Spar: a set of extensions to Java for scientific computation*

C. van Reeuwijk, F. Kuijlman, and H.J. Sips
Delft University of Technology
email: C.vanReeuwijk@cs.tudelft.nl

ABSTRACT

In this paper we present a set of language extensions that improve the expressiveness and performance of Java for scientific computation. First of all, the language extensions allow the manipulation of multi-dimensional arrays to be expressed more naturally, and to be implemented more efficiently. Furthermore, data-parallel programming is supported, allowing efficient parallelization of a large class of operations on arrays. We also provide language extensions to construct specialized array representations, such as symmetric, block, and sparse matrices. These extensions are: tuples, parameterized types, array subscript overloading, and the `inline` modifier. These extensions are not only useful to construct special array representations, but are also useful in their own right. In particular tuples are a useful addition to the language. Finally, we add complex numbers as a primitive type to the language.

We evaluate our language extensions using performance results. We also compare relevant code fragments of our extended language with standard Java implementations and language extensions proposed by others.

General Terms

Language Design, Scientific Computation

Keywords

Java, multi-dimensional array, tuple, parameterized type

1. INTRODUCTION

It is more and more apparent that there is a need for a new standard programming language for scientific applications. Traditionally, Fortran has been used for this purpose, but it lacks the strict typing, object orientation, and more rigorous language design of languages such as C, C++, and Java. For this reason more and more scientific programs are written in these languages.

However, Fortran still has a number of features that make it particularly useful for scientific programs, namely multi-dimensional arrays, complex numbers, and, in later versions, array expressions and data-parallel programming. Moreover, Fortran compilers tend to produce highly efficient code compared to compilers for other languages.

We think that it is time to replace Fortran with a modern language that also provides these features. Furthermore, experience with scientific programs in Fortran has shown that support for structured parallel programming and for specialized arrays (block, sparse, symmetric, etc.) is also desirable.

Based on these principles we have designed a set of language extensions for Java. We call the resulting language Spar/Java, where Spar is the name of the language extensions for scientific computing.

This paper expands on [22], where the design of an earlier version of Spar was described. The previous paper described the initially designed language constructs; in this paper we describe the language constructs of the latest, implemented, version of Spar, and compare Spar/Java with Java, C, C++, and HPF with respect to expressiveness and performance. We also compare our language constructs with other language extensions for Java, and discuss the implementation of Spar.

Support for multi-dimensional arrays is described and evaluated in §2. The language constructs for parallelization are briefly described in §3. They are described and evaluated in more detail elsewhere [24, 23].

To support specialized arrays, we could have provided a pre-defined set of specialized types, but there is such a wide variety of specialized array types that this approach is too inflexible. Instead, we provide a toolkit to let the user build his/her own array representation. This toolkit consists of: overloading of the subscript operator; parameterized classes; tuples and vector tuples; and the `inline` modifier. In §4 the reasons for providing this set of language extensions, the definition of the extensions, and an evaluation of the resulting support for specialized arrays are given.

Spar/Java supports complex numbers because they are important for scientific computation. The language extensions for this are described in §5. Complex numbers are used in the example of §8.

Most of the language constructs in the toolkit are useful beyond their original purpose, and we have been careful not

*This research was supported by NWO (project 'Automap'), Esprit (LTR Project 'Joses' (#28198)), and Delft University of Technology (DIOC project 'Ubicom').

to spoil their generality. A number of noteworthy other applications of the constructs are discussed in §6.

To demonstrate the practicality of our language extensions, we have implemented a compiler for Spar/Java. It is described in §7. We evaluate the performance of our demonstration compiler with the FFT benchmark from the NAS benchmark suite, see §8.

Since we extend Java, in principle the JVM instruction set must also be extended, because the Java Virtual Machine uses an instruction set that has been designed specifically for Java. Since for several reasons this is not practical, it is worthwhile to consider alternative solutions. We discuss these issues in §9.

Finally, related work is discussed in §10, we draw some conclusions in §11, and we describe future work in §12.

For the sake of brevity, in this paper we only describe the language constructs by example. In the Spar Language Specification [21] the constructs are strictly defined in a manner similar to the Java Language Specification [10].

2. MULTI-DIMENSIONAL ARRAYS

Many scientific programs work on multi-dimensional arrays. For example, many problems are formulated as matrix operations. In a programming language, a matrix is most naturally represented as a two-dimensional array. As another example, many simulation problems are solved by introducing a regular two-dimensional or three-dimensional grid. In a programming language, such a grid is most naturally represented as a multi-dimensional array.

In many languages, including Java, it is assumed that it is sufficient to provide one-dimensional arrays, and that multi-dimensional arrays can be represented as arrays of arrays. We will call this the *nested array* representation.

Unfortunately the nested array representation has some drawbacks that can greatly influence performance:

- Memory layout for nested arrays is dependent on details of the memory allocator. The rows of an array may be scattered throughout memory. This deteriorates cache behavior.
- For nested arrays, the compiler must take into account array aliasing (two array rows are the same within an array, or even between arrays), and ragged arrays (array rows have different lengths). This complicates code optimization.
- Garbage collection overhead for nested arrays is larger, since all rows of the array are administrated independently.
- Nested arrays are difficult to use in data-parallel programming, since extensive analysis is required before efficient communication code can be generated.

For these reasons we add true multi-dimensional arrays to Java. For example, a two-dimensional array in Spar/Java is declared and used as follows:

```
int a[*,*] = new int[10,10];

for( int i=0; i<a.GetSize(0); i++ )
  for( int j=0; j<a.GetSize(1); j++ )
    a[i,j] = i+j;
```

In general, arrays are indexed by a list of expressions instead of a single expression. Similarly, in an array creation expression a list of sizes is given instead of a single size. These features are straightforward generalizations of existing Java language constructs.

In the declaration of array types it is now also necessary to specify the number of dimensions (the *rank*) of an array. This is done with a comma separated list of '`*`'. For compatibility with Java, an empty list is not interpreted as a zero-dimensional array, but as a one-dimensional array. Thus, the types `int b[*]` and `int b[]` are equivalent.

Spar also allows an alternative notation to specify the rank. For example, the two type expressions `int [*,*]` and `int [*^2]` are equivalent. In general an arbitrary expression is allowed after the '`^`', provided that it is a non-negative compile-time constant of type `int`. Although the alternative notation is less readable, it allows for much more flexibility. For example, it allows the rank of an array to be dependent on a parameter of a parameterized type.

The `GetSize(int)` shown in the example is a method that returns the size of the array in the given dimension. This is an implicitly defined method on the array, similar to the `clone()` method that is defined on arrays in standard Java. There also is a `GetSize()` method without parameters that returns a vector tuple with the lengths of the array. See §4.2 for a description of vector tuples.

2.1 An example: array transposition

As a simple illustration of the costs of multi-dimensional versus nested arrays, consider the following loop, which copies the transpose of array B into array A:

```
for( int i=0; i<M; i++ )
  for( int j=0; j<M; j++ )
    A[i][j] = B[j][i];
```

The variant using multi-dimensional arrays is:

```
for( int i=0; i<M; i++ )
  for( int j=0; j<M; j++ )
    A[i,j] = B[j,i];
```

We measured the execution times of these programs for `int` arrays of 2197×2197 elements, for 40 runs of this loop. We also measured the execution times of the analogous programs working on three-dimensional arrays of $169 \times 169 \times 169$ elements, again for 40 runs of the loop.

Note that in both cases the arrays contain the same number of elements (4826809), so the difference in execution time between these programs is an indication of the overhead of using an extra dimension.

We compiled both versions of the program with our own 'Timber' compiler (see §7), and measured the execution time of the resulting programs. For comparison we also measured the execution time of the Java variants of these programs using the Java HotSpot 1.3.0 Client VM. The programs were executed on a 466 MHz Celeron with 256 MB of memory running Linux 2.2.18. The Timber compiler used the Gnu C++ 2.95.2 with the options `-O6 -funroll-loops -fomit-frame-pointer`. The shown execution times are in seconds.

Array type - compiler	2D array	3D array
Nested - Timber	64.9	123.7
Nested - Hotspot	60.6	84.6
Multidim. - Timber	6.3	7.2

The significantly larger execution times of the programs with nested arrays is caused by several factors. An indication of the overhead of one factor, bounds checking, can be found by disabling the generation of bounds checking code in the Timber compiler, and measuring the execution times again. In that case the results are (bounds checking of the HotSpot compiler cannot be disabled):

Array type - compiler	2D array	3D array
Nested - Timber	44.4	70.5
Nested - Hotspot	–	–
Multidim. - Timber	6.3	7.1

As these results indicate, for the multi-dimensional array representation the overhead of bounds checking is limited. For the nested array representation the overhead of bounds checking is larger, but there are other significant factors that contribute to the larger execution times, such as null pointer checks, memory layout issues and more complicated array index calculations.

3. PARALLELIZATION CONSTRUCTS

Since parallelization of scientific programs is often very important, we want to support parallelization in our compiler. As a minimum, we want to support data-parallel operations on multi-dimensional arrays.

To support this feature a number of language constructs had to be added; we will briefly describe them in this section.

3.1 The `each` and `foreach` statements

The parallelization constructs have been designed to make them as safe as possible, while still allowing the compiler to derive opportunities for parallelism from them. For this purpose the `each` and `foreach` commands are provided.

Given a program block such as:

```
each { a = 1; b = 1; }
```

the compiler may choose one of the execution orders `a = 1; b = 1;` or `b = 1; a = 1;`. In general the statements in an `each` block may be of arbitrary complexity. They are executed in arbitrary order, but once the execution of a statement is started, it must be completed before the next one is started. Thus, execution of the code fragment:

```
each { a = 1; { a = 2; b = a; } }
```

will always result in `b` having the value 2. However, `a` may have the value 1 or 2, depending on the order in which the two elements of the `each` were executed.

The `foreach` statement is a parameterized version of the `each` statement. The iteration range is specified as a lower bound (default 0), upper bound, and stride (default 1). These values are evaluated exactly once before execution of the loop. The iteration variable is implicitly declared as a local `final int` variable of the loop. The iteration range specification–consisting of iteration variable, bounds and stride–is called a *cardinality*. Similar to the `each` statement, the compiler may choose any order for the execution of the iterations. Although the iterations in a `foreach` statement can be executed in arbitrary order, once an iteration is started, it must be completed before the next one is started. Parallel execution is only allowed when there is no observable interference between iterations. To discover this, the compiler must do some data-dependency analysis (although less extensive than for sequential loops).

For example, the following `foreach` statement fills an array with 0:

```
foreach( i :- 0:n ) a[i] = 0;
```

We can safely use a `foreach` since we are not interested in the order in which the array is filled.

The `foreach` statement cannot use the iteration range specification of the Java `for` statement, since the iterator value for each iteration is defined relative to the iterator of the previous iteration. This makes it less suitable for parallel loops.

3.2 Cardinality lists

The `foreach` shown in the previous section contains one cardinality. In general a `foreach` statement can have an arbitrary number of cardinalities. For example, a two-dimensional array `a` can be filled with 0 with the following loop:

```
foreach( i :- 0:a.GetSize(0), j :- 0:a.GetSize(1) )
  a[i,j] = 0;
```

Such a cardinality list specifies a multi-dimensional iteration space. In case of the `foreach` statement this iteration space can be enumerated in arbitrary order. Note that this is different from a nested pair of `foreach` statements. The loops:

```
foreach( i :- 0:a.GetSize(0) )
  foreach( j :- 0:a.GetSize(1) ) a[i,j] = 0;
```

also fill the array `a` with 0, but in this case all iterations of the inner `foreach` must be completed before the next iteration of the outer `foreach` can be executed.

3.3 Vector cardinalities

As a further refinement, *vector cardinalities* are supported, where a vector tuple is iterated over a multi-dimensional iteration space. See §4.2 for a description of vector tuples. For example, the zeroing of array `a` can also be written as follows:

```
foreach( v :- 0:a.GetSize() ) a@v = 0;
```

3.4 Cardinality lists and `for` loops

Cardinality lists are also useful for sequential loops, since they allow a more compact notation for a very common case. Moreover, loops with cardinality lists tend to be easier to analyze, which makes it easier for the compiler to generate efficient code.

3.5 `for` statement inlining

Finally, Spar allows `for` loops to be unrolled explicitly, by annotating the loop with the `inline` modifier. Such an `inline for` statement must have a cardinality list with only compile-time constants.

The `inline for` statement is necessary for operations on tuples, see §4.2, and is also useful to force loop unrolling for performance reasons.

For example, the following loop:

```
inline for( i :- 0:4 ) a[i] = i;
```

is expanded by the compiler to:

```
a[0] = 0;
a[1] = 1;
a[2] = 2;
a[3] = 3;
```

4. SPECIALIZED ARRAY REPRESENTATIONS

An important goal of our languages extensions is to provide good support for more specialized array representations such as block, symmetric, sparse, etc. arrays. A number of language extensions have been introduced to allow efficient, generic, and readable implementations of such arrays and of their uses. These extensions are:

- Overloading of the subscript operator.
- Parameterized classes.
- Tuples and vector tuples.
- The `inline` modifier.

Overloading of the subscript operator greatly improves the readability of the manipulation of specialized arrays. Parameterized classes allow generic implementations of specialized arrays. In particular, the implementations can be generic in the element type and the number of dimensions. To be able to express manipulations on arrays with different numbers of dimensions generically, it is necessary to introduce vector tuples. Finally, to ensure that the use of specialized arrays is as efficient as the use of standard arrays, it is necessary that some methods are always inlined, in particular methods that access array elements. This may not always be detectable by a compiler, so it is important to allow the user to force inlining of some methods.

4.1 Parameterized classes

In its simplest form support for specialized array representations can be provided by a standard Java class. The class `java.util.Vector` can be seen as an example. Unfortunately, with this approach it is not possible to abstract from parameters such as the element type, rank, or from 'tuning' parameters such as block sizes or allocation increments.

For example, for `java.util.Vector` it would be desirable to provide the element type of the vector as a parameter. This would help to enforce stricter type checking on the elements of a vector. It would also allow a more efficient implementation, in particular for vectors of primitive types, since in the current implementation elements of primitive types must be wrapped in instances of classes such as `java.lang.Double`.

Support for some form of class parameterization is therefore highly desirable. A number of proposals have been made to add class parameterization to Java [6, 14, 28]. Also, there is a proposal in the Java Community process [5] to add [6] to standard Java.

To avoid having to extend the existing JVM definition– which would render all existing JVM implementations obsolete– most proposals only allow reference classes as parameters. With this restriction, parameterized classes can be rewritten as operations on an unrestricted version of the class, and a number of casts and assertions. Unfortunately, this restriction makes these proposals unsuitable for our purposes, since we require parameterization with primitive types (e.g. for element types of the specialized arrays), and with numeric values (e.g. for numbers of dimensions).

Therefore, Spar provides a different class parameterization mechanism, based on template instantiation[1]. Using this approach, very efficient class instantiation is possible. Moreover, arbitrary type parameters and value parameters can be supported.

For example, in Spar/Java we provide a typed vector in `spar.util.Vector`, which is implemented as follows (simplified):

```
final class Vector(| type t |)
{
    protected t elementData[] = null;

    public Vector(){}

    public Vector( int initCap ){
        ensureCapacity( initCap );
    }
    // Etc.
}
```

the sequence `(| type t |)` is the list of parameters of the class. The list of parameters can be of arbitrary length. Parameters can be of type `type`, and of primitive types. Actual parameters of a class must be types, or evaluate to compile-time constants. For every different list of actual parameters a class instance is created with the actual parameters substituted for the formal parameters.

Class `spar.util.Vector` can be used as follows:

```
// Create a new instance of an int vector with initial
// capacity 20.
Vector(| type int |) v = new Vector(| type int |)( 20 );
```

The keyword `type` is used to introduce actual type parameters. For primitive types it can be omitted. Thus, the following is also allowed:

```
Vector(| int |) v = new Vector(| int |)( 20 );
```

4.2 Vector tuples

To allow generic implementations of specialized arrays, it is necessary to allow a list of subscript expressions to be treated as a single entity, regardless of its length (and hence regardless of the rank of the subscripted array). This is easily possible by considering subscript lists as *tuples*. Thus, an ordinary array index expression such as `a[1,2]` is considered as the application of an implicit index operator on an array (`a`), and a tuple (`[1,2]`).

Spar generalizes this by allowing tuples as 'first class citizens' that can be constructed, assigned, passed as parameters, and examined, independent of array contexts. Spar also provides an explicit array subscript operator '`@`'. The following code shows tuples and the `@` operator in use:

```
[int^2] v = [1,2];          // Declare, init. tuple
int a[*,*] = new int[4,4]; // Declare, init. array
a@v = 3;                    // Assign to a[1,2]
v[0] = 2;                   // Tuple is now [2,2]
a@v = 5;                    // Assign to a[2,2]
```

Traditional index expressions are considered a special case of general array subscription where the `@` operator can be omitted. Thus, `a[1,2]` and `a@[1,2]` are considered equivalent.

[1]A common objection against the template instantiation mechanism is that it causes 'code bloat'. Provided that templates are used in moderation, we do not consider this disadvantage significant. This assessment is supported by experience: a large part of our Spar/Java compiler is implemented using a template preprocessor [20, 19], and it does not suffer from 'code bloat'.

The '@' operator can also be used in new expressions for arrays. For example:

```
[int^2] sz = [10,10];
int a[*,*] = new int@sz;
```

4.3 Subscript operator overloading

In Java, elements in array-like classes must be accessed using explicit methods, instead of the much more compact array index notation using @. For example, to swap elements 0 and 1 of a java.util.Vector instance v requires the following code:

```
Object h = v.elementAt(0);
v.setElementAt(v.elementAt(1),0);
v.setElementAt(h,1);
```

Such a notation is acceptable for occasional use, but is too awkward for intensive manipulations.

For this reason, Spar supports overloading of the index operator. If an index operator is used on an expression of a class type, this expression is translated to an invocation to a method getElement or setElement, depending on the context.

For example, assuming 'v' is a class instance, the statement:

```
v[0] = v[1];
```

is translated to:

```
v.setElement( [0], v.getElement( [1] ) );
```

Obviously, the class must implement getElement and setElement for this convention to work.

At first sight it seems more obvious to choose an existing pair of functions instead of setElement and getElement. Unfortunately, the standard Java library is not consistent on this point: java.util.Vector uses setElementAt and elementAt, java.util.Hashtable uses get and put, etc. Moreover, for reasons of generality the methods getElement and setElement take a vector tuple as parameter, which makes them incompatible with any Java method anyway.

4.4 Example

To demonstrate how the new language constructs can be used to implement specialized array representations, we implement 'elastic' arrays. Elastic arrays are similar to ordinary arrays, except that they can be grown and shrunk during their lifetime.

The implementation of the class is as follows:

```
public class ElasticArray(|type T,int n|)
{
  private T arr[*^n];
  public ElasticArray([int^n] sz){
    arr = new T@sz;
  }
  public inline T getElement([int^n] ix){
    return arr@ix;
  }
  public inline void setElement([int^n] ix,T val){
    arr@ix = val;
  }
  void SetSize([int^n] sz){
    T newarr[*^n] = new T@sz;
    [int^n] oldsz = arr.getSize();

    [int^n] overlap;
    inline for( i :- 0:n )
      overlap[i] = Math.min( oldsz[i], sz[i] );
    for( v :- 0:overlap )
      newarr@v = arr@v;
    arr = newarr;
  }
}
```

The class is parameterized with the element type of the array T, and the rank of the array n. The constructor for the class constructs a new array instance with the vector sz specifying the sizes of the dimensions of the array. The methods getElement and setElement allow index operator overloading, and the method SetSize changes the sizes of the array. All elements that are accessible in both the old and the new version of the array are copied, all other visible elements are initialized to their default value.

ElasticArray can now be used as follows:

```
ElasticArray(|int,2|) a =
  new ElasticArray(|int,2|)([10,10]);
a[5,3] = 3;
a.SetSize([8,16]);
a[2,1] = a[5,3];
```

To get an impression of the performance of such a specialized array representation, we run the array transposition benchmark of §2.1 on elastic arrays. For comparison the results for ordinary multi-dimensional arrays from §2.1 are repeated. All measurements are with our Timber compiler, on arrays of the same size.

Array type	2D array	3D array
Standard	6.3	7.2
Elastic	9.5	15.1

The difference in performance between elastic arrays and standard arrays is caused by the fact that the array is contained in an instantiation of the class ElasticArray. This prevents some optimizations, e.g. bound check simplification and some array index simplification. The performance can be improved fairly easily with more sophisticated analysis. Alternatively, it may be possible to do 'unboxing' of the ElasticArray instantiation, which essentially replaces the class with a tuple.

5. COMPLEX NUMBERS

Complex numbers are used in many scientific computations, so it is very desirable to have a compact notation and efficient support for them. The most obvious approach is to introduce a new class to represent complex numbers, and manipulations on them. This approach has been proposed, among others, by the Java Grande Forum [2] and for use with the IBM Ninja compiler [1, 13].

Unfortunately, this approach has some drawbacks: complex numbers are stored in allocated memory, manipulations on complex numbers must still be expressed as method invocations, and the complex class is still a reference type, which means that instances must be allocated, and can be aliased. To a certain extent these problems can be reduced by a smart compiler, especially if it is able to recognize the complex number class and exploit its known properties. Nevertheless, it is unlikely that such optimizations are successful in all cases.

Spar uses a more robust solution: it introduces a new primitive type complex. The operators *, /, +, and -

are generalized to handle complex numbers; and narrowing and widening conversions are generalized. Also, a wrapper class `java.lang.Complex` is added, similar to e.g. `java.lang.Double`. The class `java.lang.Complex` also contains a number of transcendental functions similar to those in `java.lang.Math`. To simplify the notation of complex constants, a new floating point suffix 'i' has been added to denote an imaginary number. Together these additions allow code like this:

```
complex c1 = 1.0+2.0i;
complex c2 = 12-3i;
complex c3 = c1*c2;
complex c4 = Complex.sin( c3 );
```

6. SPINOFFS

Many of the language constructs that we introduced for parallelization or the construction of specialized arrays are useful beyond their original purpose, and we have been careful not to spoil their generality. In a number of cases the more general applications are obvious (parameterized types, iteration spaces defined by cardinality lists), two important other 'spinoffs' are described in this section.

6.1 Other uses of tuples

As explained above, tuples of `int` elements are essential for the implementation of specialized array representations that are generic in the number of dimensions. With a little generalization it is possible to make tuples into a construct that is useful for other purposes as well.

Until now only *vector* tuples have been shown: tuples where all elements have the same type. Spar in fact supports tuples with mixed element types. These are declared and used as in the following example:

```
[int,boolean] val = [12,true];  // Declare, init. 'val'
val[0] = 8;                     // Now val = [8,true]
val[1] = false;                 // Now val = [8,false]
```

In general a tuple may have fields of arbitrary types, including reference types and other tuples.

As shown in this example, individual elements can be accessed by indexing. Index expressions must be compile-time constants, so that the compiler can determine the type of the selected element at compile time.

Elements can also be accessed by *pattern matching*. For example, assuming a tuple `val` as in the previous example, its elements can also be accessed as follows:

```
int n;
boolean b;
[n,b] = val;
```

Tuples can be used in a number of idioms that are otherwise more awkward to express. For example, the following statement exchanges the values of `a` and `b`:

```
[a,b] = [b,a];
```

Tuples also allow functions with multiple return values, such as a function that searches a list, and returns both a success flag, and the position of the match:

```
static [boolean,int] search(int a[], int val)
{
  for( int i=0; i<a.length; i++ )
    if( a[i] == val ) return [true,i];
  return [false,0];
}
```

This function would be used as follows:

```
int ix;
boolean found;
[found,ix] = search( a, 42 );
```

The standard operators have been generalized to work on tuples. Unary operators apply to each element; binary operators on two tuples apply to corresponding elements; binary operators on a scalar and a tuple apply the scalar value to each element. This way tuples can be used as lightweight representations of coordinates and other groups of values, and common calculations on these coordinates can be expressed very easily. For example, the following function calculates the point in the middle of the line between two given points:

```
static [double^3] midpoint([double^3] a, [double^3] b)
{
  return (a+b)/2;
}
```

If the same function would have to be expressed on the tuples used in Java3D [27], the function would be more cumbersome:

```
static Tuple3d midpoint( Tuple3d a, Tuple3d b ){
  return new Tuple3d(
      (a.x+b.x)/2,
      (a.y+b.y)/2,
      (a.z+b.z)/2
  );
}
```

Moreover, this variant requires dynamic memory allocation, whereas tuples do not.

6.2 Generic algorithms

The generic approach to array support also allows some algorithms to be expressed more generally. One researcher from the numerical community, Gabor Toth, considered this so important that he wrote a preprocessor for Fortran, called LASY [29], to implement this. To quote from the introduction:

> The Loop Annotation Syntax (LASY) was developed as part of the Versatile Advection Code (VAC) software package which can solve conservation laws, e.g. hydrodynamics and magnetohydrodynamics, in 1, 2, and 3 dimensional grids. The usual practice is to write a simple 1D code first, then to modify it for 2D simulations, and finally, years later, to rewrite the whole code for 3D. In the VAC project a different route was taken, namely a single general software was designed from the beginning, which can do simulations in any number of dimensions.

Such generality is not possible in standard Fortran, so the author wrote a preprocessor to generate specialized Fortran code for each number of dimensions from a generic code.

The toolkit that Spar provides to build specialized arrays also allows implementing such generic algorithms.

7. THE COMPILER

To demonstrate the practicality of our language extensions, we have implemented a compiler for Spar/Java. It largely implements Java as described in the Java Language

Specification 2nd edition [10]. To keep the compiler simple enough to implement in our research group, our current implementation does not support threads and dynamic class loading. Also, inner classes and interfaces are only partially supported. These restrictions were imposed for practical reasons, and are not fundamental to the Spar/Java language design.

The compiler does not generate JVM (Java Virtual Machine) code; since the compiler handles a superset of standard Java, it is not clear if this is possible at all. This issue is discussed in §9.

The compiler works on full programs. It consists of a frontend that generates code in the intermediate language Vnus [8], a number of parallelization engines that work on Vnus, and a backend that converts from Vnus to C++. The compiler contains optimizers that avoid compiling unused code, eliminate and optimize null pointer checks, eliminate static class administration, simplify expressions, and eliminate and optimize bound checks.

Generating C++ has as advantage that the compiler is independent of the target processor, and hence is very portable. It has as disadvantage that the optimizers in the C++ compiler are not tuned to Spar/Java programs, and sometimes must make more pessimistic assumptions about code properties. For example, since C++ pointers can point in the middle of an array, and since pointers can be cast to and from almost any other type; the compiler must take this into account when analyzing a C++ program. In contrast, Spar/Java references (which are translated to pointers), cannot point into arrays, and casting between reference types is much more restricted.

The current compiler generates code to maintain a precise root set (set of active references in the program state). This information is sufficient to implement a mark-sweep garbage collection system.

At the moment the parallelization engines generate SPMD (Single Program Multiple Data) code with explicit message passing. We demand very little of the communication library we use: we need a send and receive function, and if a broadcast or multicast is supported, we can use it. For this reason we can use almost any communication library. We currently support the communication libraries PVM, MPI, and Panda [25, 26].

The compiler is available for downloading at [18]. It is provided under the Gnu Public License (GPL).

8. A LARGER EXAMPLE: FFT

As a larger example, we have studied the FFT benchmark[2] from version 2.3 of the NASA Numerical Aerospace Simulation group (NAS) parallel benchmark suite [15, 9]. We have implemented this program in several programming languages[3]. We have measured the execution times of the different implementations.

In all cases dataset 'W' was executed: a 3D Fast Fourier Transform on a $128 \times 128 \times 32$ array. All given times are total execution times in seconds. When possible, two measurements where made: execution times were measured with bounds checking and null pointer checks enabled, and with both these checks disabled. This gives an impression of the overhead of these checks, and allows better comparison with C and C++.

[2]We have in fact implemented most of the NAS benchmarks in Spar/Java, see [16] for details

[3]The Fortran 77/MPI version is the original NAS program; Michael Frumkin from NASA was kind enough to provide us with the HPF version.

Compiler	Language	Time (s)	Time (s) (no checks)
Timber	Java	22.9	16.4
HotSpot	Java	22.7	–
Timber	Spar/Java	14.3	12.5
g++	C++	–	10.3
gcc	C	–	7.4

Figure 1: Sequential execution times of the NAS FFT benchmark using a number of different compilers and languages.

Compiler	Language	Time (s)	Time (s) (no checks)
Timber	Java	34.8	24.0
Timber	Seq. Spar/Java	24.5	21.4
Timber	Par. Spar/Java	37.2	33.2
PGI HPF	HPF	–	48.1
PGI HPF	F77	–	36.3
g++	C++	–	16.3
gcc	C	–	13.0

Figure 2: Sequential execution times of the NAS FFT benchmark on the DAS distributed supercomputer.

8.1 Sequential results

To evaluate the sequential performance of the Timber compiler, the NAS FT benchmark was implemented in C, C++, Java, and Spar, and the execution time was measured. This resulted in the execution times shown in Fig. 1.

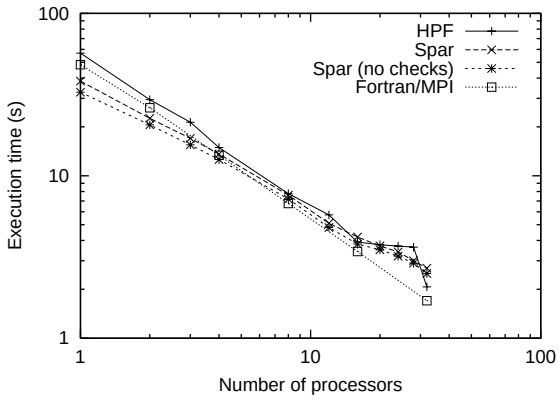
The programs were executed on a 466 MHz Celeron with 256MB of memory running Linux 2.2.18. The Timber compiler and the C and C++ versions used the Gnu C++ 2.95.2 with the options `-O6 -fomit-frame-pointer`.

The C and C++ version only differ in their representation of complex numbers: in C++ the standard `complex` class was used, and in C a gcc-specific type `__complex__` was used. The Spar compiler uses the `complex` class of C++.

In the Spar version, type `complex` and 3-dimensional arrays are used. In the Java version nested arrays are used, and complex numbers are represented by pairs of `doubles`, stored in adjacent elements of the array. The difference between the Java and Spar/Java implementations is caused by the use of nested arrays instead of flat arrays. The difference between the Spar/Java and the g++ version is caused by the different representation of Spar/Java arrays: as a pointer to an array descriptor; with a pointer to a block of elements as one of the fields. In contrast, in the C and C++ versions arrays are represented by just a pointer to a block of elements. The difference between the g++ and gcc versions is caused by the use of the standard `complex` class respectively the `__complex__` class.

8.2 Parallel results

All parallel measurements were done on the DAS dis-



proc.	Spar	Spar unchecked	Fortran /MPI	HPF
1	38.3	32.7	48.3	56.9
2	22.6	20.6	26.2	29.4
3	17.0	15.5		21.4
4	13.6	12.6	13.4	14.9
8	7.6	7.2	6.8	7.8
12	5.1	4.8		5.8
16	4.2	3.8	3.4	3.9
20	3.7	3.5		3.8
24	3.4	3.2		3.7
28	3.0	2.9		3.6
32	2.7	2.5	1.7	2.1

Figure 3: Parallel execution times of the NAS FFT benchmark. All execution times are in seconds.

tributed supercomputer [7, 4]. Each node has a 200 Mhz Pentium Pro; 64 MB RAM; 2.5 GByte local disk; and a Myrinet interface, and runs Linux. For sequential programs a single node on a single cluster was used; for parallel programs multiple nodes on a single cluster were used. In all cases, the Gnu C++ compiler version 2.95.2 was used as backend.

As reference, the original Fortran/MPI version, and a HPF version were used. Both were compiled with the Portland Group HPF compiler version 3.1, using the same MPI library as the Spar version.

For comparison with the sequential results in the previous section, and for comparison between the implementations available on this platform, the sequential execution times for HPF, Fortran 77, Spar, and Java were measured, see Fig. 2. For the overlapping cases, the results are comparable to the results of the previous section. The parallel Spar/Java version contains an extra matrix transposition that is essential for parallel execution, but slows down sequential execution.

For parallel execution the results are as shown in Fig. 3. In most cases the Spar results compare favorably with those for Fortran/MPI and HPF. For fast runs secondary effects become significant, such as initialization and setup times.

9. SPAR AND JVM

Since we extend Java, and since the Java Virtual Machine uses an instruction set that has specifically been designed for Java, in principle the language extensions of Spar require the JVM instruction set to be extended. However, since extending the instruction set would cause problems with existing JVM implementations there is a great reluctance in the Java community to any extensions. Moreover, only a limited number of bytecodes is available for extensions, which adds to the reluctance to use any of the remaining bytecodes. Together these arguments make it unrealistic to assume that we can add bytecodes, and it is therefore useful to consider alternative solutions.

The simplest solution is not to use the JVM at all, but to use a static compiler that directly generates machine code. A disadvantage of this solution is that we loose the ability to freely exchange compiled code between different platforms. We consider this disadvantage less significant for scientific programs, since in this community it is reasonable to assume that a Spar/Java compiler for the each specific target platform is easily available.

Another restriction of our reference Spar/Java compiler is that it does not support dynamic class loading, but this restriction could be lifted without using JVM bytecode. The main reason for the restriction is that the compiler does not support separate compilation. If available, the native support for dynamic class loading that is available on many platforms could be used.

Although we do not advocate this approach, most constructs in our set of language extensions could be represented in JVM bytecode, although with some restrictions and other disadvantages. Philippsen and Günthner [17] describe a scheme to translate all uses of complex numbers to traditional Java, and hence to bytecode. In their approach complex numbers are split into separate variables for the real and imaginary part. This leaves two problematic cases: complex numbers as return values of functions, and arrays of complex numbers. Both require more work, but are also handled. The approach could be generalized to support tuples.

Uses of multi-dimensional arrays could be translated to use a set of pre-defined classes. See §10 for a discussion of these sets.

Parameterized types as described in this paper cannot be translated directly to bytecode. They would require either an extension of the JVM bytecode, or the generation of a separate file for each instantiation of the parameterized class. Since the number of possible instantiations is not bounded (if nothing else, the number of parameters is unbounded), instantiations would have to be created dynamically, which is not an attractive solution.

10. RELATED WORK

Many of the new language constructs in Spar/Java resemble constructs in existing languages, although we are not aware of a single language that provides all of them. The most obvious relations are with Fortran and HPF (multi-dimensional arrays, complex numbers, and data distributions), with functional languages such as Haskell and ML (tuples), and with C++ (parameterized types). Parameterized types were also inspired by one of our previous projects, Tm [20, 19]. Also, we have been careful to adhere to existing

Java language design principles as much as possible for our language extensions.

In the Ninja project [1, 3] a compiler has been developed for pure Java. To provide support for array operations, a set of 'special' classes is defined that represent multi-dimensional arrays and complex numbers. These classes can be handled by all standard Java compilers, but the Ninja compiler recognizes these special classes, and generates efficient code for them. However, since access to multi-dimensional arrays is quite awkward, they are advocating language extensions for at least multi-dimensional array access. Based on this work, a proposal has been made through the Java Community Process to add multi-dimensional arrays to Java [12].

Philippsen and Günthner [17] propose to add a `complex` type to Java in a way that is very similar to ours. Instead of our imaginary floating point suffix 'i' they use a new keyword 'I' that represents $\sqrt{-1}$. Our approach is more compact and does not require a new keyword.

A number of Java packages for linear algebra have been proposed, see for example JAMA [11]. These packages often also introduce multi-dimensional arrays, but usually only in a restricted form. For example, JAMA introduces two-dimensional arrays (matrices) of `double`.

These proposals have as drawback that they impose restrictions on the element type and rank of the supported arrays. Moreover, the notation of array types and array access is quite awkward. For example, the Java Community proposal for multi-dimensional arrays contains the following example function:

```
void matmul( doubleArray2D a, doubleArray2D b,
  doubleArray2D c)
{
  int m = a.size(0);
  int n = a.size(1);
  int p = b.size(1);

  for(int i=0; i<m; i++){
    for(int j=0; j<p; j++){
      c.set(i,j,0);
      for(int k=0; k<m; k++)
        c.set(i,j,c.get(i,j)+
          a.get(i,k)*b.get(k,j));
    }
  }
}
```

Compared to this the Spar version is more compact and readable:

```
void matmul(double a[*,*], double b[*,*],
  double c[*,*])
{
  int m = a.GetSize(0);
  int n = a.GetSize(1);
  int p = b.GetSize(1);

  for(i :- 0:m,j :- 0:p){
    c[i,j] = 0;
    for(k :- 0:m)
      c[i,j] += a[i,k]*b[k,j];
  }
}
```

Titanium [30] resembles Spar/Java in the sense that it also provides a set of language extensions to develop Java into a language for scientific computations. It provides support for multi-dimensional arrays similar to Spar. Although is provides a `foreach` statement similar to that in Spar, this statement is not intended for parallelization. Instead, they provide explicit process-based parallelization with virtual shared memory.

See §4.1 for a discussion of related work on parameterized types.

11. CONCLUSIONS

In this paper we have shown Spar, a set of language extensions that improve the expressiveness of Java for scientific computations. They allow operations on multi-dimensional arrays to be expressed more naturally, and to be implemented more efficiently, including parallel operations. The language extensions also allow the construction of specialized array representations, such as symmetric, block, and sparse matrices.

Language extensions have been designed to be useful independent of their original purpose. In particular parameterized classes an tuples are generally useful, even beyond scientific computation.

12. FUTURE WORK

As stated in the introduction, we intent to provide a programming language that is comparable to Fortran in expressiveness for scientific computations. When we compare our current language with Fortran, we think that only one significant feature of Fortran 90 still is missing: array statements. This construct allows manipulations on entire arrays to be expressed as single statements, by selecting and manipulating ranges of array elements. For example, the following statement shifts the elements of an array left (we assume array `a` has `n` elements, also note that Fortran arrays start at element 1, so in this example element `a[n]` is the last element in the array):

```
a[1:n-1] = a[2:n];
```

A nice side-effect is that the compiler can often easily discover that no array bounds are violated, and eliminate bounds checking.

We would like to support array ranges of this form, but this will require some language design work.

Both constructs improve the expressiveness of the language, and help the compiler in generating more efficient code.

13. ACKNOWLEDGEMENTS

We would like to thank the participants of Dagstuhl Seminar 00451, "Effective Implementation of Object-Oriented Programming Languages", for their constructive and inspiring remarks on an early version of this paper.

14. REFERENCES

[1] Ninja web site. www.research.ibm.com/ninja.

[2] Class `com.ismil.math.complex`. webpage. www.vni.com/corner/garage/grande/complex.htm.

[3] P. V. Artigas, M. Gupta, S. P. Midkiff, and J. E. Moreira. High performance numerical computing in Java: Language and compiler issues. In *Proceedings of the 12'th Workshop on Language and Compilers for Parallel Computers*, Aug. 1999.

[4] H. Bal et al. The distributed ASCI supercomputer project. *ACM SIG, Operating System Review*, 34(4):76–96, October 2000.

[5] G. Bracha. JSR 000014 – add generic types to the Java programming language. webpage, 2000. java.sun.com/-aboutJava/communityprocess/jsr/jsr_014_gener.html.

[6] G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. GJ specification. Technical report, Bell Labs, May 1998. www.cs.bell-labs.com/who/wadler/-pizza/gj/Documents/index.html.

[7] DAS website. www.asci.tudelft.nl/das/das.shtml.

[8] P. Dechering, J. Trescher, J. d. Vreught, and H. Sips. V-cal: a calculus for the compilation of data parallel languages. In C.-H. H. et. al., editor, *8th Intl. Workshop, Languages and Compilers for Parallel Computing*, number 1033 in LNCS, pages 388–395, Columbus, Ohio, USA, Aug. 1995. Springer Verlag.

[9] M. Frumkin, H. Jin, and J. Yan. Implementation of NAS parallel benchmarks in high performance fortran. In *IPPS*, 1999.

[10] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification Second Edition*. The Java Series. Addison-Wesley, Reading, Massachusetts, June 2000.

[11] JAMA: Java matrix package. website. math.nist.gov/javanumerics/jama.

[12] J. E. Moreira. JSR 000083 – Java multiarray package. webpage, 2000. java.sun.com/aboutJava/-communityprocess/jsr/jsr_083_multiarray.html.

[13] J. E. Moreira, S. P. Midkiff, M. Gupta, P. V. Artigas, M. Snir, and R. D. Lawrence. Java programming for high performance numerical computing. *IBM Systems Journal*, 39(1):21–56, 2000.

[14] A. C. Myers, J. A. Bank, and B. Liskov. Parameterized types for Java. In *Proceedings of the 24th AMD Symposium on Principles of Programming Languages*, pages 132–145, Jan. 1997.

[15] NAS parallel benchmarks website. www.nas.nasa.gov/Software/NPB.

[16] S. Niemeijer. Parallel expressiveness of the Spar programming language. PDS Technical Report PDS-2000-006, Delft University of Technology, May 2000. `www.pds.twi.tudelft.nl/reports/2000/PDS-2000-006`.

[17] M. Philippsen and E. Günthner. Complex numbers for Java. *Concurrency: Practice and Experience*, 12(6):477–491, May 2000.

[18] C. v. Reeuwijk. Timber download site. www.pds.twi.tudelft.nl/timber/downloading.html.

[19] C. v. Reeuwijk. Tm website. www.pds.twi.tudelft.nl/~reeuwijk/software/Tm.

[20] C. v. Reeuwijk. Tm: a code generator for recursive data structures. *Software – Practice and Experience*, 22(10):899–908, October 1992.

[21] C. v. Reeuwijk. Spar 1.2 language specification. PDS Technical Report PDS-2000-008, Delft University of Technology, Nov. 2000. www.pds.twi.tudelft.nl/reports/2000/PDS-2000-008.

[22] C. v. Reeuwijk, A. v. Gemund, and H. Sips. Spar: A programming language for semi-automatic compilation of parallel programs. *Concurrency – Practice and Experience*, 11(9):1193–1205, Nov. 1997.

[23] C. v. Reeuwijk, F. Kuijlman, H. Sips, and S. Niemeijer. Data-parallel programming in Spar/Java. PDS Technical Report PDS-2000-005, Delft University of Technology, May 2000. www.pds.twi.tudelft.nl/reports/2000/PDS-2000-005.

[24] C. v. Reeuwijk, F. Kuijlman, H. Sips, and S. Niemeijer. Data-parallel programming in Spar/Java. In *Proceedings of the Second Annual Workshop on Java for High-Performance Computing*, pages 51–66, May 2000.

[25] T. Rühl, H. Bal, R. Bhoudjang, K. Langendoen, and G. Benson. Experience with a portability layer for implementing parallel programming systems. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 1477–1488, Aug. 1996.

[26] T. Rühl and R. Bhoudjang. The Panda 4.0 interface document. Technical report, Vrije Universiteit, Amsterdam, Jan. 1999.

[27] H. Sowizral, K. Rushforth, and M. Deering. *The Java 3D API Specification*. The Java Series. Addison-Wesley, 1995.

[28] K. Thorup. Genericity in Java with virtual types. In *European Conference on Object-Oriented Programming, LNCS 1241*, pages 444–471. Springer Verlag, 1997.

[29] G. Toth. The LASY preprocessor and its application to general multi-dimensional codes. *Journal of Computational Physics*, pages 981–990, 1997.

[30] K. Yelick, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella, and A. Aiken. Titanium: a high-performance Java dialect. In *ACM Workshop on Java for High-Performance Network Computing*, pages 1–13, Feb. 1998.