Spar: a programming language for semi-automatic compilation of parallel programs

Kees van Reeuwijk, Arjan J.C. van Gemund, Henk J. Sips

August 4, 1997

Abstract

We present Spar, a programming language for semi-automatic parallel programming, in particular for the programming of array-based applications. The language has been designed as part of the Automap project, in which a compiler and run-time system are being developed for distributed-memory systems. As suggested by its name, Automap aims at completely automatic code and data mapping at either compile-time or run-time. This relieves the programmer of a difficult task, and makes Spar programs completely portable.

Apart from a few minor modifications, Spar is a superset of Java. This provides Spar with a modern, solid, language as basis, and makes Spar more accessible. Spar extends Java with constructs for parallel programming, extensive support for array manipulation, and a number of other powerful language extensions.

1 Introduction

Although parallel computers are well-established, they have never become very popular. An important reason for this is that they are considerably more difficult to program than sequential computers. Therefore, parallel computers are currently only used for applications where the large additional effort and expense can be justified. There is an important class of applications, however, where the use of a parallel computer could also be justified, if only programming such a computer would be simpler. This means that the programmer should be provided with a programming language and compiler that supports automatic or semi-automatic parallelization. This is likely to incur some cost, in the same way that high-level languages incur some cost compared to hand-written assembly programs, but this is often an acceptable tradeoff for the shorter program development time. An advantage of semi-automatic parallelization is that it is more likely to be portable, since the details of program optimization for a given machine can be handled by the compiler. Again, there is a clear analogy with high-level programming languages.

Obviously, fully automatic parallelization is the most convenient solution, so the sequential programming model would be the most attractive [1, 27]. However, the nature of the data dependencies associated with most sequential programs still prohibits fully automatic parallelization. Moreover, a sequential program is often overspecified, since dependencies can be created that are not essential to the problem.

At the other extreme, the explicit parallelism in languages like Occam [18], CC++ [5], and Java [13]; and communication libraries like MPI [21, 4], guarantees that the parallelism in the program is exposed, but there is a severe risk of introducing unintended non-determinism and non-portability, since the code and data mappings are specified by the user.

A compromise was found in data parallel languages [15, 16], where the user only has to provide mapping annotations, but in practice this still means that the user must have detailed understanding of the complex interplay between program and target machine. Although highly efficient code can be generated [7, 24], the limited expressiveness of the data parallel model has already led to a number of extensions to include more explicit parallelism [6, 14, 11].

Within the Automap project [25] another compromise is sought. Using parallel constructs, the user divides the program into a number of medium-grain tasks, and the compiler will automatically map the code and data onto a distributed-memory system. The program is still highly portable, yet

the performance of the resulting code will be high. To simplify the cost estimation and scheduling mechanisms required for automatic mapping, the Automap project currently restricts programs to a class conforming to the SPC [22] (Series-Parallel Contention) programming model. Clearly, there will be loss of parallelism due to this restriction, but this has been conjectured to be small. Compelling evidence in support of this claim is provided in [10]. Nevertheless, future versions of the Spar compiler may lift this restriction.

To support the SPC programming model, Spar provides the `foreach` loop construct, that specifies that the iteration order is immaterial. After some analysis, the compiler can transform most `foreach` loops into explicit parallel loops in SPC form. This intermediate form is then used for automatic task mapping. The same loop construct has been proposed by Wolfe [26].

The language constructs that we need for the Automap project could be embedded in many languages. Although we have considered 'established' languages such as Fortran, C, and C++, most of them would require too much implementation effort, or would not easily lend themselves to extension. An alternative, a newly defined language, would not be attractive to potential users, and would require considerable language design effort. We finally decided on Java as the core language, since it is relatively easy to implement and extend. The suitability of Java as a (core) language for scientific computation has received growing attention recently [12]. Our first intention was to use Java only as the starting point, to modify the language to support the constructs that we would need, and leave out the constructs that could not be supported. During the design it has become clear that our constructs could be added without changing Java, and that we could support almost all of Java. As a result, Spar is a superset of Java, apart from a few minor modifications. Interestingly, in [19] a similar history is sketched. For the moment, Spar does not support Java threads, since these do not fit in the SPC form that we currently support.

The parallel language extensions of Spar are determined by the needs of the Automap project, and by the desire to make the formulation of parallel programs as reliable as possible. The array manipulation constructs are inspired by Fortran 90 [17] and Booster [20]. Spar macros and parameterized classes (see Sections 2.7 and 2.4) are inspired by functional or other non-imperative languages, by the language theory of Boute [2] and by the FORFUN project [23]. Spar is not the first language to add parameterized types to Java; for example, in Pizza [19] parametric polymorphism, higher-order functions, and algebraic types are added to Java. Parametric types are also mentioned in [13], although no details are given.

The paper is organized as follows. In Section 2 we present the Spar programming language as far as the extensions to Java are concerned. Due to space limitations we can not include a program to demonstrate the language features. Instead, we demonstrate the use of all new language constructs in small program fragments. We briefly describe the compilation of Spar in Section 3. In Section 4 we draw some conclusions.

2 The Spar programming language

Since most of the language constructs are inherited from Java, only the extensions and the (rare) modifications are described.

Obviously, Spar provides extensions to support parallel programming, in the form of the `each` and `foreach` language constructs.

Since Spar is designed to support array-based computations, it has more extensive support for arrays than Java. Spar generalizes Java arrays to multi-dimensional arrays that can be grown and shrunk during their life-time. Array elements can be manipulated in the large with array statements.

Another important goal in the development of Spar was to provide support for sparse matrices. Since there are more sparse matrix representations that can be reasonably incorporated in any programming language, no attempt has been made to support sparse matrices directly. Instead, the language allows easy definition of new matrix representations. In particular, any class that implements the Array interface enjoys the same privileged notation as ordinary arrays.

2.1 Parallel programming

Even if we restrict ourselves to the SPC programming model, there is a range of possible language constructs to express parallelism. They mainly differ in the tradeoff between ease of use for the programmer and ease of use for the compiler.

At one extreme there are constructs with clear semantics and which are deterministic in the sense that parallel or sequential execution yields the same results. An example is the `forall` construct of HPF, which can have array statements in its loop body. In Dechering et al. [8] it is shown how this construct can be generalized to cover arbitrary loop bodies. The main problem in such a general construct is that it heavily relies on copy-in/copy-out semantics. In practice this sometimes leads to unexpected behavior.

At the other extreme there are explicitly parallel loops, e.g. HPF DO loops that are annotated as independent, or the `parfor` of CC++; which specify that the iterations can be executed in parallel with no consideration for the interactions. This does not require any analysis by the compiler, but the construct has inconvenient semantics for programmer, because parallel and sequential execution may yield surprisingly different results.

In Spar we provide the `foreach` construct, which specifies that the iterations can be executed in arbitrary order, but that each iteration is to be executed sequentially. This model is fairly easy to understand for the programmer, and requires less complicated analysis by the compiler than analyzing strict sequential `for` loops. In principle the semantics are the same for sequential or parallel execution. Moreover, reduction operations can be formulated quite elegantly, which is not possible with the `forall` or with explicitly parallel loops.

2.1.1 The `each` statement

Given a block such as:

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

The statements `s1` and `s2` are executed in arbitrary order. It is guaranteed, even for compound statements, that every statement is executed as one state transition.

Thus, the compiler will choose one of the execution orders "`s1; s2;`", or "`s2; s1;`", even if the statements are compound.

2.1.2 The `foreach` statement

The `foreach` statement is a parameterized version of the `each` statement of the previous section. For example,

```
foreach( i=0:n ){
    a[i].init();
}
```

invokes the init method of n members of array `a`. As for the `each` statement, it is guaranteed that every iteration instance of a `foreach` statement is executed as one state transition. Thus, an iteration can only influences other iterations when it has been completed.

To allow easier analysis, the `foreach` has a range syntax rather than the traditional `while`-like syntax of the `for` statement of Java. For reasons of orthogonality Spar also allows the range syntax in the `for` statement, and the `while`-like syntax in the `foreach` statement.

The `foreach` range can also be described as a vector range (see Section 2.3 for an explanation of vectors). For example, a two-dimensional array `b` would be initialized completely by:

```
foreach( i=[0,0]:getSize( b ) ){
    (b i).init();
}
```

As a further refinement, the range syntax allows masks. For example, if we wanted to initialize only the non-null elements of an array of objects, we could write:

```
foreach( i=[0,0]:getSize( b ), b i != null ){
    (b i).init();
}
```

2.2 Basic language constructs

Spar supports all the primitive types of Java (`int`, `float`, etc.). It also supports the new primitive type `complex`, since complex numbers are important in many numerical programs.

2.3 Vectors

Spar has extensive support for arrays, but this support is designed for bulk operations on variable-length arrays. There are circumstances where a more light-weight construct is appropriate. For this reason Spar provides vectors. Vectors are important as indices into multi-dimensional arrays, in particular in the parameterized classes that implement custom array representations.

A vector type is written as the name of an element type, followed by a Cartesian exponentiation operator ('^'), followed by a constant expression. For example, the following are valid vector declarations:

```
int^3 x;
double^2 y;
```

Like primitive types, vectors do not have to be created explicitly, and like primitive types, they are passed by value. A vector of length 1 is *not* the same as its element.

Provided that the operations are defined on the elements, Spar allows vector assignment, equality comparison, addition, subtraction, and multiplication. Vector expressions can be written as a list of values surrounded by square brackets. The code below demonstrates most of these operations.

```
int^3 x;

x[0] = 1;     // Fill elements of the vector
x[1] = 2;
x[3] = 3;
x = [1,2,3];  // This is equivalent to the above
int^3 y = x;  // Declaration with initialization
y += [1,0,0]; // Vector addition
y = y-x;      // Vector subtraction
y *= 2;       // Element-wise multiplication
```

2.4 Class objects

Spar generalizes Java classes by allowing parameterization. For example, the Java tutorial [3] shows a stack class that can hold elements of arbitrary type. If we wanted to restrict the stack to elements of a given type, we could implement a separate class for each type, but it is much more useful to implement a generic, parameterized, stack. In Spar this is possible as follows:

```
class TypedStack( type t ) {
    static final int STACK_EMPTY = -1;
    t[] stackelements;
    int topelement = STACK_EMPTY;

    void push( t e ){
        stackelements[++topelement] = e;
    }
```

```
    t pop() {
        return stackelements[topelement--];
    }
    boolean isEmpty(){
        return ( topelement == STACK_EMPTY );
    }
}
```

An instance of this class could be used as follows:

```
TypedStack(char) s = new TypedStack(char)();

s.push( 'a' );
s.push( 'b' );
char c = s.pop();
```

Parameterization is not restricted to type parameters. In Spar, parameterized classes are always expanded at compile-time.

2.5 Interfaces

Just like classes, Spar interfaces can be parameterized:

```
interface Collection( type t ){
    void add( t obj );
    void delete( t obj );
    t find( t obj );
    int currentCount();
}
```

A class can inherit this interface as follows:

```
Class Bag( type t ) implements Collection( t ) {
    . . .
};
```

An important use of parameterized interfaces is the Array interface, see Section 2.6.5.

2.6 Arrays

2.6.1 Array types

An array type is written as the name of an element type, followed by a number of abstract shape specifications. For example:

```
int[*] v;               // A 1-dimensional array
int[*,*] A;             // A 2-dimensional array
```

For compatibility with Java, Spar also allows Java-style declarations of one-dimensional arrays:

```
int[] n;                // A 1-dimensional array
```

This is a special case.

Here are some examples of declarations of array variables that create array objects:

```
int a[*] = new int[4];
short b[*,*] = new short[6,8];
int sq[*] = { 1, 4, 9, 16, 25, 36 };
real ident[*,*] = {{1,0,0}, {0,1,0}, {0,0,1}};
real vv[*][*] = {{1,0,0}, {0,1,0}, {0,0,1}};
String[] aos = { "array", "of", "string" };
```

Note that `ident` and `vv` have the same initialization expression, but they are *not* equivalent. The first is a two-dimensional array, the second is a one-dimensional array of one-dimensional arrays.

A component of an array is accessed by an expression that consists of an array reference followed by an `int` vector expression, as in: `A[i,j]`. All arrays start at element 0. A one-dimensional array with length n can be indexed by the integers 0 to $n-1$.

For example, the following assigns 2 to array element `[2,3]` of array `A`.

```
int[*,*] A = new int[9,9]; // A 2-dim array
A[2,3] = 2;                // An array access
```

This looks very similar to array access in other languages, but in the case of Spar this is a special case of a more general access construct, where arbitrary vector expressions can be used to access an array. Since any vector expression can be used, very powerful access expressions are possible. This is demonstrated in the following code:

```
int[*,*] A = new int[9,9]; // A 2-dim array
int^2 v = [1,1];           // A vector of 2 elm.
A v = 3;                   // A[1,1] = 3
A 3*v = 4;                 // A[3,3] = 4
```

As much as possible, array accesses are checked for bounds violations at compile time. If the array access cannot be checked at compile time, an index that is out of bounds causes an IndexOutOfBoundsException to be thrown.

2.6.2 Elastic arrays

Spar arrays that implement the ElasticArray interface can be grown and shrunk during their lifetime. For every array an `int` vector is maintained that contains the current *size* of the array in each of its dimensions, and an `int` is maintained that contains the *room* of the array: the current number of elements in the memory block of the array. Obviously, the room of an array is always large enough to hold the current array.

The size vector of an array is available through the method getSize, and can be set through the method setSize. The latter causes the array to grow or shrink; the room of the array will be enlarged if necessary.

The room of an array is available through the method getRoom, and can be set through the methods setRoom and fitRoom. The method setRoom will force the array to be grown to at least the given room or volume; it will never shrink the room. The method fitRoom reduces the room of the array to the current size of the array. At any time, even immediately after an invocation of fitRoom, the actual room of the array may be larger than requested.

2.6.3 Array expressions

An array expression is a shorthand notation for the construction of a (partial) copy of an array. For example, the following code will first construct an array `a`, and then construct a copy of the first row of `a`, and assign it to `v`.

```
int[*,*] a = {{0,1,2},{3,4,5},{6,7,8}};
int[*] v = a[0,0:a.getSize()[1]];
```

Note that `v` is a *copy*, so subsequent assignments to elements of `v` are not visible in `a`. Contrary to array range notations in many other languages, the top of the specified range is the first element *not* to be included.

The usual range shorthands apply: if no start of the range is given, 0 is assumed, and if no end of the range is given, the size in that dimension is assumed. Thus the declaration of `v` in the previous code fragment could be written as:

```
int[*] v = a[0,:];
```

2.6.4 Array statements

Array statements are a shorthand notation for a `foreach` statement that is executed for all elements of a selected range. For example, the code fragment

```
Block[*,*] a = new Block[5,7];
a[:,:].init();
```

will invoke the method init on all elements of Block array `a`. Since this is equivalent to a `foreach` statement, the init method of each of the array elements is not invoked in a prescribed order.

Similarly, array assignments are a shorthand for repeated assignments. For example:

```
int[*,*] a = new int[5,7];

a[:,:] = 0;
```

will zero the entire array `a`.

The expression at the right-hand side of the assignment will be evaluated only once. Thus,

```
int ix = 0;
int[*,*] a = new int[5,7];

a[:,:] = ix++;
```

will again zero the entire array `a`, and will leave `ix` with the value 1.

Last but not least, array assignments may contain an array at the right-hand side, instead of a single element. In that case every iteration of the `foreach` will use the implicit iteration vector as index for every assignment.

For example:

```
int[*,*] a = new int[5,7];
int[*,*] b = new int a.getSize();

b[:,:] = 1;
a[:,:] = b;
```

will copy `b` into `a`. The last statement could also be written as:

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

but a naive compiler would first create a copy of `b`, and leave it for the garbage collector.

Array statements never change the size of the array they work on. Any access that is out of bounds is detected at compile-time or run-time, and causes an error message or an IndexOutOfBoundsException exception.

2.6.5 The Array and ElasticArray interfaces

To support sparse matrices and other alternative matrix implementations, Spar allows any class that implements the Array interface to use the array access syntax.

The Array interface is defined as follows:

```
interface Array( type t, int n )
{
    t getElement( int^n index )
        throws IndexOutOfBoundsException;
    void storeElement( int^n index, t elm )
        throws IndexOutOfBoundsException,
               ArrayStoreException;
    int^n getSize();
}
```

If a class wants to allow growing and shrinking of its arrays, it should implement the ElasticArray interface:

```
interface ElasticArray( type t, int n )
    extends Array( t, n )
{
    void setSize( int^n sz )
        throws NegativeArraySizeException;
    int getRoom();
    void setRoom( int rm );
    void setRoom( int^n rm );
    void fitRoom();
}
```

For example, the following class defines a 'view' on the diagonal of a two-dimensional array:

```
class DiagonalView( type t ) implements Array(t,1)
{
    Array( t, 2 ) ref;

    DiagonalView( Array( t, 2 ) a ){ ref = a; }
    t getElement( int^1 ix ){
        return ref[ix[0],ix[0]];
    }
    void storeElement( int^1 ix, t elm ) {
        ref[ix[0],ix[0]] = elm;
    }
    int^1 getSize() {
        int^2 dims = ref.getSize();
        return [Math.min( dims[0], dims[1] )];
    }
}
```

This class can now be used as follows:

```
int[*,*] a = new double[5,5];
DiagonalView(double) v = DiagonalView( a );
a[:,:] = 0;
v[:] = 1;
```

This will construct and fill 'a' with the identity matrix (1 on the diagonal, 0 elsewhere).

As another example, the following class implements a transpose view on an array of arbitrary size.

```
class TransposeView( type t, int n )
    implements Array( t, n )
{
    Array( t, n ) ref;

    TransposeView( Array( t, n ) a ){ ref = a; }
    t getElement( int^n ix ){
        return ref revVector( n, ix );
    }
    void storeElement( int^n ix, t elm ){
        ref revVector( n, ix ) = elm;
    }
    int^n getSize(){
        return revVector( n, ref.getSize() );
    }
```

```
    static int^n revVector( int n, int^n v ){
        int ix;
        int^n res;

        for( ix=0; ix<n; ix++ ){
            res[(n-ix)-1] = v[ix];
        }
        return res;
    }
}
```

Note that this class even works for 0-dimensional and 1-dimensional arrays. Also note that since this class allows a view on an arbitrary Array class it is possible to compose views. This is demonstrated below, where two transpose views are composed.

This could be used as follows:

```
int[*,*] a = new double[5,5];
TransposeView(double) v = TransposeView( a );
TransposeView(double) w = TransposeView( v );
a[:,:] = 0;
v[:,0] = 1;
w[:,0] = 2;
```

This results in a matrix 'a' where all elements are set to 0, except that the first *column* (apart from a[0,0]) is set to 1 through the transpose view 'v' on 'a', and the first *row* is set to 2 through the transpose view 'w' on 'v'. Note that since 'w' is a transpose view on a transpose view, it is in effect an identity view.

2.7 Macros

Spar provides macros for two different reasons: (a) to allow the abstraction of simple constructs without paying the cost of a function call, and (b) to allow type abstraction, so that part of the functionality of C++ templates can be provided.

A macro is similar to an ordinary class, method, or constructor, but it is declared to be a macro with the `macro` keyword. For example:

```
class Stats {
    long sum;
    int n;

    macro Stats() { sum = 0; n = 0; }
    macro void update( int val ) {
        n++;
        sum += val;
    }
    macro float average() {
        return ((float) val)/((float) n);
    }
}
```

Macro methods are very similar to ordinary methods, but the compiler is required to expand them at compile time. Moreover, macro methods can have `type` parameters, which is not allowed in ordinary methods.

For example, classes such as the DiagonalView and TransposeView classes shown in the previous section usually declare most of their methods as macros for increased efficiency.

3 Compiling Spar

The compilation of Spar is very different from traditional Java compilation. Whereas Java code is traditionally compiled to a separate `.class` file for each Java class, all Spar code, including imported code, is collected in a single program file. All macros and parameterized classes are expanded, all methods are rewritten to stand-alone functions, etc. The resulting program is represented in our intermediate language Vnus [9]. In the Vnus representation, most language constructs described in this paper are rewritten; only multi-dimensional arrays and the `each` and `foreach` constructs remain. Subsequent engines will rewrite the `each` and `foreach`, map the tasks, and will generate C++ code with explicit parallelization and communication.

4 Conclusion

We have presented Spar, a programming language for semi-automatic parallel programming, in particular for the programming of array-based applications. The language has been developed as part of the Automap project, in which a compiler and run-time system are being developed for convenient and portable programming of parallel distributed-memory computers, in particular for array-based problems.

Apart from a few minor modifications, Spar is a superset of Java. This provides Spar with a modern, solid language as basis, and makes Spar more accessible. Spar extends Java with constructs for parallel programming, extensive support for array manipulation, and a number of other powerful language extensions. Although Java was designed for an entirely different purpose, it was surprisingly easy to embed our language constructs. The resulting programming language, Spar, retains the elegance of Java, and provides the programmer with sufficient expressive power to encode parallel array-based algorithms conveniently.

The language design for Spar has been completed, but an implementation is not yet available. A first prototype is expected shortly. A fully working parallelizing compiler for Vnus, the intermediate language in the Automap project, is already available.

Acknowledgements

The Spar language design has greatly benefitted from the comments of other members of the Automap team: Leo Breebaart, Paul Dechering, Frits Kuijlman, and Hai-Xian Lin.

References

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

[2] R. T. Boute. Funmath: towards a general formalism for system description in engineering applications. In P. P. Silvester, editor, *Advances in Electrical Engineering Software*, pages 215–226. Computational Mechanics Publications, Southampton, and Springer-Verlag, Berlin, August 1990.

[3] Mary Campione and Kathy Walrath. *The Java Tutorial.* The Java Series. Addison-Wesley, Reading, Massachusetts, August 1996.

[4] 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.

[5] K. Mani Chandy and Carl Kesselman. CC++: A declarative concurrent object oriented programming notation. Technical report, California Institue of Technology, September 1992.

[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] S. Chatterjee, J. R. Gilbert, F. J.E. Long, R. Schreiber, and Shang-Hua Teng. Generating local addresses and communication sets for data-parallel programs. *Journal of Parallel and Distributed Computing*, 26(1):72–84, April 1995.

[8] Paul Dechering, Leo Breebaart, Frits Kuijlman, and Kees van Reeuwijk. Semantics and implementations of a generalized `forall` statement for parallel languages. In *International Parallel Processing Symposium*, pages 542–548, Geneva, April 1997. IEEE.

[9] 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, P. Sadayappan, U. Bannerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *8th Intl. Workship, Languages and Compilers for Parallel Computing*, number 1033 in LNCS, pages 388–395, Columbus, Ohio, USA, August 1995. Springer Verlag.

[10] A. González Escribano, Valentín Cardeñoso Payo, and A.J.C. van Gemund. On the loss of parallelism by imposing synchronization structure. In *Proc. 1st EURO-PDS Int'l Conf. on Parallel and Distributed Systems*, Barcelona, June 1997.

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

[12] Geoffrey C. Fox and Wojtek Furmanski. Java for parallel computing and as a general language for scientific and engineering simulation and modelling. *Concurrency, Practice and Experience*, 9(6):415–425, June 1997.

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

[14] 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.

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

[16] Susan Flynn Hummel, Ton Ngo, and Harini Srinivasan. SPMD programming in Java. *Concurrency, Practice and Experience*, 9(6):621–631, June 1997.

[17] ISO/IEC. *ISO/IEC 1539 (Fortran 90)*, second edition, July 1991.

[18] David May. Occam. In *IFIP Conference: System Implementation Languages: Experience and Assessment, Canterbury*, September 1984.

[19] Martin Odersky and Philip Wadler. Pizza into Java: Translating theory into practice. In *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 146–159, Paris, France, January 1997. ACM.

[20] E.M. Paalvast, H.J. Sips, and L.C. Breebaart. Booster: a high-level language for portable parallel algorithms. *Applied Numerical Mathematics: Transactions of IMACS*, 8(2):177–192, September 1991.

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

[22] A.J.C van Gemund. The importance of synchronization structure in parallel program optimization. In *Proc. 11th ACM Int'l Conf. on Supercomputing*, Vienna, July 1997. ACM. to be published.

[23] C. van Reeuwijk. *The implementation of a system description language and its semantic functions*. PhD thesis, Delft University of Technology, Department of Electrical Engineering, Delft, The Netherlands, July 1991.

[24] 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.

[25] C. van Reeuwijk, H.J. Sips, H.X. Lin, and A.J.C. van Gemund. Automap: A parallel coordination-base programming system. Tech. Rep. 1-68340-44(1997)04, Delft University of Technology, April 1997.

[26] M. Wolfe. Doany: Not just another parallel loop. In Uptal Banerjee, David Gelernter, Alex Nicolau, and David Padua, editors, *Proceedings of the 5th International Workshop on Languages and Compilers for Parallel Computing*, Lecture Notes in Computer Science, pages 421–433, New Haven, Connecticut, August 1992. Springer-Verlag.

[27] M. Wolfe. *High Performance Compilers for Parallel Computing.* Addison-Wesley, 1995.