McGill University
School of Computer Science
Sable Research Group

Compiling MATLAB for High Performance Computing via X10[1]

Sable Technical Report No. sable-2013-03

Vineet Kumar and Laurie Hendren

October 12, 2013

Contents

List of Figures

Abstract

Matlab is a popular dynamic array-based language commonly used by students, scientists and engineers, who appreciate the interactive development style, the rich set of array operators, the extensive builtin library, and the fact that they do not have to declare static types. Even though these users like to program in Matlab, their computations are often very compute-intensive and are better suited for emerging high performance computing systems. Our solution is MiX10, a source-to-source compiler that automatically translates Matlab programs to X10, a language designed for "Performance and Productivity at Scale"; thus, helping scientific programmers make better use of high performance computing systems.

This paper addresses two major challenges in compiling Matlab to X10 for high performance computing: (1) efficiently transforming dynamically-typed Matlab arrays to the best high-level, statically-typed array representation in X10; and (2) effectively exposing concurrency in Matlab and generating efficient concurrent code in X10. We have implemented the techniques presented in this paper and provide an empirical study on a set of benchmarks, examining both the efficiency of the generated sequential X10 code and speedups for the concurrent versions.

1 Introduction

Matlab is a popular numeric programming language, used by millions of scientists, engineers as well as students worldwide [17]. Matlab programmers appreciate the high-level matrix operators, the fact that variables and types do not need to be declared, the large number of library and builtin functions available, and the interactive style of program development available through the IDE and the interpreter-style read-eval-print loop. However, even though Matlab programmers appreciate all of the features that enable rapid prototyping, their applications are often quite compute intensive and time consuming. These applications could perform much more efficiently if they could be easily ported to a high performance computing system.

On one hand, all the aforementioned characteristics of Matlab make it a very user-friendly and thus a popular application to develop software among a non-programmer community. On the other hand, these same characteristics, together with a lack of a formal language specification, unconventional semantics, and the fact that it closed source, make it challenging to develop a static Matlab compiler. Furthermore, the use of arrays as default data type and the dynamicity of the base types and shapes of arrays make it even harder to add support for concurrency in a static Matlab compiler.

The de facto standard, the Mathworks' implementation of Matlab, is essentially an interpreter with a *JIT accelerator* [22], which is generally slower than highly-optimized static languages. Mathworks' proprietary solution for concurrency is the *Parallel Computing Toolbox* [16], which allows users to use multicore processors, GPUs and clusters. However, this toolbox uses heavyweight worker threads and has limited scalability.

Our aim is to provide Matlab's ease of use, to benefit from the advantages of static compilation, and to expose scalable concurrency. Our solution is MiX10, an open source-to-source compiler that statically translates Matlab programs to X10. X10 is an object-oriented and statically-typed language which uses cilk-style arrays indexed by *Point* objects and rail-backed multidimensional arrays, and has been designed with well-defined semantics and high performance computing in mind [8]. The X10 compiler can generate C++ or Java code and supports various communication interfaces including sockets and MPI for communication between nodes on a parallel computing system.

We have concentrated both on providing an efficient translation for the sequential core of X10, as well as providing an effective bridge to the concurrency features of X10. One key way in which we interface with concurrency in Matlab is by designing and implementing a translation of the Matlab `parfor` construct to X10. We also introduced concurrency constructs in Matlab analogous to those provided in X10, thus allowing users to further specify fine-grained concurrency in their programs.

The overall goal of the MiX10 project it to allow scientists and engineers to write programs in Matlab (or use existing programs already written in Matlab), and at the same time enjoy the benefits of high performance computing via the X10 system without having to learn a new and unfamiliar language. Also, since the X10 compiler has back-ends that can produce both C++ and Java, MiX10 can also be used by systems that use Matlab for prototyping and C++ or Java for production.

The major contributions of this paper are as follows:

Identifying key challenges: We have identified the key challenges in compiling Matlab to X10.

Techniques for efficiently compiling Matlab arrays: Arrays are the core of Matlab. All data, including scalar values are represented as arrays in Matlab. Efficient compilation of arrays is the key for good performance. We provide techniques to compile Matlab arrays to two different representations of arrays provided by X10.

Comparison of the two array representations: X10 provides two types of array representations for multidimensional arrays: (1) Cilk-styled, region-based arrays and (2) rail-backed arrays. We compare and contrast these two array forms for a high performance computing language in context of being used as a target language.

Code generation strategies for `parfor` and vectorized instructions: `parfor` allows parallel execution of for loop iterations in Matlab. We provide technique to effectively compile `parfor` construct to X10. We also discuss our strategy to handle vectorized instructions in a concurrent fashion.

Introduction of fine-grained concurrency constructs in Matlab: We have introduced X10 like concurrency constructs in Matlab, allowing Matlab programmers to expose fine-grained concurrency in their programs.

Open implementation and empirical evaluation: We have implemented the techniques provided in this paper in an open and extensible framework (`www.sable.mcgill.ca/mclab/mix10.html`), and have experimented with it on a set of benchmarks. Our initial results show considerable improvements when customized and optimized X10 array representations are used, and that X10 parallel performance significantly outperforms Matlab on our benchmarks.

The remainder of this paper is structured as follows. In Section 2 we describe the background, and how the entire project is structured. Section 3 gives an introduction to arrays in X10. In Section 4 we provide our compilation strategies for different array representations and a comparison of both the approaches. In Section 5 we describe our strategies for handling `parfor` and vectorized instructions, we describe how we introduced fine-grained concurrency constructs in MATLAB, and we provide an empircal evaluation of our approach. Finally, we provide a discussion of related work in Section 6, and conclude and discuss some planned future work in Section 7.

2 Background

MIX10 is implemented on top of several existing MATLAB compiler tools. The overall structure is given in Figure 1, where the new parts are indicated by the shaded boxes, and future work is indicated by dashed boxes.

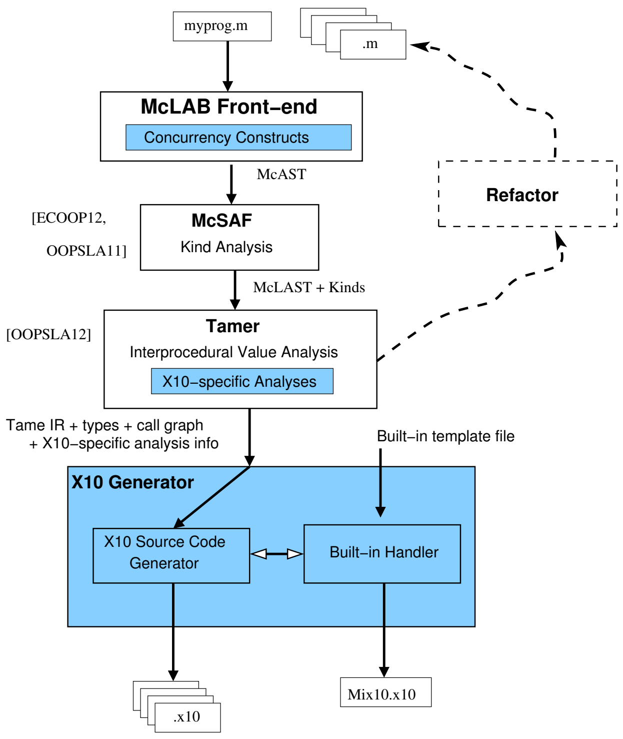


Figure 1: Overview of MIX10 structure

As illustrated at the top of the figure, a MATLAB programmer only needs to provide an entry-point MATLAB function (called `myprog.m` in this example), plus a collection of other MATLAB functions and libraries (directories of functions) which may be called, directly or indirectly, by the entry point. The programmer may also specify the types and/or shapes of the input parameters to the entry-point function. As shown at the bottom of the figure, our MIX10 compiler automatically produces a collection of X10 output files which contain the generated X10 code for all reachable MATLAB functions, plus one X10 file called `mix10.x10` which contains generated and specialized X10 code for the required builtin MATLAB functions. Thus, from the MATLAB programmer's point of view, the MIX10 compiler is quite simple to use.

MATLAB is actually quite a complicated language to compile, starting with its rather unusual syntax, which cannot be parsed with standard LALR techniques. There are several issues that must be dealt with including distinguishing places where white space and new line characters have syntactic meaning, and filling in missing **end** keywords, which are sometimes optional. The McLab front-end handles the parsing of Matlab through a two step process. There is a pre-processing step which translates Matlab programs to a cleaner subset, called *Natlab*, which has a grammar that can be expressed cleanly for a LALR parser. The McLab front-end delivers a high-level AST based on this cleaner grammar. For the MiX10 project we have extended the front-end to handle X10- inspired concurrency constructs, exposed in the Matlab program as extended comments.

After parsing, the next major phase of MiX10 uses the McSaf framework [3, 4] to disambiguate identifiers using *kind analysis* [5], which determines if an identifier refers to a *variable* or a *named function*. This is required because the syntax of Matlab does not distinguish between variables and functions. For example, the expression a(i) could refer to four different computations, a could be an array or a function, and i could refer to the builtin function for the imaginary value i, or it could refer to a variable i. The McSaf framework also simplifies the AST, producing a lower-level AST which is more amenable to subsequent analysis.

The next major phase is the Tamer [6], which is a key component for any tool which statically compiles Matlab. The Tamer generates an even more defined AST called *Tamer IR*, as well as performing key interprocedural analyses to determine both the call graph and an estimate of the base type and shape of each variable, at each program point. The call graph is needed to determine which files (functions) need to be compiled, and the type and shape information is very important for generating reasonable code when the target language is statically typed, as is the case for X10.

The Tamer may find dynamic Matlab features which cannot be statically compiled, in which case it flags that feature as not tame, and the ultimate goal is to support a refactoring tool which would aid the programmer to restructure their input Matlab program in order to eliminate the wild feature.

The Tamer also provides an extensible *interprocedural value analysis* and an interprocedural analysis framework that extends the intraprocedural framework provided by McSaf. Any static backend will use the standard results of the Tamer, but is also likely to implement some target-language-specific analyses which estimate properties useful for generating code in a specific target language. We have currently added an analysis for determining if a Matlab variable is *real* or *complex*.

For the purposes of MiX10, the output of the Tamer is a low-level, well-structured AST, which along with key analysis information about the call graph, the types and shapes of variables, and X10-specific information. These Tamer outputs are provided to the code generator, which generates X10 code, and which is the main focus of this paper. As shown in the X10 generator box in Figure 1, the X10 source code generator actually gets inputs from two places. It uses the Tamer IR it receives from the the Tamer to drive the code generation, but for expressions referring to built-in Matlab functions it interacts with the *Built-in Handler* which used the built-in template file we provide. We have described the functioning of the built-in handler and the basic code generation strategy for ordinary sequential constructs at the X10 workshop [12], where we also pointed out the challenges of generating efficient code for X10 arrays. In this paper we provide an overview of the whole X10 compiler, but most importantly we focus on the key challenges of generating efficient array code and effectively utilizing the X10 concurrency features.

3 Introduction to X10 arrays

In order to understand the challenges of translating MATLAB to X10, one must understand the different flavours and functionality of X10 arrays.

At the lowest level of abstraction, X10 provides an intrinsic one-dimensional fixed-size array called `Rail` which is indexed by a `Long` type value starting at `0`. This is the X10 counterpart of built-in arrays in languages like C or Java. In addition, X10 provides two types of more sophisticated array abstractions in packages, `x10.array` and `x10.regionarray`.

Rail-backed Simple arrays are a high-performance abstraction for multidimensional arrays in X10 that support only rectangular dense arrays with zero-based indexing. Also, they support only up to three dimensions (specified statically) and row-major ordering. These restrictions allow effective optimizations on indexing operations on the underlying `Rail`. Essentially, these multidimensional arrays map to a `Rail` of size equal to number of elements in the array, in a row-major order.

Region arrays are much more flexible. A *region* is a set of points of the same rank, where `Points` are the indexing units for arrays. Points are represented as n-dimensional tuples of integer values. The `rank` of a point defines the dimensionality of the array it indexes. The rank of a region is the rank of its underlying points. Regions provide flexibility of shape and indexing. *Region arrays* are just a set of elements with each element mapped to a unique point in the underlying region. The dynamicity of these arrays come at the cost of performance.

Both types of arrays also support distribution across places. A *place* is one of the central innovations in X10, which permits the programmer to deal with notions of locality.

4 Compilation to X10 arrays

Arrays are the core of the MATLAB programming language. Every value in MATLAB is a *Matrix* and has an associated array shape. Even scalar values are represented as 1×1 arrays. Most of the data read and write operations involve accessing individual or a set of array elements. Given the central role of arrays in MATLAB, it is of utmost importance for our MiX10 compiler to find effective and efficient translations to X10 arrays.

Our strategy is use the more efficient rail-backed simple arrays whenever possible, and to fall back to the more flexible, but less efficient region-based arrays when necessary. In addition, we have made a custom version of the rail-backed array implementation which maps well to MATLAB arrays, thus giving even better performance.

MiX10 uses the shape analysis engine built on top of McSAF analysis framework [3,4] and Tamer [6], to statically estimate the shapes of involved arrays. When the shapes can be determined accurately, the rail-backed simple arrays are used. When it is not possible to statically determine the accurate shape of the involved matrix, and when arrays of have more than three dimensions, MiX10 falls back on region arrays which are flexible enough to support such dynamic array operations. MiX10 users can also explicitly force our compiler to always use the region arrays, which allows them to experiment with different kinds of generated code.

4.1 Techniques for compiling to Simple Arrays

In dealing with the simple rail-backed arrays, there were two important challenges. First, we needed to determine when it is safe to use the simple rail-backed arrays, and second, we needed an implementation of simple rail-backed arrays that handles the column-major, 1-indexing, and linearization operations required by MATLAB.

When to use simple rail-backed arrays: After the shape analysis of the source MATLAB program, if shapes of all the arrays in the program: (1) are known statically, (2) remain same at all points in the program and (3) are supported by the X10 implementation of simple arrays; then MIX10 generates X10 code that uses simple arrays.

Enhancements to the X10 implementation of simple arrays: In order to make X10 simple arrays more compatible with MATLAB we modified the implementation of the `Array_2` and `Array_3` classes in `x10.array` package to use column-major ordering instead of the default row-major ordering when linearizing multidimensional arrays to the backing `Rail` storage. MATLAB.[2] MATLAB uses column-major ordering to linearize arrays. This modification also makes it trivial to support linear indexing operations in MATLAB.[3] We also added methods to get the sub-array of an array, given the range of indices for the sub-array in each dimension of the array. These methods are especially useful for supporting the `colon` operator in MATLAB.[4]

Given that we can determine when it is safe to use the simple rail-backed arrays, and our improved X10 implementation of them, we then designed the appropriate translations from MATLAB to X10, for array construction, array accesses for both individual elements and ranges.

Array construction: Given the number of dimensions and the size of each dimension, it is easy to construct a simple array. For example a two-dimensional array `A` of type `T` and shape $m \times n$ can be constructed using a statement like `val A:Array_2[T] = new Array_2[T](m,n);`. Additional arguments can be passed to the constructor to initialize the array.

Array access: In MATLAB, an individual array element can be accessed using one or more integral subscripts and a set of elements can be accessed using an expression involving `colon` operator instead of an integer subscript. Also, note that MATLAB uses one-based indexing, whereas X10 uses zero-based indexing; thus all array access indices need to be offset by one.

MATLAB naturally supports linear indexing for individual element access. More precisely, if the number of subscripts in an array access is less than the number of dimensions of the array, the last subscript is linearly indexed over the remaining number of dimensions in a column-major fashion. Our modification to use column-major ordering for the backing `Rail` make it easier and more efficient to support linear indexing by allowing direct access to the underlying `Rail` at the calculated linear offset.

MATLAB allows the use of an expression such as `a:b` (or `colon(a,b)`) to create a vector of integers `[a, a+1, a+2, ... b]`. In another form, an expression like `a:i:b` can be used to specify an integer

[2] `http://www.sable.mcgill.ca/mclab/mix10/x10_update/`

[3] `http://www.mathworks.com/help/matlab/math/matrix-indexing.html`

[4] `http://www.mathworks.com/help/matlab/ref/colon.html.`

interval of size `i` between the elements of the resulting vector. Use of a `colon` expression for array subscripting takes all the elements of the array for which the subscript in a particular dimension is in the vector created by the `colon` expression in that dimension.[5] Consider the following MATLAB code:

```
function [x] = crazyArray(a)
    y = ones(3,4,5);
    x = y(1,2:3,:) ;
end
```

Here `y` is a three-dimensional array of shape $3 \times 4 \times 5$ and `x` is a sub-array of `y` of shape $1 \times 2 \times 5$. Such array accesses can be handled by simply calling the `getSubArray[T]()` that we introduced in the X10 runtime library. The generated X10 code for this example is as follows:

```
static def crazyArray (a: Double){
   val y: Array_3[Double] = new Array_3[Double](Mix10.ones(3, 4, 5));
   val mc_t0: Array_1[Double] = new Array_1[Double](Mix10.colon(2, 3));
   var x: Array_3[Double];
   x = new Array_3[Double](Helper.getSubArray(1, 1, mc_t0(0), mc_t0(1), 1, 5, y)) ;
   return x;
}
```

4.2 Techniques for compiling to Region Arrays

With MATLAB's dynamic nature and unconventional semantics, it is not always possible to statically determine the shape of an arrays accurately. Luckily, with some thought to a proper translation, X10's region arrays are flexible enough to support MATLAB's "wild" arrays. Also, since `Point` objects can be a set of arbitrary integers, there is no restriction on the starting index of the arrays. Region arrays can easily use one-based indexing. MATLAB allows the use of keyword `end` or an expression involving `end` (like `end-1`) as a subscript. `end` denotes the highest index in that dimension. If the highest index is not known the `numElems_i` property of the simple arrays is used to get the number of elements in the `i`th dimension of the array.

Array construction: Array construction for region arrays involves creating a region over a set of points (or index objects) and assigning it to an array. Regions of arbitrary ranks can be created dynamically. For example, consider the following MATLAB code snippet:

```
function[x] = foo(a)
    t = bar(a);
    x = t;
    ...
end
```

```
function[y] = bar(a)
    if  (a == 3)
        y = zeros(a,a+1,a+2,a+3);
    else
        y = zeros(a,a+1,a+2);
    end
end
```

In this code, the number of dimensions of array `t` and hence array `x` cannot be determined statically at compile-time. In such case, it is not possible to generate X10 code that uses simple arrays, however, it can still be compiled to the following X10 code for function `foo()`.

[5] Use of ':' in place of an index without lower and upper bounds indicates the use of all the indices in that dimension.

```
static def foo(a: Double){
  val t: Array[Double] =
    new Array[Double](bar(a));
  val x: Array[Double] =
    new Array[Double](t);
  ...
  return x;
}
```

```
static def bar(a:Double){
  var y:Array[Double]=null;
  if (a == 3) {
    y = new Array[Double]
      (Mix10.zeros(a,a+1,a+2,a+3));
  }
  else {
    y = new Array[Double]
      (Mix10.zeros(a,a+1,a+2));
  }
  return y;
}
```

In this generated X10 code, `t` is an array of type `Double` which can be created by copying from another array returned by `bar(a)` without knowing the shape of the returned array.

Array access: Subscripting operations to access individual elements are mapped to X10's region array subscripting operation. If the rank of array is 4 or less, it is subscripted directly by integers corresponding to subscripts in MATLAB otherwise we create a `Point` object from these integer values and use it to subscript the array. In case an expression involving `end` is used for indexing and the complete shape information is not available, method `max(Int i)`, provided by the `Region` class is used, allowing to determine the highest index for a particular dimension at runtime.

Mapping an array access for a set of elements indexed by an expression involving colon operator in a completely dynamic manner is more complex for region arrays than it is for simple arrays. The X10 code below shows how the previously shown `crazyArray()` method is mapped for using region arrays without any statically known shape information.

```
public static def crazyArray(a: Double){
    val y: Array[Double] = new Array[Double]( Mix10.ones(3,4,5));
    val mc_t0: Array[Double] = new Array[Double]( Mix10.colon(2,3));
    val x: Array[Double];
    val mix10_pt_y: Point;
    mix10_pt_y = Point.make(1−(1L), 1−(mc_t0(1,mc_t0.region.min(1)) as Int), 0);
    x = new Array[Double](
      (Region.make((1..1)) ∗
      (Region.make((mc_t0.region.min(1)) as Int .. (mc_t0.region.max(1)) as Int)) ∗
      (Region.make((y.region.min(2)) .. y.region.max(2))),
      (p:Point(3))=>y(p.operator−(mix10_pt_y))
   );
}
```

If the shape of an array involving `colon` operator is not known, the `Region.min(Int i)` and `Region.max(Int i)` methods are used to compute the correct values at run time. In the above code, a new `Point` object is created that serves as an offset to get the elements at the correct position of the array accessed. Then the new array with region derived from the resultant vector from the `colon` operator for second dimension and from the third dimension of the source array `y` is created. Thus the resultant array `x` has the region `1..1*1..2*1..5`.

Rank specialization: Although region arrays can be used with minimal compile-time information, providing additional static information can improve performance of the resultant code by eliminating run-time checks involving provided information. One of the key specializations that we introduced with use of region arrays is to specify the rank of an array in its declaration, whenever it is known statically. For example if rank of an array `A` of type `T` is known to be two, it can be declared as `val A:Array[T](2);`. This specialization provided substantial performance improvements over unspecialized code as shown in section 4.3.

4.3 Evaluation and Comparison

In order to evaluate the effectiveness of our strategy to use simple arrays when we have the required information and fall back to region arrays when necessary, we did a performance comparison of simple arrays, region arrays and region arrays with rank specialization. For this experiment we used some of the benchmarks used in the previous McFor project [13]. For this paper we present the results for the following seven benchmarks.

- *bubble* is the standard bubble sort. We chose this because it involves nested loops and consists of many array read and copy operations.
- *capr* computes the capacitance per unit length of a coaxial pair of rectangles. It involves four methods and operations on 2-dimensional matrices. It is also dominated by a large number of 2-dimensional array accesses.
- *dich* finds the Dirichlet solution to Laplace's equation. It involves mathematical operations on a 2-dimensional matrix and is dominated by a large number of 2-dimensional array accesses inside nested loops.
- *fiff* is a finite difference solution to a wave equation. It is dominated by a number of trigonometric operations and involves 2-dimensional matrix data.
- *mbrt* computes a mandelbrot set. The main features of this benchmark are computations involving complex numerical values and loops.
- *nb1d* simulates the gravitational movement of a set of objects. It involves computations on column vectors inside nested loops.
- *nb1d_a* is a version of *nb1d* that uses a specialized version of the MiX10 library which has specialized methods for column vectors.

Our previous experiments showed the performance characteristics for X10 code generated by MiX10 and compiled with different X10 compiler options [12], and we noted the overheads when using region arrays. Based on those results we developed the techniques presented in this paper, and we compare the performance of X10 code generated with different kinds of arrays (region arrays, region arrays with rank specialization and simple arrays).

X10 **compilation flow:** X10 provides a native(C++) backend and a separate managed(Java) backend. Both the backends provide various switches to manage optimization of generated binary or bytecode. According to [7] two switches `-O` and `-NO_CHECKS` are important for achieving acceptable multi-dimensional array performance. `-O` turns on the X10 compiler's optimizations and `-NO_CHECKS` turns off the array bounds checking.

Experimental setup: We compiled the X10 code generated by our current implementation of MiX10 using both C++ and Java backends with following optimization switches: `-NO_CHECKS` and `-O -NO_CHECKS`. We also tested our benchmarks for correctness using small data sizes. For performance measurements, we chose a data size for which the original Matlab code took around 100 seconds to execute. All the programs were executed on a machine with Intel(R) Core(TM) i7-3820 CPU @ 3.60GHz processor and 16 GB memory running GNU/Linux(3.2.0-26-generic #41-Ubuntu). The Matlab version used was R2011a and X10 programs were built and executed using pre-release version of x10-2.4 checked out from the trunk in June 2013, Oracle Java version 1.7.0 and gcc version 4.6.3. We collected the execution times (averaged over 5 runs) of Matlab programs and generated X10 code for them for all three versions of arrays for both backends for `-NO_CHECKS` and `-O -NO_CHECKS` switches. We normalized the executions times for Matlab programs to one and measured *speedups* compared to them for all executions of respective X10 programs.

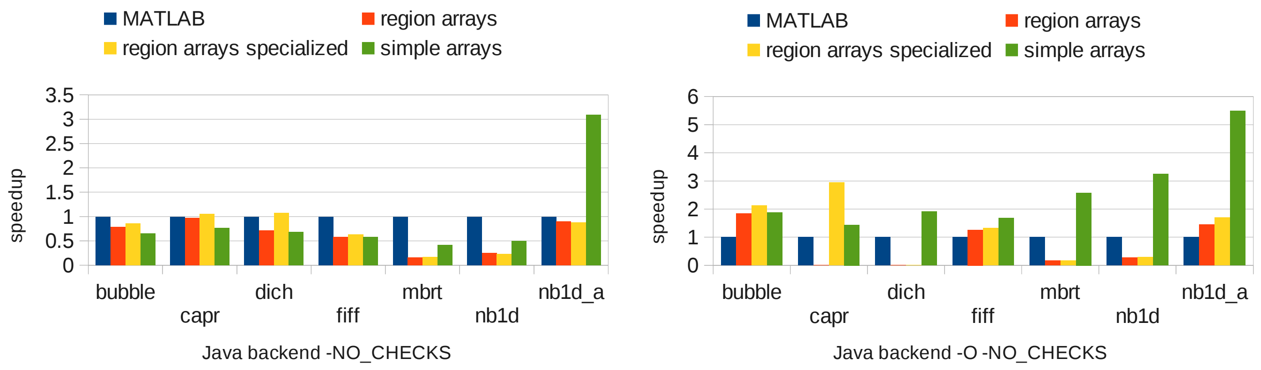
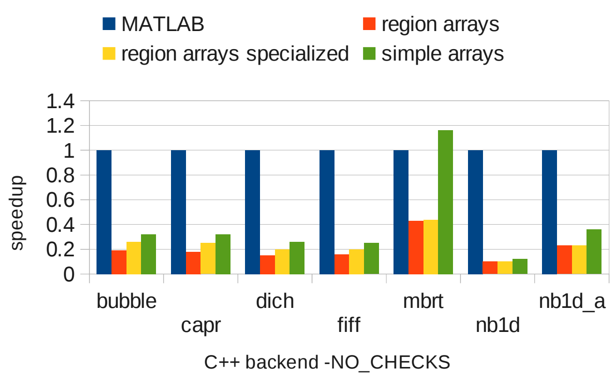


Figure 2: Speedups for Java backend

Figure 2 shows the speedups, as compared to the original Matlab code, for X10 code compiled with Java backend. The graph on the left, with unoptimized compilation of X10 code does not show speedups for simple arrays (compared to other two arrays) except for `mbrt`, `nb1d` and `nb1d_a`. This is expected because performance of simple arrays rely on the optimizations implemented for `Rail`. `mbrt` involves library calls on complex numbers, for which Java libraries are not optimized. `nb1d` and `nb1d_a` are 2.02 and 3.39 times more efficient than their respective region array versions. `nb1d` involves operations on column-vectors, which thanks to our modification to X10 runtime library to use column-major layout, are much more efficient with simple arrays. `nb1d_a` is further optimized for column-vector operations, hence the high speedup, with `-O` they give even better speedups.

With `-O` turned on, we get 1.4 to 5.5 (averaging 2.6) times speedups for simple arrays over Matlab. It is particularly interesting to note the huge difference between region arrays and simple arrays for the `capr` and `dich` benchmarks when using the `-O` flag. These benchmarks are characterized by many 2-dimensional array accesses. With the region arrays the X10 compiler inlines a large number of dynamic shape checks for each array access, which in turn causes the Java JIT compilers to fail, and to revert to the interpreter, leading to 100 times slowdowns. However, with simple arrays these benchmarks show 1.5 to 2 times speedups. `bubble` uses 1-dimensional arrays, which do not benefit from simple arrays because of the conversion overhead to `Rail`, instead they should directly be expressed as a `rail`.

Speedups for the C++ backend are shown in Figure 3. Elimination of dynamic checks by using simple arrays provide around 10 to 20 percent improvements in performance over the region array



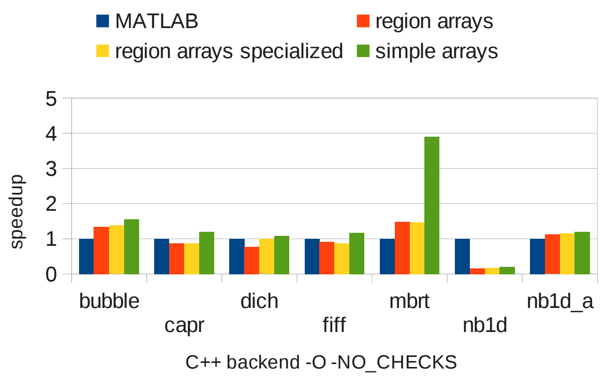


Figure 3: Speedups for C++ backend

versions. For `mbrt` we observed around 2 times better speedup. Using the `-O -NO_CHECKS` switch we consistently achieved around 20 percent better performance than MATLAB.

To conclude, by using simple arrays, significant performance gains can be achieved, specially for programs that involve large number of multi-dimensional array accesses. Even though X10 provides fairly dynamic array operations, it is very beneficial to gather static information for using X10 as a target language.

5 Introducing concurrency controls in MATLAB

MATLAB programmers often recognize the parallel nature of computations involved in their programs but cannot express it due to the lack of fine-grained concurrency controls in MATLAB. Some concurrency can be achieved using controls like `parfor` and other tools in Mathwork's parallel computing toolbox, but this has several drawbacks: (1) the parallel toolbox is limited in terms of scaling (MATLAB currently supports only up to 12 workers *processes* to execute applications on a multicore processor [16]); (2) the parallel toolbox must be purchased separately, so not even all licensed MATLAB users will have it available; and (3) MATLAB's concurrency is often slower compared to X10's concurrency controls (as shown in section 5.4).

Vectorization [6] is a technique to convert loop-based scalar operations to vector operations, for which MATLAB is optimized. So, another way of exposing parallelism in MATLAB is to optimize these instructions to perform the computations concurrently on the elements of the vector.

5.1 Introduction to X10 concurrency controls:

The Asynchronous Partitioned Global Address Space Model [9] provides the following four types of concurrency constructs [10]:

1. *Async* is the fundamental concurrency construct in X10. `async S` creates an "activity" that executes statement `S` in parallel to the parent activity and share the same heap memory as the parent.

[6] `http://www.mathworks.com/help/matlab/matlab_prog/vectorization.html`

2. *Finish* is a construct that waits on all the activities spawned transitively from within its scope, to terminate.

3. *Atomic* specifies a set of statements to be executed in a single step with respect to all other activities in the system. *When* is a conditional atomic statement.

4. *at* simply specifies a X10 *place*(a processing unit in X10) at which a certain activity should be executed.

5.2 Introducing concurrency controls in Matlab

In order to enable Matlab to be compiled for high performance computing it is important to let programmers exploit fine-grained concurrency in their Matlab programs. Due to the lack of fine-grained concurrency controls in traditional Matlab, we decided to introduce such controls in Matlab that can be translated by our MiX10 compiler to analogous concurrency controls in X10. However it was important that introduction of such controls should not have any side-effects when compiled by traditional Mathworks' Matlab compiler, so we introduced them as structured special comments.

We introduced the following concurrency constructs in Matlab: (1) `%!async`, (2) `%!finish`, (3) `%!atomic`, (4) `%!when(condition)` (5) (where `condition` is a boolean expression) and (6) `%!at(p)` (where `p` is an integer value denoting a place in X10). Programmers can express these constructs before the statements that they want to control and specify the end of a control by using `%!end` after the statements. Note that because of the preceding `%` sign these constructs will be treated like comments by other Matlab compilers and will not cause any side effects. Figure 4 shows an example of how to use these controls in Matlab followed by the generated X10 code for it.

```
function [x] = parallelFoo(a)
  %!finish
  for (i  = 1:length(a))
    %!async
    a(i )=a(i)*2;
    %!end
  end
  %!end
end
```

```
static def parallelfoo (a: Array_1[Double]){
  var mc_t2: Double = Mix10.length(a);
  var mc_t4: Double = 1;
  var i: Double;
  finish {
    for (i in (mc_t4 as Long)..(mc_t2 as Long)){
      async{
        var mc_t0: Double;
        mc_t0 = mtimes(a(i as Int −1), 2) ;
        a(i as Int −1) = mc_t0 ;
      }
    }
  }
  val x: Array_1[Double] = new Array_1[Double](a);
  return x;
}
```

Figure 4: Example of introduced concurrency controls, Matlab with introduced concurrency on the left, generated X10 on the right.

5.3 Handling parfor and vectorized instructions

For high performance computing, besides introducing new concurrency controls, it is also important to support important parallelization instructions supported by Matlab by default (with the use of parallel computing toolbox) and to exploit concurrency in other kind of instructions that can benefit from parallelization.

Supporting parfor instruction: `parfor` (or parallel for loop) is an important parallelization control provided by the MATLAB parallel computing toolbox that can be used to execute each iteration of the for loop in parallel with each other. The challenge was to implement it with X10's concurrency controls while maintaining its complex semantics and aiming for better performance than provided by the parallel computing toolbox.

There are three important semantic characteristics of MATLAB's `parfor` loop: (1) the scope of variables inside a `parfor` loops, including the loop index variable, is limited to each iteration; (2) if a variable defined outside the loop is modified inside the loop such that its value after the loop is dependent on the sequence of execution of iterations, then its value after the loop is set to its value before the loop; and (3) if a variable defined outside the loop is modified in a reduction assignment i.e., the final value after the iterations is independent of the order of execution of iterations, the updated value is retained after the `parfor` loop. Consider the MATLAB code given on the left of Figure 5.

```
function [] = saneParfor(v)
d = v;
x=0;
A=zeros(1,10);
parfor  i  = 1:10
    x = x+i;
    d = i*2;
    A(i)  = d;
end
disp(d);
end
```

```
static def saneParfor (v: Double)
{ var d: Double = v;
  var x: Double = 0;
  val A: Array_1[Double] =
     new Array_1[Double](Mix10.zeros(1, 10));
  var mc_t3: Double = 1;
  var mc_t4: Double = 10;
  finish {
    for (i in (mc_t3 as Long)..(mc_t4 as Long))
     async {
       atomic x = Mix10.plus(x, i as Double);
       var mc_t2: Double = 2;
       var d_local: Double =
          Mix10.mtimes(i as Double, mc_t2);
       A(i as Long −1) = d_local ;
     }
  }
}
```

Figure 5: Example of `parfor`, MATLAB with `parfor` on the left, generated X10 on the right.

Here `x = x+i;` is a reduction assignment [15] statement. The value of `d` is local to each iteration and the initial value before the loop is retained after the loop. Note that the value of `d` outside the loop is invisible inside the loop. For statement `A(i) = d;`, each iteration modifies a unique element accessible only to it, hence the final value of `A` is independent of order of execution; thus its value is updated after the loop. Our MIX10 translator uses the following strategy to translate it to X10:

1. Introduce `finish` and `async` constructs to control the flow of statements in parallel.
2. Any variable defined inside the loop and not declared outside the loop previously is declared inside the `async` scope to make it local to the iteration.

3. Any variable defined inside the loop that is previously defined outside the loop and is not a reduction variable is replaced by a local temporary variable defined inside the loop.

4. Statements identified to be reduction statements are made atomic by using the `atomic` construct in X10.

An example of the X10 code generated for the example MATLAB code is given on the right side of Figure 5.

Thus we can translate the `parfor` in MATLAB to semantically equivalent code in X10 and since X10 can handle massive scaling, we can get significantly better performance for X10 compared to MATLAB as shown by our experimental results.

parallelizing vectorized instructions: The use of vectorized instructions is another optimization technique used by MATLAB to speedup single operations on multiple scalar values by combining scalar values in a vector and executing the operation on the vector. Such *Single instruction, multiple data* style operations are good candidates for parallelization. However, efficiency of parallelization of such operations depends on the size of the vector, the complexity of the operation involved, and the executing hardware. Thus, in order to make it most effective, we wanted to provide full support for parallelization of vector instructions and give the programmer the ability to control when the vector operation is executed concurrently, based on the size of the vector.

Our solution to the problem is to introduce a parallelization specialization in the MIX10's builtin handling framework. We implemented a concurrent version of the relevant builtin operations that can operate in a parallel fashion on vectors of arbitrary sizes. We also introduced a compiler switch for MIX10 that lets programmers specify a vector length threshold for all builtins or a specific builtin above which the concurrent version of the builtin will be executed. For example, if the user wants an operation `sin(A)` to be executed concurrently only if `A` is a vector of length greater than, say, 1000; then while invoking the MIX10 compiler she can specify the threshold by using the switch `-vec_par_length sin=1000`. MIX10 will generate a call to the concurrent version of `sin()` if the length of `A` is greater than `1000` else it will call the sequential version. Using the `-vec_par_length` switch programmer can specify threshold for one or more or all builtin methods. For example `-vec_par_length all=500 sin=1000 cos=1000` will set the threshold for `sin()` and `cos()` to `1000` and to `500` for all other builtins.

5.4 Evaluation

To evaluate the performance of supporting parallel constructs in MATLAB, we performed two experiments. First we compared speedups of X10 sequential version, MATLAB `parfor` version and X10 parallel version (generated by MIX10) to MATLAB sequential version, normalized to one. We did this for both Java and C++ backends. This experiment was done on a 4-core processor. For the second experiment we repeated the first experiment on a 10-core processor and compared the speedups for the two machines. The hardware for first experiment and software for both experiments was the same as in 4.3. The 10-core machine had an Intel(R) Xeon(R) CPU E7- 4850 @ 2.00GHz processor and 64 GB memory. We considered the best configuration of number of worker threads (worker pools for MATLAB) and compiler switches for each execution and considered the average of 5 runs for each execution. We used three benchmarks: (1)*Matmul*, which is the standard matrix multiplication, (2)*mcpi*, Monte Carlo method for calculating the value of π and (3)*nb1d*, which is the same used in 4.3 and involves computations on column-vectors inside nested for loops.

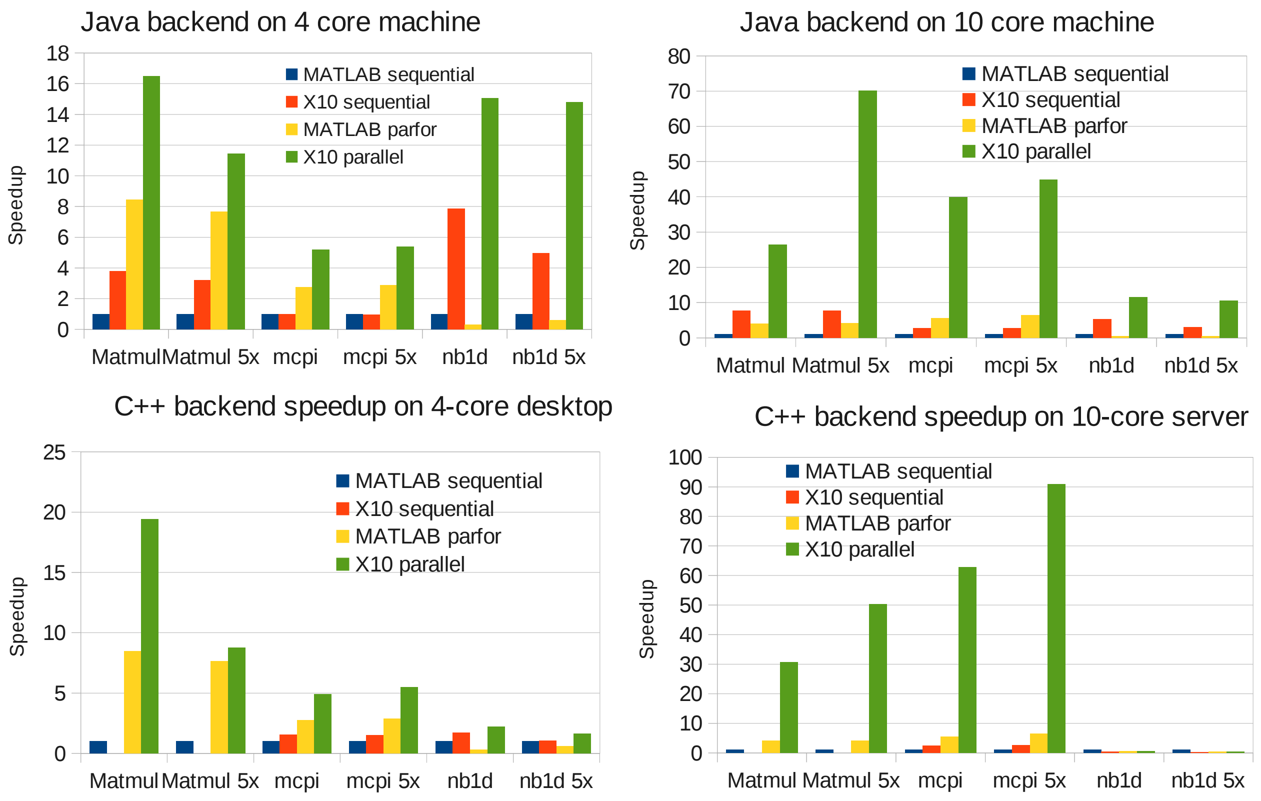


Figure 6: Speedups for parallel executions

Figure 6 shows the speedup graphs for Java and C++ backends on both the machines compared to normalized sequential MATLAB code. Note that 5x signifies a scale of 5 times in terms of input data size. It is exciting to note that the parallel version of X10 is almost always significantly faster than MATLAB's sequential and `parfor` versions. On the 4-core machine speedups range from 2 to 20 times for C++ backend and 5 to 16 times on Java backend over sequential MATLAB code and around 2 times over parallel MATLAB code. On the 10-core machine, speedups are even more exciting, ranging from 30 to 90 times for C++ backend over sequential MATLAB code and around 10 times over parallel MATLAB code and 10 to 70 times for Java backend code over sequential MATLAB code and around 6-7 times over MATLAB parallel code. Thus, X10 scales significantly better than MATLAB.

`nb1d` [7] is a noticeable exception. It creates a large number of concurrent activities each with significantly small computation. Even the MATLAB parallel version is twice as slow as the sequential version. For C++ backend on 4-core machine we get around 2 times speedup as compared to only 0.5 times on 10-core machine. This slowing down for 10-core machine is also seen for Java backend for which the speedup on 4-core machine is 15 times and on 10-core machine is 10 times. Also, speedups for X10 parallel code are around 10% better than sequential X10 code for C++ and over 2 times better for Java backend.

[7]Note that we have used the version of nb1d with MiX10 library functions specialized for column vectors

To summarize, X10 concurrency constructs are really powerful and much more efficient than MATLAB parallel computing toolbox. Although, it is important to use concurrency only with programs where each concurrent unit performs substantial amount of computations.

6 Related Work

The work presented in this paper provides an alternative to Mathworks' de facto proprietary implementation of MATLAB. Our approach is open and extensible and leverages the high-performance computing abilities of X10.

Although our focus is on handling MATLAB itself, notable open source alternatives of MATLAB like Scilab [11], Julia [2] and NumPy [20] and Octave [1] provide limited concurrency features. They concentrate on providing open library support and have not tackled the problems of static compilation. We are investigating if there is any way of sharing some of their library support with MIX10. The MEGHA project [18] provides an interesting approach to map MATLAB array operations to CPUs and GPUs, but supports only a very small subset of MATLAB.

There have been previous research projects on static compilation of MATLAB which focused particularly on the array-based subset of MATLAB and developed advanced static analyses for determining shapes and sizes of arrays. For example, FALCON [19] is a MATLAB to FORTRAN90 translator with sophisticated type inference algorithms. The McLab group has previously implemented a prototype Fortran 95 generator [13], and is developing the next generation Fortran generator in parallel with the MIX10 project. Some of the solutions can be shared between the projects, especially the parts which extend the Tamer.

MATLAB Coder is a commercial compiler by MathWorks [14], which produces C code for a subset of MATLAB.

In terms of source-to-source compilers for X10, we are aware of two other projects. StreamX10 is a stream programming framework based on X10 [23]. StreamX10 includes a compiler which translates programs in COStream to parallel X10 code. Tetsu discusses the design of a Ruby-based DSL for parallel programming that is compiled to X10 [21].

7 Conclusions and Future Work

This paper is about providing a bridge between two communities, the scientists/engineers/students who like to program in MATLAB on one side; and the programming language and compiler community who have designed elegant languages and powerful compiler techniques on the other side.

The X10 language has been designed to provide high-level array and concurrency abstractions, and our main goal was to develop a tool that would allow programmers to automatically convert their MATLAB code to efficient X10 code. In this way programmers can port their existing MATLAB code to X10, or continue to develop in X10 and use our MIX10 compiler as a backend to generate X10 code. Since X10 is publicly available under the Eclipse Public License (`x10-lang.org/home/x10-license.html`), users could have efficient high-performance code that they could freely distribute. Further, X10 itself can compile the code to either Java or C++, so our tool could be used in a tool chain to convert MATLAB to those languages as well.

Our tool itself is part of the McLab project, which is entirely open source. Thus, we are providing infrastructure for other compiler researchers to build upon this work, or to use some of our approaches to handle other popular languages such as R.

In this paper we demonstrated the end-to-end organization of the MiX10 tool, and we identified that the correct handling of arrays and concurrency features were the key challenges. We developed a custom version of the simple rail-backed X10 arrays, and identified where and how this could be used for generating efficient X10 code. For cases where precise static array shape and type information is not available, we showed how we can use the very flexible region-based arrays from X10, and our experiments demonstrated that it is very important to use the simple rail-backed arrays, for both the Java and C++ backends.

One of the main motivations of choosing X10 as the target language is that it supports high-performance computing, which is often desirable for the computation-intensive applications developed by the engineers and scientists. We have identified three main ways of effectively exposing and using the X10 concurrency. The first is by implementing an effective translation of the Matlab `parfor` construct to semantically equivalent X10. The second is by exposing the key X10 concurrency constructs to the Matlab programmer via structured comments. These comments will be ignored by standard Matlab implementations, but will be used to generate concurrent X10 code by our MiX10 tool. Finally, we provided a parallel version of Matlab vector operations, and compile-time switches to control when the parallel versions should be used.

Our experiments showed significant performance gains for our generated parallel X10 code, as compared to Matlab's parallel toolbox. This confirms that compiling Matlab to a modern high-performance language can lead to significant performance improvements.

Based on our positive experiences to date, we plan to continue improving the MiX10 tool. The code that we generate is already quite clean, but we would like to apply further transformations on it to aggregate some low-level expressions, and to make the generated code look as "natural" as possible. We also would like to experiment further to find the best way to tune the generated code for different sorts of parallel architectures. We also hope that other research groups will use our infrastructure.

References

[1] GNU Octave. `http://www.gnu.org/software/octave/index.html`.

[2] J. Bezanson, S. Karpinski, V. B. Shah, and A. Edelman. Julia: A Fast Dynamic Language for Technical Computing. *CoRR*, abs/1209.5145, 2012.

[3] J. Doherty. McSAF: An Extensible Static Analysis Framework for the MATLAB Language. Master's thesis, McGill University, December 2011.

[4] J. Doherty and L. Hendren. McSAF: A static analysis framework for MATLAB. In *Proceedings of ECOOP 2012*, pages 132–155, 2012.

[5] J. Doherty, L. Hendren, and S. Radpour. Kind analysis for MATLAB. In *In Proceedings of OOPSLA 2011*, pages 99–118, 2011.

[6] A. Dubrau and L. Hendren. Taming MATLAB. In *Proceedings of OOPSLA 2012*, pages 503–522, 2012.

[7] IBM. Performance tuning. http://x10-lang.org/documentation/practical-x10-programming/performance-tuning.html, Feb. 2012.

[8] IBM. X10 programming language. http://x10-lang.org, Feb. 2012.

[9] IBM. Apgas programming in x10 2.4, 2013. http://x10-lang.org/documentation/tutorials/apgas-programming-in-x10-24.html.

[10] IBM. An introduction to x10, 2013. http://x10.sourceforge.net/documentation/intro/latest/html/node4.html.

[11] INRIA. Scilab, 2009. http://www.scilab.org/platform/.

[12] 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.

[13] J. Li. McFor: A MATLAB to FORTRAN 95 Compiler. Master's thesis, McGill University, August 2009.

[14] MathWorks. MATLAB Coder. http://www.mathworks.com/products/matlab-coder/.

[15] Mathworks. Reduction variables. http://www.mathworks.com/help/distcomp/advanced-topics.html#bq_of7_-3.

[16] MathWorks. Parallel computing toolbox, 2013. http://www.mathworks.com/products/parallel-computing/.

[17] C. Moler. The Growth of MATLAB and The MathWorks over Two Decades. http://www.mathworks.com/company/newsletters/news_notes/clevescorner/jan06.pdf.

[18] A. Prasad, J. Anantpur, and R. Govindarajan. Automatic compilation of MATLAB programs for synergistic execution on heterogeneous processors. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI '11, pages 152–163, New York, NY, USA, 2011. ACM.

[19] L. D. Rose and D. Padua. Techniques for the translation of MATLAB programs into Fortran 90. *ACM Trans. Program. Lang. Syst.*, 21(2):286–323, 1999.

[20] Scipy.org. Numpy. http://www.numpy.org/.

[21] T. Soh. Design and implementation of a DSL based on Ruby for parallel programming. Technical report, The University of Tokyo, Jan. 2011.

[22] The Mathworks. Technology Backgrounder: Accelerating MATLAB, September 2002. http://www.mathworks.com/company/newsletters/digest/sept02/accel_matlab.pdf.

[23] H. Wei, H. Tan, X. Liu, and J. Yu. StreamX10: a stream programming framework on X10. In *Proceedings of the 2012 ACM SIGPLAN X10 Workshop*, X10 '12, pages 1:1–1:6, New York, NY, USA, 2012. ACM.