}^T\mathbf{W}\mathbf{H})$
    $\mathbf{W} = \mathbf{W} \cdot (\mathbf{V}\mathbf{H}^T \;/\; \mathbf{W}\mathbf{H}\mathbf{H}^T)$
end for
```

```
for (1..n) {
  // H
  WtV.transMult(W, V);
  WtW.transMult(W, W);
  WtWH.mult(WtW, H);
  WtV.cellDiv(WtWH);
  H.cellMult(WtV);
  // W
  VHt.multTrans(V, H);
  HHt.multTrans(H, H);
  WHHt.mult(W, HHt);
  VHt.cellDiv(WHHt);
  W.cellMult(VHt);
}
```

Figure 1: Non-negative Matrix Factorization: pseudocode and X10 code using GML

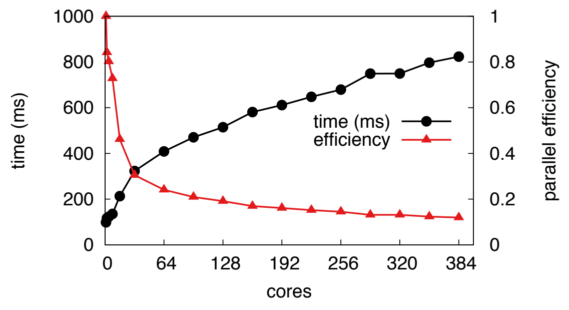

Figure 2: X10 NMF performance on Power 775

GML. The matrices `G` and `Gp` are distributed over all places, while the vector `p` is duplicated across all places.

In ANUChem [1], single-place and distributed dense matrices and linear algebra operations are used to implement a Hartree–Fock self-consistent field calculation using the Resolution of the Coulomb Operator [10]. Key matrices such as the Fock matrix are block distributed using the `DistDenseMatrix` class, and BLAS operations are used on local blocks to transform auxiliary integrals into Fock matrix contributions.

```
for $(k = 1 \ldots n)$ do
    $\mathbf{p} = \alpha\mathbf{G}\mathbf{p} + (1-\alpha)\mathbf{E}\mathbf{U}^T\mathbf{p}$
    if $(\|\mathbf{p}^{(k)} - \mathbf{p}^{(k-1)}\|_1 <$ tolerance$)$ break
end for
```

```
for (1..n) {
  Gp.mult(G, p).scale(alpha);
  Gp.copyTo(vGp); // dist->local dense
  vP.mult(E, U.dotProd(vP))
    .scale(1-alpha).cellAdd(vGP);
  p.sync();       // broadcast
  val delta = vP.l1Norm(prevVP);
  if (delta < tolerance) break;
  vP.copyTo(prevVP);
}
```

Figure 3: PageRank: pseudocode and X10 code using GML

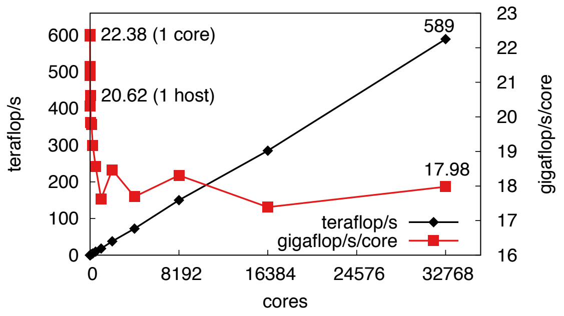

Figure 4: X10 HPL performance on Power 775

5. Application-specific Arrays

A goal of the X10 approach is to enable application-specific array abstractions to easily be built. In this section we briefly describe three successful examples of such work.

5.1 Global HPL

HPL is a standard HPC kernel benchmark which computes the LU factorization of a large square matrix. The X10 implementation of this benchmark features a two-dimensional block-cyclic data distribution, a right-looking variant of the LU factorization with row-partial pivoting, and a recursive panel factorization. The heart of this code is an application-specific `BlockedArray` class which in only 129 lines of code implements a distributed, block-block cyclic matrix using `Rail` as the backing data store in each `Place`. Figure 4 shows the performance of HPL on a 32,768 core Power 775 system; more details can be found in [17].

5.2 Random Access

The Global RandomAccess benchmark measures the system's ability to update random memory locations in a table distributed across the system, by performing XOR operations at the chosen locations with random values. In X10, this table is implemented as a small (78 lines of code) application-specific `DistRail` class. `DistRail` utilizes customized memory allocation to ensure that the backing `Rail` is allocated using large pages and at the same virtual address in every `Place` and augments operators `()` and `()=` with an additional remote atomic XOR operation that exploits underlying hardware acceleration on systems that provide it.

5.3 MiX10

MiX10 [8] is an open MATLAB compiler that translates MATLAB programs to X10. After analysis and optimization, MiX10 is able to optimize many MATLAB vectors and matrices into X10 arrays. As reported in more detail in [7], MiX10 both targets X10's existing `x10.regionarray` and `x10.array` classes and has defined additional column-major, one-indexed subclasses of `x10.array.Array` that are better suited for MATLAB's linear indexing. MiX10 demonstrates that X10 provides a suitable compilation target for high-level array based program languages and opens up a number of interesting avenues for potential future research.

6. Related Work

A comprehensive survey of the rich history of array sub-languages is beyond the scope of this paper. Therefore we just comment on a few closely related efforts. The design of `x10.array` was driven by an analysis of the high-performance arrays of the Fortran and C/C++ languages and the capabilities of platform compilers to optimize operations on them. PGAS languages such as UPC and Coarray Fortran extend their base language with built-in distributed arrays. This approach differs from ours in that the design choices for the distributed arrays are codified as part of the language specification and not easily extensible or modifiable by users. Chapel's user-defined domain maps represent an ambitious attempt to combine compiler-supported and optimized global view distributed arrays with high degrees of user extensibility [4]. Just like the `x10.regionarray` package, we believe it is still an open research question if such a flexible approach can achieve acceptable levels of performance for general usage in numeric codes.

7. Future Directions

The goal of this paper has been to describe the current state of array-based programming in X10, to assess what has worked well and what could be done better, and to lay out avenues for future work.

Based on our experience thus far, we believe the fundamental approach taken in X10 of supporting array-based programming through general language and toolchain capabilities for building reusable software frameworks is sound. Arrays in X10 certainly feel like an integral part of the language, but yet can be easily evolved since they are primarily defined as vanilla X10 classes using normal language mechanisms. Furthermore, the investment in building these language and toolchain capabilities is applicable for building many other libraries and frameworks.

We certainly see opportunities to explore more advanced constraint solving technology to enable compile time bounds checking and the expression of data structure invariants involving arithmetic and numeric inequalities. The development of library-level interprocedural optimization and link-time specialization in our toolchain could improve the performance of the `x10.regionarray` package sufficiently to make it usable in more scenarios. The implementation of `Rail` (and other generic container types) in Managed X10 could also be improved to reduce boxing and other object-model overheads on boundary crossings between generic and non-generic code.

Finally, the Global Matrix Library presents a number of opportunities. We would like to extend GML with additional operations, apply it to more applications, and study its scalability and performance on large scale systems. Efforts like MiX10 are also intriguing areas for future collaboration with GML: one of the original design goals for the GML was to serve as a compilation target for a high-level array programming language.

Acknowledgments

This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002 and by the Department of Energy under award DEFOA-0000619.

A number of members of the X10 team and user community have contributed both ideas and code to the material presented in this paper. We especially want to thank Bruce Lucas, Nate Nystrom, Igor Peshansky, Vijay Saraswat, and Juemin Zhang.

All source code lines statistics were generated using David A. Wheeler's 'SLOCCount'.

References

[1] ANUChem. `http://cs.anu.edu.au/~Josh.Milthorpe/anuchem.html`. accessed: 10 April 2014.

[2] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and S. Ostrouchov. LAPACK users' guide, release 2.0. Technical report, SIAM, Philadelphia, 1995.

[3] L. Blackford, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, M. Heroux, L. Kaufman, A. Lumsdaine, A. Petitet, R. Pozo, and Remington. An updated set of basic linear algebra subprograms (BLAS). *ACM Transactions on Mathematical Software*, 28(2):135–151, 2002.

[4] B. L. Chamberlain, S.-E. Choi, S. J. Deitz, D. Iten, and V. Litvinov. Authoring user-defined domain maps in Chapel. Chapel Users Group, May 2011.

[5] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioğlu, C. von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '05, pages 519–538, New York, NY, USA, 2005. ACM.

[6] A. Ghoting, R. Krishnamurthy, E. Pednault, B. Reinwald, V. Sindhwani, S. Tatikonda, Y. Tian, and S. Vaithyanathan. SystemML: declarative machine learning on MapReduce. In *2011 IEEE 27th International Conference on Data Engineering (ICDE)*, pages 231–242, Apr. 2011.

[7] V. Kumar. MiX10: Compiling MATLAB to X10 for high performance. Master's thesis, McGill University, April 2014.

[8] V. Kumar and L. Hendren. First steps to compiling MATLAB to X10. In *Proceedings of the Third ACM SIGPLAN X10 Workshop*, X10 '13, pages 2–11, New York, NY, USA, 2013. ACM.

[9] D. D. Lee and H. S. Seung. Algorithms for non-negative matrix factorization. In T. K. Leen, T. G. Dietterich, and V. Tresp, editors, *Advances in Neural Information Processing Systems 13*, pages 556–562. MIT Press, 2001.

[10] T. Limpanuparb, J. Milthorpe, A. Rendell, and P. Gill. Resolutions of the Coulomb operator: VII. Evaluation of long-range Coulomb and exchange matrices. *Journal of Chemical Theory and Computation*, 9(2):863–867, 2013.

[11] J. Milthorpe, V. Ganesh, A. P. Rendell, and D. Grove. X10 as a parallel language for scientific computation: practice and experience. In *Proceedings of the 25th IEEE International Parallel and Distributed Processing Symposium*, IPDPS '11, pages 1080–1088. IEEE Computer Society, May 2011.

[12] L. Page, S. Brin, R. Motwani, and T. Winograd. The PageRank citation ranking: Bringing order to the Web. Technical Report SIDL-WP-1999-0120, Stanford University, Nov. 1999.

[13] V. Saraswat, G. Almasi, G. Bikshandi, C. Cascaval, D. Cunningham, D. Grove, S. Kodali, I. Peshansky, and O. Tardieu. The Asynchronous Partitioned Global Address Space Model. In *AMP'10: Proceedings of The First Workshop on Advances in Message Passing*, June 2010.

[14] V. Saraswat, B. Bloom, I. Peshansky, O. Tardieu, and D. Grove. The X10 language specification, v2.4. Aug. 2013.

[15] V. Saraswat and R. Jagadeesan. Concurrent clustered programming. In *Concur'05*, pages 353–367, 2005.

[16] V. Saraswat, O. Tardieu, D. Grove, D. Cunningham, M. Takeuchi, and B. Herta. A brief introduction to X10 (for the high performance programmer). `http://x10.sourceforge.net/documentation/intro/latest/html/`, Feb. 2013.

[17] O. Tardieu, B. Herta, D. Cunningham, D. Grove, P. Kambadur, V. Saraswat, A. Shinnar, M. Takeuchi, and M. Vaziri. X10 and APGAS at petascale. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14, pages 53–66, New York, NY, USA, 2014. ACM.