Annotating Spar/Java for data-parallel programming*

C. van Reeuwijk, F. Kuijlman, H.J. Sips, and S.V. Niemeijer

Parallel and Distributed Systems group, Delft University of Technology
C.vanReeuwijk@cs.tudelft.nl
url: `www.pds.twi.tudelft.nl/timber/spar`

Keywords: data parallelism, Java, high-performance computations, compilers

Abstract

Java is increasingly popular as a programming language for high-performance computation. This is because it is a modern, accessible language that is expressive enough for large classes of applications. However, since Java was originally not designed for high-performance computation, it is not surprising that some significant improvements in expressiveness and efficiency can be gained by augmenting the language with constructs that address the needs of high-performance programs.

We have designed a such set of language constructs. It contains support for array operations, complex numbers, annotations, and parallelization. Together, these extensions are called Spar. *In this paper we will describe the support in Spar for parallel programming, in particular data-parallel programming.*

An important part of our support for parallel programming is a mechanism to annotate data, statements and expressions with placement directives. These annotations help the compiler generate efficient parallel code.

In this paper we will describe the Spar support for parallel programming. We also present benchmark results from our Spar/Java compiler.

1 Introduction

Java is increasingly popular as a programming language for high-performance computation. This is because it is a modern, accessible language that is expressive enough for large classes of applications. Nevertheless, Java was originally not designed for high-performance computation. It is therefore not surprising that some significant improvements in expressiveness and efficiency can be gained by augmenting the language with constructs that address the needs of high-performance programs.

At Delft University of Technology, we have developed a set of language extensions for Java to improve the expressiveness for high-performance programs, and to easily allow compilation to highly efficient code. For this purpose we have augmented Java with a set of language constructs for multi-dimensional arrays; a 'toolkit' to build specialized array representations such as block, symmetric, or sparse arrays; annotations; and parallelization. Together, these extensions are called *Spar*.

The Spar/Java language is described in detail elsewhere [16, 14]; in this paper only those components are described that are relevant for data-parallel programming and for the presented performance results: multidimensional arrays, complex numbers, constructs for structured parallelization, and annotations.

An important part of our support for parallel programming is a mechanism to annotate data, statements, and expressions with placement directives. These annotations help the compiler generate efficient parallel code.

In its simplest form, parallelization can be achieved by using an explicit parallel programming interface, such as provided by libraries like MPI [20], or Java threads [9]. Unfortunately, programming with this model requires a detailed understanding of the complex interplay between program and machine. Using this programming model is therefore often extremely laborious and error-prone. Moreover, the program is often not portable. Instead, we follow a more implicit approach, where parallelism is generated by the

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

compiler, guided by user annotations in the program. The necessary parallelization constructs for this programming model are described in Section 2.3.

In recent years, embedded systems with multiple processors have become increasingly important. These systems often consist of a general-purpose processor and one, or even several, digital signal processors (DSPs). For portability, flexibility, and robustness it is often useful to regard such a configuration as a single heterogeneous parallel system. It is possible to generalize some of the programming models that are used for homogeneous parallel systems to heterogeneous systems. To support embedded systems, Spar provides support for heterogeneous processor configurations. This is reflected in the annotations for distribution described in Section 4.

In Section 2 the Spar language constructs relevant to this paper are described. In Section 3 the annotation language is described; in Section 4 the parallelization constructs are discussed. In Section 6 the parallelization constructs are compared with those of HPF. The current compiler is described in Section 8, and in Section 9 the initial results of the compiler are shown.

2 The Spar language extensions

2.1 Multidimensional Arrays

Spar provides multidimensional arrays as a straightforward generalization of Java arrays. For example:

```
int[*,*] a = new int[5,5];
a[0,0] = 42;
```

creates a twodimensional array, and assigns 42 to element `[0,0]` of the array.

2.2 Complex numbers

Spar/Java supports complex numbers as a new primitive type `complex`. A floating point literal with the suffix `i` represents an imaginary number. For example:

```
complex c = 2+3i;
double re = (double) c;
double im = Complex.imag(c);
complex cc = re + -1i*im;
```

2.3 Parallelization constructs

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.

The safest form of parallelization would be to have no special parallelization constructs at all, but to leave parallelization entirely to the compiler. Unfortunately, this is currently only feasible for limited classes of programs.

At the other extreme, the user can write explicitly parallel programs using communication libraries such as PVM or MPI, and perhaps explicit threading such as provided by the Java `Thread` class. However, this method is difficult, error-prone, and often results in programs that are not portable.

Since both these extremes are undesirable, Spar provides parallelization constructs that are in between: the user must provide *some* additional information to the compiler, but the generation of explicitly parallel code is left to the compiler.

In a sequential program the execution order of all statements is specified exactly. This is often an over-specification: the programmer may not care about the execution order of statements, even if the observable results differ. For example, the function

```
int search( int b[], int v ){
  for( int i=0; i<b.length; i++ )
    if( b[i] == v ) return i;
  return -1;
}
```

returns the index of the *first* element of `b` with the value `v`. If the user does not explicitly want the index of the *first* matching element of the array returned, but of *an arbitrary* matching element, the program is *overspecified*. Since this form of overspecification often prevents parallelization of the program, Spar provides the `each` and `foreach` commands. Given an `each` statement such as:

```
each { s1; s2; }
```

the compiler may choose one of the execution orders `s1; s2;` or `s2; s1;`. Once the execution of a statement has been started, it must be completed before another statement can be executed.

The `foreach` statement is a parameterized version of the `each` statement. For example:

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

Similar to the `each` statement, the compiler may choose any order for the execution of the iterations.

The `each` and `foreach` statements can only be executed in parallel when there is no observable interference between statements or iterations. To discover this, the compiler must do some data-dependency analysis (although less extensive than for sequential loops), or the user must inform the compiler with an annotation.

3 The annotation language

The Spar/Java compiler provides a general mechanism to annotate a program. Each annotation consists of a list of *pragmas*. These pragmas allow the user to give the compiler further information about the program, and give hints for efficient compilation. The user may assume that a pragma does not influence the behavior of a program[1]; it only improves the efficiency of the program (in terms of execution time, memory use, or any other measure of efficiency).

For example, the annotation:

```
<$ independent,
   boundscheck=false, iterations=42 $>
```

consists of three pragmas. The first one, `independent`, does not have a value, and is called a *flag pragma*. The other two, `boundscheck` and `iterations`, are called *value pragmas*.

Identifiers in the pragma name and pragma value are completely independent of those in Spar/Java[2]. It *is* possible, however, to refer to variables in the host language by prefixing a name with a '`@`'. For example, in the code fragment

```
foreach( i :- 0:n ) <$ iterations=@n $> {
  sum += i;
}
```

the value of the `iterations` pragma contains a reference to the variable `n`.

Instead of a single value, a pragma may have a *list* as value. Such a list is written as a sequence of expressions surrounded by brackets, similar to lists in the programming language Lisp. Since the lists can be nested to an arbitrary depth, this allows pragmas to have expressions of arbitrary complexity as value. For example:

```
<$cost=(lambda (i j)
  (sum (prod 5 i i j) (prod i j) 42)$>
```

As a service to the user, the Spar/Java frontend allows a number of binary operators to be used in pragma expressions. The traditional precedence rules on these operators are obeyed. Expressions with binary operators are immediately translated to expression lists. For example, the pragma expression `3*n*n+5*n+1` is translated to '`(sum (prod 3 n n) (prod 5 n) 1)`'.

[1]Since the `each` and `foreach` statements are non-deterministic, we consider a program to have the same behavior for all possible execution orders of these statements. Therefore, an annotation may cause another execution order of such a statement.

[2]Without this strict separation of namespaces, compiler engines using these pragmas would have to know too much about the translation of Spar/Java expressions to their own level.

As further 'syntactic sugar', the Spar/Java frontend allows subscript-like expressions. Such an expression is immediately translated to a list starting with the identifier '`at`'. For example, the pragma expression '`p[1,a]`' is translated to '`(at p 1 a)`'.

The following language constructs can be annotated:

- the entire program
- statements
- expressions
- types
- declarations
- formal parameters

4 Pragmas for parallelization

To help the compiler with the parallelization of a program, the user may annotate a program with pragmas to specify the placement of data, or the place where a block of code is executed. For this purpose the parallelization engines support the following annotations:

The `ProcessorType` pragma is used to declare names of processor types and to associate them with processors characteristics (e.g. alignment of data structures and endianness of primitive types etc.) and capabilities (e.g. whether it contains a FPU etc.). `ProcessorType` pragmas must be global. For example:

```
<$ ProcessorType=((Gpp "Pentium2")
   (Dsp "Trimedia")) $>
```

The strings `"Pentium2"` and `"Trimedia"` refer to processor descriptions that are known to the compiler, for example through configuration files.

A `Processors` pragma is used to name the processors in a system, and describe their arrangement to the compiler. The `Processors` pragma must be a global pragma.

Either a single processor can be declared, or an array of processors. The processor array can have any number of dimensions. For example:

```
<$ Processors=((Gpp gpp1)
   (Dsp dsp1D[4]) (Dsp dsp2D[2,3])) $>
```

The system specified above consists of a single processor `gpp1` of type `Gpp`, a one-dimensional processor array `dsp1D`, and a two-dimensional array `dsp2D`, both of type `Dsp`. Each modeled processor corresponds to a single physical processor.

A Spar/Java program can be annotated at specific points with an `on` pragma that allows users to place data and work on specific processors. Pragmas can annotate expressions, statements, and member functions.

```
<$ on=Gpp $>
<$ on=Dsp[0] $>
```

Pragmas may be nested; an `on` pragma on an inner block or expression overrules the enclosing specification.

The special `on` value `_all` is used to denote all processors; the value '`_`' (a single underscore) is used to denote an unspecified placement ('don't care'). For example:

```
<$ on=_all $>
<$ on=_ $>
<$ on=Dsp[_all] $>
<$ on=Dsp[_] $>
```

It is also allowed to use two special functions as `on` expressions: The `(block a*i+b m)` placement function places index i onto processor $p = (a \cdot i + b)/m$. The value of p is bounded by the index range allowed in the corresponding dimension of the processor type. If no `m` is specified, the value is derived from the context: if there are N elements in the corresponding array dimension, or if there are N iterations in the corresponding iteration range, $m = N/P_{ext}$ is assumed. The `(cyclic a*i+b m)` placement function places index i onto processor $p = ((a \cdot i + b)/m) \bmod P_{ext}$. If no m is specified, the value 1 is assumed.

For data that is distributed with the `block` and `cyclic` functions, the compiler is able to generate highly efficient code for the enumeration of local elements and the translation of global to local array indices; see [15] for details. With these functions, all the data mappings of High-Performance Fortran (HPF) 2.0 [12] can be specified. See Section 6 for more details.

4.1 Annotating declarations

By annotating a member function, the user can specify the group of processors allowed to execute the member function. For example:

```
<$ on dsp1D[_all] $>
int myfunc( int i, String s ) {
    return i+1;
}
```

4.2 Annotating statements

A statement annotated with an `on` pragma will be executed only on the specified processor(s). Arbitrary statements may be annotated, but in practice the annotation is mainly interesting for code *blocks*. For example:

```
foreach( i :- 0:100 )
<$ on=dsp1D[(block @i 25)] $> {
    a[i] = a[i] + 1;
}
```

The assignment of `a[i]` is executed on processor `dsp1[i/25]`. In other words, the iteration space is divided into blocks of size 25, and each block is executed on a different processor.

4.3 Annotating expressions

In principle, any expression can be annotated with an `on` pragma. The `new` expression is a special case, since this not only specifies the placement of the constructor execution (if not overridden by an annotation on the constructor), but also the placement of the newly constructed class or array instance. For example, the pragma:

```
String a = <$ on=gpp1 $> new String();
```

specifies that a new `String` must be constructed on `gpp1`.

In the case of array `new` expressions, a slightly extended version of the `on` pragma is allowed. For example, the pragma:

```
int[*,*] b =
  <$ on=(lambda (i j)
     dsp2D[(block j 5),_all]) $>
  new int[50,50];
```

specifies that every array element `b[i,j]` is constructed on processors `dsp2D[(block j 5),_all]`. The `_all` expression in the second dimension means that the elements are *replicated* in the second dimension of the processor array.

Note that the formal parameter `i` is not used in the distribution expression. Therefore, the first dimension of the array does not influence the distribution of an element.

5 Other annotations

The Spar programs that we discuss in Section 9 use two other annotations. These annotations could often be generated by analysis engines, but implementation of such engines is currently beyond the scope of our work.

5.1 The `reduction` annotation

The `foreach` statements allows reduction operations such as

```
int sum = 0;
foreach( i :- 0:a.length )
  <$ reduction $> sum += a[i];
```

If the array that is being reduced is `block` or `cyclic` distributed, and if the loop is annotated as a reduction as shown above, the parallelization engines are able to generate efficient parallel code for such a reduction.

5.2 the `independent` annotation

As mentioned in Section 2.3, the `each` and `foreach` statements can only be executed in parallel when there is no observable interference between statements or iterations. Since at the moment we do not have analysis engines to discover this, we annotate such loops with the `independent` annotation.

6 Comparison with HPF

Since Spar allows data-parallel programming in a way that is similar to that of HPF [12], it is useful to make a direct comparison between the two programming languages.

Spar supports the same class of distributions as HPF 2.0, but since its distribution annotations are more expressive than the `distribute` directive of HPF, it does not need (or support) templates. The absence of templates, and alignment in general, makes some distributions somewhat less convenient to express, but we think that this disadvantage is of limited significance. Moreover, by leaving out templates we avoid the introduction of a cumbersome and ill-fitting language construct.

Spar currently supports exactly one processor array declaration. In contrast, HPF supports an arbitrary number of declarations. However, the correspondence between processors in the different processor arrays in HPF is not fully specified. In practice this makes it difficult to write efficient programs that 'mix' processor arrays. Although not shown in this paper, the Spar annotation mechanism can easily be extended to allow 'views' on the processor array, so that, for example, a 4×4 processor array can also be seen as a one-dimensional array of 16 processors. This would provide similar functionality to the multiple processor arrays of HPF, but without the associated correspondence problems. We plan to incorporate this feature in a future version of the compiler.

The `forall` loop construct of HPF requires (at least conceptually) that copies of the arrays used in the loop body are made. However, in Java and Spar arrays are accessed through *references*, and several references can potentially point to the same array. Therefore, it is very difficult to define a loop construct similar to the HPF `forall` that still has easily understandable behavior, and that can be implemented efficiently[3].

Instead, Spar defines the `foreach` statement, which *can* be implemented efficiently. Despite its non-deterministic semantics, we believe it is has easily understandable behavior.

Also, the body of a HPF `forall` may only contain assignments to array elements, whereas the `foreach` statement allows arbitrary loop bodies. In particular, the `foreach` allows reduction operations, see Section 5.1. Some reduction operations can be expressed in HPF with intrinsic functions such as `SUM`, but this is a much more limited feature.

7 The targeted hardware

The compiler allows a set of processors of any size, with any number of processor types. Each processor should be capable of running compiled C++ code[4]. Each processor may have local memory, may share memory with other processors, or both. The system may contain groups of identical processors; the annotations allow each group to be arranged into a one- or more-dimensional processor array.

For the moment we assume all processors can communicate with all other processors with the same efficiency. Without this assumption the placement and scheduling of tasks would be much more complicated, because the varying communication costs would have to be taken into account.

8 The compiler

The compiler consists of a frontend that generates code in the intermediate language Vnus [7], a number of parallelization engines that work on Vnus, and a backend that converts from Vnus to C++.

The frontend generates a single Vnus file containing all relevant code. To reduce code size, the frontend tries to avoid compiling classes and methods that are not used. We do not support bytecode or dynamic class loading.

The current compiler implementation supports almost all of Java as described in the Java Language Specification [9]. The `Thread` class is not yet supported. Some other libraries (e.g. windowing libraries) are not supported due to the large implementation effort they would require. The compiler also supports most of the constructs that have been added for Java 1.1 and 1.2. From Spar, all the constructs described in this paper are supported. The restrictions and extensions are described in detail in [14].

Our implementation of Spar/Java is available for downloading under the Gnu Public license[5].

[3] For related reasons the array statements of Fortran 90 are not easily incorporated in Java or Spar.

[4] The generated code is mostly acceptable to a `C` compiler, but we use the C++ `throw` and `catch` statements.

[5] See `www.pds.twi.tudelft.nl/timber/spar`.

At the moment of writing, Version 1.1 of the Spar compiler is current.

The public version does not yet support parallelization, since we do not yet consider the parallelization engines robust enough for public release. We plan to improve the robustness in the near future, and release a version that at least supports the parallelization features described in this paper.

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 [18, 19].

9 Results

To measure the initial performance of the compiler, we have implemented two benchmarks from the NASA Numerical Aerospace Simulation group (NAS) parallel benchmark suite[6] Version 2.3. Guided by the HPF and Fortran 77 versions we had available, the algorithms were reimplemented in Spar/Java. All NAS benchmark programs contain a verification phase that ensures that the algorithm implementation generates the correct answers. The Spar/Java versions pass this verification phase, so we are confident that we implement the NAS benchmark algorithms correctly.

The programs were run on one cluster of the DAS[7] distributed supercomputer. Each node runs Linux, and contains a 200 Mhz Pentium Pro, 64 MB RAM, 2.5 GByte local disk, and a Myrinet interface.

At the moment, the compiler does not support garbage collection (explicit deletion is used). For the performance measurements null pointer checks are switched off. Preliminary measurements indicate that the overhead of null pointer checks can be reduced to less than 10% by eliminating redundant checks. Unfortunately, formal measurements have not yet been completed.

The compiler *does* perform bounds checking. Any access beyond the bounds of the array causes an exception. For the moment, no attempt is made to eliminate redundant bounds checks. Note that since Spar supports multi-dimensional arrays, the cost of this bounds checking is much less than it would be in Java, where a multi-dimensional array would have to be implemented with nested arrays, with their associated cost in in bounds checking and null pointer checking. This is particularly significant for the FT benchmark described below, since it uses three-dimensional arrays.

For the runs of the Spar compiler, the Gnu C++ compiler Version 2.8.1 was used as backend. The same compiler was used for the g++ reference runs.

Before we discuss the parallel performance results, we show some performance results for sequential programs. These results are intended as 'background information' for the parallel performance results; we plan to evaluate the sequential performance of the Spar/Java compiler in more detail in a separate paper.

9.1 Sequential performance of the compiler

To evaluate the sequential performance of the Spar compiler, the NAS FT benchmark was implemented in C, Java, and Spar, and the execution time was measured on a computer with an Intel Pentium II at 400MHz, 192MB main memory, and Windows NT[8]. This resulted in the execution times listed below. All times are elapsed times; the median value of 20 runs was taken. Startup time (including JIT compilation) was not measured. The C compiler (gcc and g++) is the egcs compiler that is part of CygWin 20.2.

In all cases NAS FT dataset 'W' was executed: a 3D Fast Fourier Transform on a $128 \times 128 \times 32$ array.

Compiler	Language	time [s]
'java' from JDK 1.2.2	Java	161.80
Spar with g++	Java	15.43
javac with Symantec JIT	Java	13.69
g++	C++	12.46
Spar with g++	Spar	11.21
gcc	C	6.98

In the Spar version, type `complex` and threedimensional arrays were used. In the Java version nested arrays were used, and complex numbers were represented by pairs of `double`s, stored in adjacent elements of the array. Apart from these aspects, the Spar and Java versions are the same. Since the representation of complex numbers is not likely to impact the execution times significantly, the slower execution times for the Java version (24% for the Spar/Java compiler, 22%

[6] See `www.nas.nasa.gov/Software/NPB`.

[7] See `www.asci.tudelft.nl/das/das.shtml`.

[8] We used Windows NT for the sequential measurements, since most of the fast JVM implementations are not available for Linux. Unfortunately, this makes the sequential performance results more difficult to compare with the parallel results.

for the Symantec JIT compiler) are mainly attributable to the use of nested arrays instead of true 3-dimensional arrays.

The reason that the standard JVM implementation of the Sun JDK shows such a poor performance (more than 10 times slower than the other versions) is of course that pure interpretation is used.

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++.

9.2 NAS EP benchmark

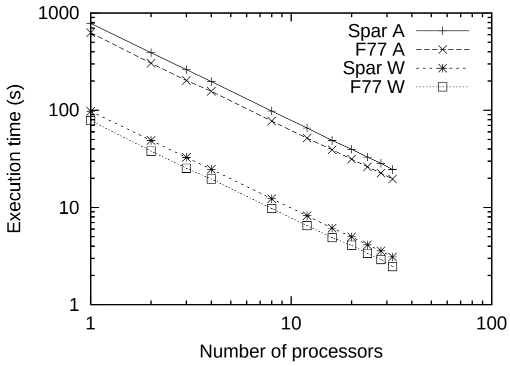
The NAS EP benchmark approximates π by choosing random points in a square, and counting the percentage that falls in the inscribed circle of the square.

To parallelize the code, the Spar implementation divides the calculations in batches. The results for each batch (the percentage of 'hits') is stored in an element of an array; the global result can then be calculated with a reduction operation on this array. The calculations for each batch, and the results array, are distributed in a *cyclic* fashion over the processors.

The Spar version contains the following annotations: a distribution annotation for the results array; a distribution annotation for the loop that iterates over the batches; and an annotation to label the result reduction loops as reductions. From these annotations only, the compiler is able to generate a parallel program, with the results shown below.

As reference, the Fortran 77 version from the NAS 2.3 package was used. To parallelize, it uses the same strategy as the Spar implementation, but with explicit communication. This implementation was compiled with Gnu Fortran version 1.0.3, and used MPI. Unfortunately, we did not have a HPF version available. For Spar the communication library Panda [18] was used. The MPI library we used for our measurements also uses of the Panda library.

Both implementations were run for the 'W' dataset (2^{25} samples) and the 'A' dataset (2^{28} samples).



proc.	Spar W	f77 W	Spar A	f77 A
1	102.45	78.50	818.23	628.22
2	50.92	38.00	407.05	304.28
3	34.25	25.38	273.75	202.75
4	25.57	19.63	204.13	156.71
8	12.79	9.76	102.06	77.32
12	8.62	6.50	69.55	51.65
16	6.58	4.90	53.84	39.18
20	5.18	4.09	41.40	31.38
24	4.32	3.37	35.80	26.18
32	3.23	2.46	26.34	19.60

As can be seen from these results, the Spar program scales just as well as the version using Fortran 77 and MPI, but is somewhat (30%) slower.

9.3 NAS FT benchmark

The NAS FT benchmark performs a 3D Fast Fourier Transform (FFT).

The Spar version distributes the FFT array in *block* fashion in one dimension. Thus, if a $64 \times 64 \times 64$ array is distributed over 2 processors, each processor gets a $64 \times 64 \times 32$ processors slice of the array. For 4 processors the slice is $64 \times 64 \times 16$, and so on. In the NAS benchmark, The 3D FFT is implemented by doing a 1D FFT on all one-dimensional array sections parallel to the x axis, then on all one-dimensional array sections parallel to y axis and then the z axis.

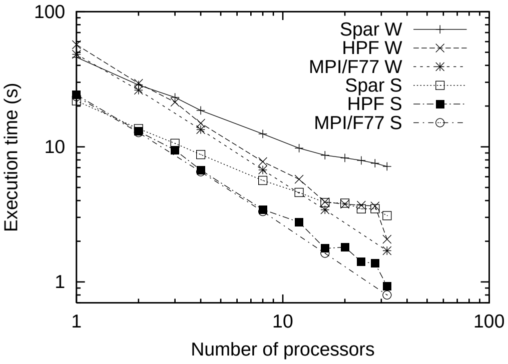
The Spar version always performs 1D FFT transforms on array sections parallel to the x axis. For the last two phases, it copies and permutes the elements in the original array to a suitable temporary array, performs the 1D transform, and copies the elements back again. Since we distribute the temporary array in a different dimension, all elements for the 1D FFT transform on one array section are always locally available, which allows it to be implemented very efficiently.

The Spar version contains the following annotations: distribution annotations for the distributed arrays, including the temporary array mentioned above; a distribution for the iteration over the 1D FFT transform; an annotation to label the loop that calculates the verification checksum as a reduction, and a number of annotations to label loops as 'independent'. Using these annotations only, the compiler generates a parallel program, with the results shown below.

As reference, a HPF version and a Fortran 77 with explicit MPI communication were used. Both were compiled with the Portland Group F90/HPF compiler Version 3.1, using MPI. Again, for Spar the underlying Panda library was used.

Both the HPF and the F77/MPI version use a similar strategy for parallelization as the Spar version, but in the case of F77/MPI explicit communication is of course necessary.

All implementations were run for the 'S' dataset ($64 \times 64 \times 64$ elements) and the 'W' dataset ($128 \times 128 \times 32$ elements). Due to its method of implementation, the Fortran 77 version with explicit MPI communication could only be run for a number of processors that is a power of 2.



proc.	f77 S	Spar S	HPF S	f77 W	Spar W	HPF W
1	23.50	22.12	24.24	48.30	42.92	56.90
2	12.73	12.91	13.01	26.22	26.49	29.43
3		10.23	9.46		20.49	21.40
4	6.53	8.34	6.72	13.38	17.17	14.93
8	3.32	4.91	3.43	6.75	11.00	7.76
12		3.79	2.77		9.04	5.75
16	1.63	2.93	1.77	3.42	7.17	3.91
20		2.96	1.81		6.51	3.76
24		2.54	1.41		6.13	3.69
32	0.80	2.11	0.93	1.70	5.35	2.07

For comparison, when the programs Spar and HPF versions are compiled for sequential execution, the execution times are as follows:

Spar S	HPF S	Spar W	HPF W
17.3	23.33	36.3	48.08

For 1 and 2 processors, Spar is faster than both other implementations, presumably because it generates more efficient code for 'send to self' messages. Beyond that, Spar incurs overhead that gradually increases compared to the other implementations. We are aware of one contributing factor: in several communication layers Spar does more dynamic memory allocation than is strictly necessary.

However, even the current results show that the performance and scalability of the Spar version matches the HPF and even the F77/MPI version quite closely.

10 Related work

Although several attempts have been made to define a proper model for capturing semi-implicit parallelism in languages [6, 8, 10, 17], success has been limited due to limitations in expressibility, parallel constructs being too explicit, limitations in automatic analysis [11], or incorporation into languages such as Fortran which are less suited to embedded systems programming. The most successful of these attempts is HPF, which has been discussed in Section 6.

For Java, Spar is not the only proposal to extend the language for high-performance computing. HPJava [5, 22] is a proposal to add multi-dimensional arrays and data-parallel programming to Java. In contrast to Spar, the programs are explicitly data-parallel. That is, any data transfer between processors has to be explicitly done in the program by a call to the HPJava communication library.

Blount, Chatterjee, and Philippsen [4] describe a compiler that extends Java with a `forall` statement. To execute the `forall` statement, the compiler spawns a Java thread on each processor, and the iterations are evenly distributed over these threads. Synchronization between iterations is done by the user using the standard Java synchronization mechanism. No explicit communication is performed; a shared-memory system is assumed. Due to the dynamic nature of the implementation, they can easily handle irregular data and nested parallelism. The paper also mentions a library of collection classes, but few details are given.

Titanium [21] provides vectors, multidimensional arrays, iteration ranges, and a `foreach` statement comparable to those in Spar. In most cases the Spar version of these constructs is more general. They explicitly state that their `foreach` is not intended for parallelization. Titanium supports iterations over arbitrary sets; moreover these iteration ranges are 'first-class citizens'; they can be handled and modified independent of any loop statements. Unsurprisingly, support for arbitrary iteration sets leads to inefficiencies. Therefore, Titanium provides a separate representation for rectangular iteration. In contrast, Spar supports 'classical' iteration sets that can be expressed as nested iteration ranges with strides. Spar's iteration ranges are more general than the rectangular iteration sets of Titanium, and can be implemented just as efficiently.

In the Ninja project [2, 13] a compiler is developed for pure Java. To provide support for array operations, a set of 'special' classes is defined that represent multi-dimensional classes 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.

A number of Java packages for linear algebra have been proposed, see for example JAMA [1]. These packages often also introduce multi-dimensional arrays, but usually only in a limited form. For example, JAMA introduces two-dimensional arrays (matrices) of `double`.

In Javar [3], a parallel loop is identified with a special annotation. Since Javar annotations are represented by special comments, Javar programs are compatible with standard Java compilers. There are no provisions for multi-dimensional arrays or complex numbers.

11 Conclusions

This paper presents language extensions to Java, and a compiler, for data-parallel computation. The language extensions are part of a larger set to support high-performance computation in general [14].

The language extensions allow the user to write parallel programs in a portable and understandable manner. The placement annotations allow precise and flexible control of the placement of data and tasks on specific processors or sets of processors. This allows the development of programs for both traditional supercomputer systems, and embedded systems with a heterogeneous set of processors.

To evaluate the performance of the compiler, we have compared both sequential and parallel programs with implementations in other languages. The results for sequential programs are already sufficiently close to 'traditional' implementations to justify the use of Spar/Java as a programming language for high-performance computation.

The results for parallel programs are also encouraging, but it is clear that some refinements are still necessary. We believe that with some work Spar/Java will become an attractive programming language for high-performance parallel computation on both single-processor and multiprocessor systems.

The current translation scheme imposes some restrictions, however. Since message passing requires cooperation from both sides, only data-parallel and a somewhat restricted form of task-parallel programs are supported. For a more general approach a virtual shared-memory system or one-sided communication is required. Therefore, the current compiler only support data-parallel programming. This is a temporary restriction, however; we plan to generalize the parallelization engines in the near future.

12 Acknowledgements

The authors like to thank Sidney Cadot for his help in writing and executing the benchmarks in this paper. The authors like to thank Michael Frumkin from NASA for letting us use his HPF implementation of the NAS FT benchmark.

References

[1] JAMA: Java matrix package. WWW page. url: `gams.nist.gov/javanumerics/jama`.

[2] 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*, August 1999.

[3] Aart J.C. Bik and Dennis B. Gannon. Automatically exploiting implicit parallelism in Java. *Concurrency, Practice and Experience*, 9(6):579–619, June 1997.

[4] Brian Blount, Siddhartha Chatterjee, and Michael Philippsen. Irregular parallel algorithms in Java. In *Irregular'99: Sixth International Workshop on Solving Irregularly Structured Problems in Parallel*, April 1999.

[5] Bryan Carpenter, Yuh-Jye Chang, Geoffrey Fox, Donald Leskiw, and Xiaoming Li. Experiments with "HPJava". *Concurrency, Practice and Experience*, 9(6):633–648, June 1997.

[6] B. Chapman, P. Mehrotra, J. Van Rosendale, and H. Zima. A software architecture for multidisciplinary applications: Integrating task and data parallelism. In *Proc. Fifth Workshop on Compilers for Parallel Computers*, pages 454–466, Malaga, June 1995.

[7] P.F.G. Dechering, J.A. Trescher, J.P.M. de Vreught, and H.J. Sips. V-cal: a calculus for the compilation of data parallel languages. In C.-H. Huang et. al., editor,

8th Intl. Workship, Languages and Compilers for Parallel Computing, number 1033 in LNCS, pages 388–395, Columbus, Ohio, USA, August 1995. Springer Verlag.

[8] I. Foster. Task parallelism and high-performance languages. *IEEE Parallel and Distributed Technology*, pages 27–36, Fall 1994.

[9] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. The Java Series. Addison-Wesley, Reading, Massachusetts, August 1996.

[10] Thomas Gross, David R. O'Hallaron, and Jaspal Subhlok. Task parallelism in a high performance Fortran framework. *IEEE parallel and distributed technology*, pages 16–26, Fall 1994.

[11] M. R. Haghighat and C. D. Polychronopoulos. Symbolic analysis for parallelizing compilers. *ACM Transactions on Programming Languages*, pages 477–518, July 1996.

[12] High Performance Fortran Forum. *High Performance Fortran Language Specification*, 2.0 edition, February 1997.

[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] C. van Reeuwijk. Spar 1.2 language specification. PDS Technical Report PDS-2000-008, Delft University of Technology, November 2000. url: `www.pds.twi.tudelft.nl/-reports/2000/PDS-2000-008`.

[15] C. van Reeuwijk, W. Denissen, H.J. Sips, and E.M. Paalvast. An implementation framework for HPF distributed arrays on message-passing parallel computer systems. *IEEE Transactions on Parallel and Distributed Systems*, 7(9):897–914, September 1996.

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

[17] M.C. Rinard, D.J. Scales, and M.S. Lam. A high-level, machine independent language for parallel programming. *Computer*, pages 28–38, June 1993.

[18] Tim Rühl, Henri Bal, Raoul Bhoudjang, Koen Langendoen, and Gregory 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, August 1996.

[19] Tim Rühl and Raoul Bhoudjang. The Panda 4.0 interface document. Technical report, Vrije Universiteit, Amsterdam, January 1999.

[20] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI: The Complete Reference*. MIT Press, Cambridge, MA., 1996.

[21] Kathy Yelick, Geoff Pike, Carleton Miyamoto, Ben Liblit, Arvind Krishnamurthy, Paul Hilfinger, Susan Graham, David Gay, Phil Colella, and Alex Aiken. Titanium: a high-performance Java dialect. In *ACM Workshop on Java for High-Performance Network Computing*, pages 1–13, February 1998.

[22] Guansong Zhang, Bryan Carpenter, Geoffrey Fox, Xingying Li, and Yuhong Wen. Considerations in HPJava language design and implementation. In S.Chatterjee et. al., editor, *Proceedings of the 11th International Workshop on Languages and Compilers for Parallel Computing*, pages 18–33. Springer, August 1998.