Booster: A High-Level Language for Portable Parallel Algorithms†

Edwin M. Paalvast, Henk J. Sips*, Leo C. Breebaart*

Institute of Applied Computer Science (ITI-TNO), Delft, The Netherlands
*Delft University of Technology, Delft, The Netherlands

abstract

The development of programming languages suitable to express parallel algorithms in is crucial to the pace of acceptance of parallel processors for production applications. As in sequential programming, portability of parallel software is a strongly desirable feature. Portability in this respect means that given an algorithm description in a parallel programming language, it must be possible with relatively little effort, to generate efficient code for several classes of (parallel) architectures.

In this paper, the language *Booster* is described. *Booster* is a high-level, fourth generation, parallel programming language. The language has been designed to program parallel algorithms for a wide variety of target parallel architectures. *Booster* has a strong separation of concerns, featuring a.o. a clear separation of algorithm description and algorithm decomposition and -representation. Programs written in *Booster* are translated to imperative languages, such as FORTRAN or C, and can be easily integrated in large applications. Parallelism can be obtained by applying data- and/or code decomposition. Once algorithm and decomposition are described the transformation is done automatically.

1. Introduction

In applications in the field of technical and scientific programming the actual calculation- intensive kernel is often restricted to a small part of the application. An important area of research is to find programming languages and paradigms to describe the algorithms in these kernels. A number of criteria can be formulated with respect to the parallel properties of such a language. Among these are ease of programming of parallel algorithms, preservation of parallelism, maintainability, and decomposition properties (with respect to code, data, and granularity). Also of importance are efficiency of the generated target code and the class of applicable parallel architectures (a.o. the suitability for vector- and/or parallel processing).

Ease of programming refers to the level of abstraction provided in the language. Language constructs with a high level of abstraction can ease programming considerably, resulting in

† This research is partially funded by SPIN.

very maintainable programs, and also offering opportunities to preserve parallelism. However, a price often being paid is a severe loss of execution efficiency. Also parallelism, although preserved, cannot always be extracted in the required form, i.e. the granularity cannot be influenced and/or certain coherence information (such as vector-properties) is lost.

The class of applicable parallel architectures is another important factor. When designing parallel applications for execution on general-purpose parallel architectures, software development and maintenance costs must be kept to an acceptable level. Therefore, the class of parallel target architectures must be kept as large as possible. This is also of importance because it is not to be expected that in the near future a single architecture will emerge from the current developments. But even within a single parallel architecture, different properties of the elements of the model of computation of that architecture can have a large impact on the performance of a parallel program.

To meet the above demands, the *Booster* programming language has been developed. *Booster* is a high-level, fourth-generation, parallel programming language: it offers users a high level of abstraction in programming algorithms and a strong separation of concerns. At the same time, it remains possible to generate efficient parallel programs

After a short review of the current status in parallel programming in Section 2, the basic concepts of the *Booster* language are explained in Section 3. Section 4 shows how data representations can be tailored to machine architectures, independently from the program implementing the algorithm. The *Booster* language is illustrated by means of a worked out example in Section 5. Finally, in Section 6 it is described how the translation of *Booster* programs takes place and how these programs are embedded in applications.

2. Developments in Parallel Programming

The problem of parallel programming can be viewed through the different roles of those involved in application design. The first role is that of the *algorithm developer*, whose main requirements for a language and corresponding support environment are expressiveness, the amount of feed-back on the algorithm's performance that is received, and ease of experimentation. The second role is that of the *programmer*, who is mainly interested in a language model that is close to the problem model, in efficient and verbose compilers, and advanced support tools. Third and last is the role of the software *maintainer*. Here the focus is on portability, maintainability, and machine and/or target language independence. The aforementioned roles are used to access current developments in parallel programming.

Much effort has been placed on the extraction of parallelism from standard imperative languages like FORTRAN [Allen85, Padua80,86]. However, the clear advantage of being applicable to programs already written for sequential machines is paired with a number of unwanted features, such as the lack of appropriate high-level abstractions. The consequence is that many existing codes must be re-written to bring them in a form which is more suited to parallel pro-

cessing. Another lacking feature is that data decomposition is done implicitly by code restructuring and therefore the consequences, especially for distributed memory architectures, are hard to evaluate. The same holds for the granularity issue. These facts pose problems to algorithm developers who want to experiment with different parallel algorithms, rather than with parallelizing compilers.

A further improvement is to let the programmer specify the problem in terms of multiple (concurrent) processes, where each process contains a block of sequential code with explicit definitions of communication with other processes (examples are concurrent MODULA [Mühl88], concurrent C [Gehani86], Occam [May86] and POOL-T [America86]). Although in this approach more flexibility is gained, many of the flaws mentioned in the previous approach remain.

A different approach is to use a programming paradigm that does not obscure parallelism and leaves parallelization to the compiler. Examples are dataflow- (e.g. ID Nouveau [Arvind88], SISAL [Sked85]) and functional languages (Haskell [Hudak89], Miranda [Turner 85], Crystal [Chen86]). A problem regarding the execution efficiency with those languages is a.o. the single assignment nature, yielding to much overhead when manipulating large data structures. Compilers for functional programs use graph rewriting techniques; by solving parts of the graphs concurrently, parallelism can be obtained [Peyton87]. Extracting parallelism from graphs poses many problems with respect to granularity and load balancing. If programmers know an optimal mapping of their program to a given machine, this cannot be expressed in the language (an exception is ParAlf [Hudak88]).

Of a different nature are programming models like LINDA [Carriero89] and STRAND [Foster89]. In both approaches, standard sequential programs can be "glued" together by means of a restricted number of primitives. Although these models lead to fast implementations and portability over a wide range of architectures, tuning of a specific algorithm is not possible or results in re-writing of the algorithm.

Some attempts are directed towards the development of application oriented high-level parallel language formalisms. By tailoring the language constructs to an application domain, more information regarding parallelism is available at compile time. An example of this approach is SUSPENSE [Ruppelt88], aimed at the specification of multi-grid methods for solving PDE's.

A conclusion from the above approaches is that in current languages, the associated compilers are not capable of generating efficient code automatically for arbitrary program-machine combinations. This insight has led to the approach of parallelism through program annotations, incorporating (explicit) data decomposition(s) . From these data decomposition specifications, SPMD (Single Process Multiple Data) code [Karp87] can be generated from the program automatically. This approach is followed by [Callahan88, Gerndt89, Kennedy89] in FORTRAN, by [Rogers89] in Id Nouveau, by [Koelbel87] in BLAZE, and by [Quinn89] in C*. This concept is also followed in *Booster* [Paalvast90].

3. *Booster* Language concepts

Booster is a high-level, fourth generation, algorithm description language for sequential- and parallel computers. Parallel computers may be either shared- or distributed memory systems. The basic computation engines in any machine in the aforementioned classes are assumed to have the traditional von Neumann type of machine as model of computation. Each computation engine may have vector processing facilities. *Booster* has been designed such that many optimizations can be done at compile time, since this is the key factor for realizing efficient code generation.

3.1 Index- and Data Domains

In a conventional programming language (such as Fortran) the array is used as the basic data structure for storing related data elements. These elements can be accessed by use of indices to the array. An index is a rigid pointer to a memory-location. It is not possible in these languages to reason about or manipulate indices themselves. Only the data can be moved around or changed, and it is precisely this which makes arrays so awkward whenever sorting or insertion (for example) needs to take place. The use of indirect addressing (e.g. index files) to keep a large database sorted on different keywords is an example of how useful it can be to regard the indices to arrays as a separate, manoeuvrable collection of entities. This is particularly true for parallel programming, where it is often important to identify sets of index values that refer to data upon which computations can be executed in parallel. The importance of this has already been recognized in a language like ACTUS [Perrott87].

In *Booster*, these observations have resulted in a strict distinction between data- and index-domain of a program. The *data-domain* consists of several possible data types, just as in conventional languages. Supported in *Booster* are integers, reals, booleans, and records. The *index-domain* consists of non-negative integer values. On the index-domain ordered index sets can be defined, and operations can be performed on these sets independent of the data-elements that the index values in question refer to.

3.2 Shapes and Views

There are two concepts in *Booster* to reflect the two domains. The first is the *shape*, *Booster's* equivalent of a traditional array: a finite set of elements of a certain data-type, accessible through indices. Unlike arrays, shapes need not necessarily be rectangular (for convenience we will, for the moment, assume that they are). Shapes serve, from the algorithm designer's point of view, as the basic placeholders for the algorithm's data: input-, output-, and intermediate values (if any) are stored within shapes. As we will see later on, this does not necessarily mean that they are represented in memory that way, but the algorithm designer might think so. Shapes have an associated ordered index set, through which the elements of the shape can be accessed.

The second concept is that of the *view*. A view is a function that takes the index set of a certain shape as input, and outputs a different index set. Through the indices in this new index set one can still access the elements of the original shape, but it is as though we now 'view' the shape's data-elements through a different window, hence the name.

Views basically come in four flavours, each of which we will illustrate with a simple example.

First, we need to know how to define shapes in *Booster*:

```
SHAPE A (20) OF REAL;
SHAPE B (3#10) OF REAL;
```

The definition of shapes is fairly standard. In the first statement, A is declared to be a vector of *20* numbers of the type real. The basic index set for this shape is the ordered set $\{0, 1, ..., 19\}$ (index sets of shapes are ordered from zero on in each dimension). Next, B is declared to be a matrix of *3* by *10* elements. The index set for this shape is the ordered set $\{(0,0), (0,1), (0,2), ... (2,8), (2,9)\}$.

3.3 Content Statements

In so-called *content statements* we can manipulate the data stored in shapes:

```
A := 2.5;
A[10] := 5;
B[1,8] := 3.1416;
```

In the first content statement, all elements of A are initialized to *2.5*. In the second statement, the value *5* is stored in the 10th element of A, and so on. Content statements are *Booster's* equivalent of the standard assignment statement in procedural languages. Apart from standard scalar operators, *Booster* supports their multi-dimensional equivalents. For example, a vector A times a vector B is written as

```
A := A*B;
```

Multi-dimensional operators are defined strictly *element-wise*, that is the operator is applied to the elements which have the same *ordinal* number[1] and all such operations are applied *concurrently*. Hence, the above statement is interpreted as `A[1]:=A[1]*B[1]`, `A[2]:=A[2]*B[2]`, etc. For linear algebra type of multi-dimensional operators the *function* construct in *Booster* is to be used, as will be discussed shortly

[1] The ordinal number of an element is the sequence number derived from ordering the indices lexicographically, e.g. (0,0), (0,1), (1,0), (1,1) for a 2#2 shape. Hence, the element (1,0) has an ordinal number of 2.

3.4 View Statements

We can now manipulate index sets in so-called *view statements*. The easiest view to define is the identity view:

```
V <- A;
```

V is called a *view identifier* and does not need to be declared. Through the statement V<-A, the view identifier V is bound to the index set of the shape A. Note the different assignment symbols for view- and content-statements; '<-' for view statements and ':=' for content statements. After the view statement in the code given above, the three content statements below will have exactly the same effect.

```
A[0] := A[10];
A[0] := V[10];
V[0] := V[10];
```

3.5 Selection Views

The first type of view is the *selection* view:

```
V <- A[5:15];
```

The *index expression* 5:15 selects the subset or *range* of indices *5* through *15* of A and binds them to the view identifier V. The identifier V can be used to access A through the index set {*0, 1, ... 14*}. Again, the two content statements below are semantically identical, given the view statement that precedes them.

```
V[0] := V[5];
A[5] := A[10];
```

Note that the element V[0] actually refers to A[5], etc: *renumbering* of the index sets after a view statement causes all index sets to start from zero, just as the original index set does. A itself is never affected by any view statement.

3.6 Permutation Views

The second type of view is the *permutation* view:

```
V[i] <- A[19-i];
```

Again the following statements are semantically equivalent:

```
V[0]   := V[5];
A[19]  := V[5];
A[19]  := A[14];
```

Here a new language construct is introduced, the *free variable* `i`. We will come to the exact syntax and semantics of this construct later, but the above example should be intuitive enough: through `V` we access the elements of `A` in reverse order. Permutation views are an efficient way of creating high level indirect addressing.

3.7 Dimension Changing Views

Free variables can be used for even more powerful purposes, as is illustrated by the third type of view: the *dimension changing* view. First an example of a *dimension increasing* view:

```
V{4#5}[i,j] <- A[(4*i)+j];

      // Equivalent content statements:
V[0,0] := V[2,3];
A[0]   := A[11];
```

Here `V` is a two dimensional view identifier and `A` a one dimensional shape. The relation between the index sets of `V` and `A` is defined by the functional relations of the free variables `i` and `j`. In the permutation view statement given in the previous example, the domain of `i` and hence the resulting index set could be deduced by the compiler from the declaration of `A`. In this case the compiler needs to be told how to divide the *20* elements of `A` over the two dimensions of `V`. Hence, the required extra specification {4#5} in the view statement, effectively defining the domain of `V`. For all practical purposes, the identifier `V` now becomes a two dimensional structure. The fact that the index set of this structure refers, *via* a view function, to a one-dimensional structure is completely transparent to the 'user' of `V`.

Dimension decreasing views are relatively simple:

```
V <- B[1,_];
```

Here the second row of `B` is selected and assigned to the view identifier `V`. The underscore character stands for selecting all elements in that dimension.

3.8 Content selection views

The views presented so far, only define structural relations, i.e. independent on the contents (values) of the elements of a shape. Some algorithms have a content sensitive behavior, hence

we need a construct for defining content sensitive relations. In *Booster*, these relations can be expressed by *content selection views*. They are best explained by an example. Take the following content statement:

```
A[B[0,_]>1] := 5;
```

`B[0,_]>1` returns all indices of the first row of `B` which are larger than one. Those index values are then used to assign `5` to the corresponding elements of `A`. Also in content selection views, the convention of element-wise operations must be obeyed. If not, an error will result.

3.9 View Functions

Related view statements can be capsulated into *view functions*. View functions are declared by specifying the name of the view function, the input- and output arguments, and their respective index sets. For example, take the following view function `Unzip`, which splits the indices of a vector in its even and odd numbered indices:

```
VIEW FUNCTION Unzip (E) -> (Even,Odd);
E (n);
Even (n div 2 + n mod 2);
Odd (n div 2);

BEGIN
    Even[i] <- A[2*i];
    Odd[i]  <- A[2*i+1];
END;
```

The formal arguments `E`, `Even`, and `Odd` denote the input and output index sets, respectively. The use of the implicit *index parameter n* allows the view function to be applied to vectors of arbitrary length. No content statements may be used in the body of a view function. Note that renumbering compacts the selected collections of non-consecutive indices into rectangular index sets that start from zero.

Being defined, the view function `Unzip` can now be used in other view statements, such as in

```
(Ev,Od) <- Unzip(A);
```

The actual input argument of `Unzip` is now `A`, and the even and odd indices are bound to the view identifiers `Ev` and `Od`, respectively.

3.10 Non Rectangular Selections

We have not exhausted the expressiveness of the *Booster* language yet. As a final example, we will show how to use free variables *within range expressions* to select non-rectangular structures, such as triangular matrices:

```
SHAPE C (2*n#2*n) OF REAL;
BEGIN
    INDEX bs_x = n*(i div 2); bs_y = n*(i mod 2) END;
    V{4#n#n}[i,j,k] <- C[bs_x:bs_x+n-1, bs_y:bs_y+n-1][j,k];
    V[i,j,0:j] := V[i,j,0:j] * 2;
END;
```

The shape C is declared to be a square matrix with an even number of rows and columns (see Fig.1). The INDEX statement is a *Booster* macro statement: it has the effect of textual substitution, making complex index expressions more readable. The view statement that follows is a dimension increasing view, which makes V a vector of submatrices of C. The final content statement has the effect of selecting all the lower under triangular parts of these submatrices, and changing their values.

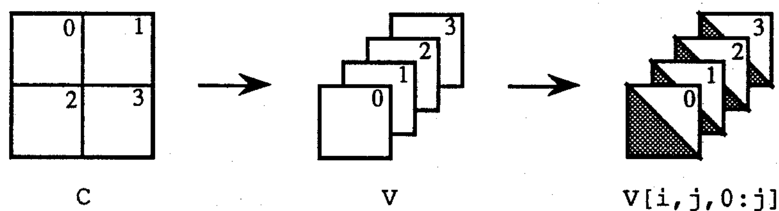


Fig.1 Non-rectangular selection example.

3.11 Functions

Functions can be used in *Booster* to encapsulate *both* view- and content statements. Consider the function Inprod, which defines the inner product of two vectors:

```
FUNCTION Inprod (C,D) -> (E);
C(n), D(n), E(1) OF REAL;
BEGIN
    E := REDUCE(+,C*D);
END;
```

The REDUCE operator is a built-in function for performing reduction operations. Once defined the function can be applied, e.g.

```
A[0] := Inprod(A[0:19],B[0,_])
```

Functions in *Booster* contain no side-effects, i.e. no values are retained between invocations. When a function has one output argument and at most two input arguments, it can also be used in infix notation.

3.12 Modules

The standard encapsulation facility in *Booster* is the *module*. A *Booster* program consists of a set of modules. A module has an interface definition like a function. The main difference between a module and a function is that modules are the interface to the outer world. The integration of *Booster* programs in applications is always handled through the module.

3.12 Control Flow

For control flow, *Booster* offers several control flow constructs similar to those found in conventional languages. Available are the standard IF-THEN-ELSE statement for conditional execution and WHILE/ITER statements for repetitive executions (ITER-loops execute a fixed number of times). Note the total absence of For-like loops.

4. Mapping data and algorithms to (parallel) machines

In programming sequential machines one is in general not concerned about the way the data structures are represented in real memory. Only when inefficiencies are obvious, such as in sparse matrices, the programmer will try to optimize memory access and/or occupation. However, this reluctance in dealing with these issues is often not realistic: column-wise or row-wise traversal of a data structure can have a large impact on performance; when the real machine has a complex memory hierarchy with caches, vector lengths of operands are of importance with respect to performance. On the other hand, the programmer who is willing to deal with this issue, is often confronted with a complicated rewrite of the code. This makes experimentation and tuning difficult.

In parallel programming the situation is even more complicated, because extra information on data- and algorithm decomposition and -synchronization has to be supplied. Also in this case, the programmer would like to consider a parallel machine as if it were a sequential one and supply as little extra information as possible. On the other hand, maximal performance and control is expected in return. This ambivalence is reflected in parallel language paradigms, which, regarding to the representation problem, can be partitioned into two categories: implicit and explicit.

Implicit specification methods, like functional and dataflow languages, leave the detection of parallelism and mapping on a parallel machine to the compiler. However, as remarked before, contemporary compilers do not produce efficient translations for arbitrary program-machine combinations.

In explicit specification methods, communication and synchronization is explicitly specified in the algorithm. This has the disadvantage that one has to program multiple threads of control, which can be very hard to debug. Besides that, experimentation with different versions of the same parallel algorithm, for example different granularities and decompositions, is in general rather cumbersome. Usually comparably small changes require major restructuring of the programs communication and synchronization framework. An example is alternating between a row- and column-wise decomposition in a matrix multiplication.

To discuss *Booster's* method of mapping, let us return to the concepts of shapes and views. To the algorithm designer, shapes define the total amount of data space needed in the algorithm. However, shapes need not be necessarily directly translated in equally dimensioned data structures in the target language. In order to do so, a mechanism is needed to enforce other mappings than the default mappings made by the compiler. In *Booster* this mechanism is provided by the same index manipulation construct as used in constructing algorithms, namely the *view*. Hence, the relation between shapes and their actual representation on a machine (or, alternatively, a language accepted by the machine) is defined by views. This principle is illustrated in Fig.2.

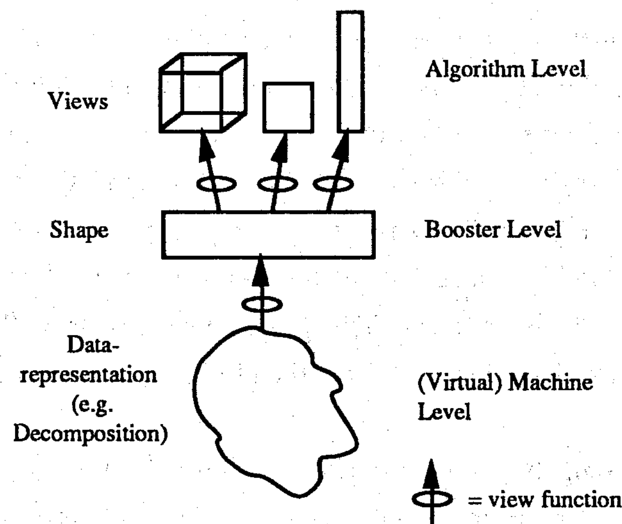


Fig.2. Data representation

As an example, consider a shape A with index set n#m and the following four different mappings of this shape on a memory:

```
A[i,j] <- Mem[i,j];
A[i,j] <- Mem[j,i] ;
A[i,j] <- Lin_mem[i*n+j];
A[i,j] <- Lin_mem[j*m+i];
```

The first mapping just defines the default mapping of the shape to an identically structured representation. In the second mapping, the indices are reversed, changing a row traversal in a column traversal. The last two mappings give a one-dimensional representation of the two-dimensional shape A; the first a row-wise storage scheme and the second a column-wise storage scheme.

Data decomposition

Data decomposition is one of the most successful techniques to obtain parallelism. For distributed memory machines the data also has to be physically split and assigned to processing elements. To obtain a distributed mapping of a shape, again the view concept of *Booster* can be used. If, for example, a two-dimensional shape is to be decomposed in a row-wise fashion for parallel machine with p processors, this is described with the following *dimension decreasing* view:

```
VIEW FUNCTION row_decompose (R,p) -> (Q)
Q (n # n);
R (p # (n div p) # n);
BEGIN
    Q[i,j] <- R[i div (n div p),i mod (n div p),j];
END;
```

The principle is illustrated below in Fig. 3.

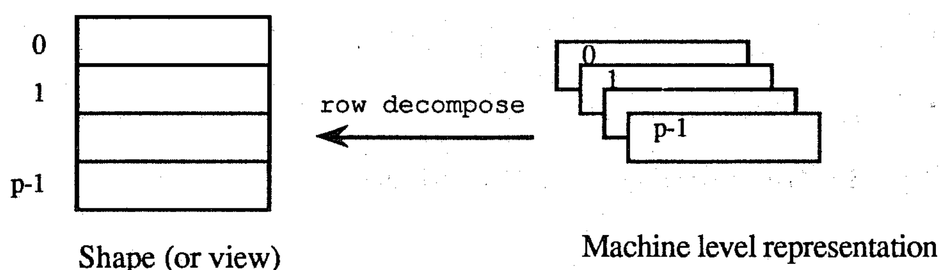


Fig.3 Row-wise decomposition

In *Booster*, the mapping and representation problem is solved as follows: the programmer has the possibility to express the mapping of the algorithm explicitly, but separate from the algorithm. In this way, the flexibility and clarity of the implicit method with the efficiency of the explicit is combined. The mapping can be described on several levels of detail, ranging from no mapping at all, to a detailed description of data decomposition and assignment, data transport, and data representation. The compiler takes both algorithm and mapping description as input and produces an explicit parallel algorithm including synchronization and communication in a parallel extension of imperative languages like FORTRAN and C or in languages like OCCAM and ADA.

5. Example: Block Factorization

To illustrate the programming and mapping of algorithms in *Booster*, a (2,2) block factorization of a matrix **A** [Duff89] is taken as an example. Let

$$\mathbf{A} = \begin{pmatrix} \mathbf{A}_{11} & \mathbf{A}_{12} \\ \mathbf{A}_{21} & \mathbf{A}_{22} \end{pmatrix}$$

where $\mathbf{A}_{11}$ and $\mathbf{A}_{22}$ are square submatrices. The LU factorization of of **A** may be written in the form

$$\begin{pmatrix} \mathbf{A}_{11} & \mathbf{A}_{12} \\ \mathbf{A}_{21} & \mathbf{A}_{22} \end{pmatrix} = \begin{pmatrix} \mathbf{L}_{11} & \\ \mathbf{L}_{21} & \mathbf{L}_{22} \end{pmatrix} \begin{pmatrix} \mathbf{U}_{11} & \mathbf{U}_{12} \\ & \mathbf{U}_{22} \end{pmatrix}$$

where $\mathbf{L}_{11}$ and $\mathbf{L}_{22}$ are lower triangular submatrices and $\mathbf{U}_{11}$ and $\mathbf{U}_{22}$ are upper triangular submatrices. Hence, the algorithm involves the solution of the following four equations:

(1)	$\mathbf{A}_{11} = \mathbf{L}_{11}\mathbf{U}_{11},$	LU-factorization
(2)	$\mathbf{L}_{11}\mathbf{U}_{12} = \mathbf{A}_{12},$	forward substitution
(3)	$\mathbf{L}_{21}\mathbf{U}_{11} = \mathbf{A}_{21},$	backward substitution
(4)	$\mathbf{A}_{22} - \mathbf{L}_{21}\mathbf{U}_{12} = \mathbf{L}_{22}\mathbf{U}_{22}$	LU-factorization

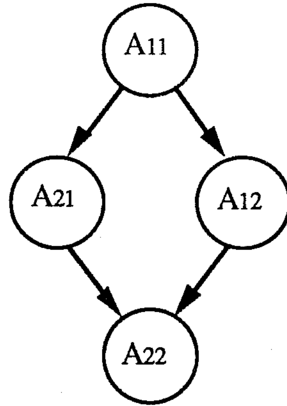
The corresponding *Booster* program is as follows:

```
MODULE Block_Factorize (A) -> (A);
A (n#n) OF REAL;


FUNCTION LU-Factorize (A) -> (A);
BEGIN
        ..... // Booster code performing the LU factorization
END;

FUNCTION matrix_mult PRIORITY 7 (A,B) -> (C);
A,B,C (n#n) OF REAL;
BEGIN
        C[i,j] := REDUCE(+,A[i,_]*B[_,j]);
END;

BEGIN
 // Define the four regions
        INDEX h = (n div 2)-1;
        A11 <- A[0:h,0:h];
        A12 <- A[0:h,h+1:upb];
        A21 <- A[h+1:ubp,0:h];
        A22 <- A[h+1:upb,h:upb];
        A11 := LU_Factorize(A11);
        A12[i,j] := A12[i,j] - A11[i,0:i]*A12[0:i,j];
        A21[i,j] := (A21[i,j] - A11[0:j-1,j]*A21[i,0:j-1])/A11[j,j];
        A22 := LU_Factorize(A22 - A21 matrix_mult A12);
END.
```

In the module header, the structure A is imported and need not be declared as a shape. What follows are two function definitions. The first function, LU-Factorize is not specified here, but can be found in [Paalvast89]. The second function is a matrix multiplication definition. REDUCE is the built-in reduction function. Because the function can be used in infix notation, a priority has to be declared. The program itself consists of four view statements, defining the four regions in A and four content statements defining the actual operations.

Decomposition of the algorithm:

Lets first consider the dependence graph of the above algorithm.



This dependence graph shows that the second and third step of the algorithm can be executed in parallel. This influences the decomposition of the submatrices: A_{11} and A_{22} are decomposed row-wise in p segments and mapped onto all p processors. A_{12} and A_{21} can be executed in parallel and are decomposed in $(p/2)$ segments and each mapped onto the half of the processors. The decomposition is illustrated in Fig.4.

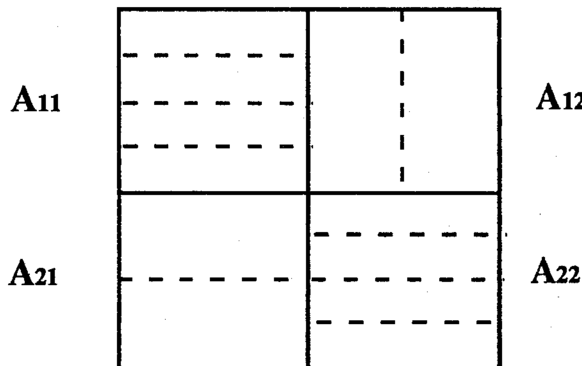


Fig.4. Depiction of the decomposition on matrix A.

The associated annotation module is written down as

```
ANNOTATION MODULE Block_Factorize;
IMPORT A11, A21, A12, A22 (n#n) FROM Block_Factorize;
       Proc (p#n#n) FROM Some_Processor_Model;

VIEW FUNCTION column_decompose (R) -> (Q)
Q (n # n);
R (q # (n div q) # n);
BEGIN
      Q[i,j] <- R[i mod (n div q),i div (n div q),j];
END;
```

```
BEGIN
        INDEX h = (p div 2)-1;
              A11 <- row_decompose(Proc[0:p-1,_,_]);
              A21 <- row_decompose(Proc[0:h,_,_]);
              A21 <- column_decompose(Proc[h+1,_,_]);
              A22 <- row_decompose(Proc[0:p-1,_,_]);
        END;
END.
```

The view function `row_decompose` was already introduced in Section 4.

6. Translating *Booster* to Conventional Languages.

Booster programs are not translated into machine language programs for the intended (parallel) target computers. Instead, code in a conventional, procedural high level language is generated. The current target languages are parallel dialects of Fortran and C.
The reason for choosing high-level languages is that the development of efficient compilers for generating machine code is a very laborious task and requires detailed understanding of that machine. Compilers for actual parallel machines are in general best provided by the manufacturers of that machine.

Having a powerful language is one thing; having optimized code is another thing. The same holds for decomposition: a given decomposition must result in individual programs for the processing elements and the appropriate communication and synchronization primitives as defined by the programming model of the target machine.

The optimization on *Booster* programs is done by compile-time reductions of views in content expressions, together with classical dependence analysis between content expressions. To this purpose, a calculus with associated rewrite rules has been developed to perform these compile-time optimizations [Gemund89].

Since *Booster* programs are translated in procedural languages, the *Booster* language kernel has been kept small. Input/output operations have been omitted and must be done in the language or environment *Booster* is to be embedded in.

7. References

[Allen85] J.R. Allen, K. Kennedy, "A parallel programming environment," *IEEE Software*, July 1985.

[America86] P. America, "Definition of the programming language POOL-T," *Internal report No. 0091 of Esprit project 415-A*, Philips Research Laboratories, Eindhoven, The Netherlands, 1986.

[Arvind88] Arvind, K. Ekanadham, "Future scientific programming on parallel machines," *Journal on Parallel and Distributed Computing*, Vol.5, no.5, October 1988.

[Callahan88] D. Callahan, K. Kennedy, "Compiling Programs for Distributed-Memory Multiprocessors," *The Journal of Supercomputing*, Vol. 2, No. 2, October 1988, pp. 151-169.

[Carriero89] N. Carriero, D. Gelernter, "How to write parallel programs: a guide to the perplexed," *ACM Computing Surveys*, Vol. 21, no.3, Sepetember 1989.

[Chen86] M.C. Chen, "Very-High-Level parallel programming in Crystal," *Proceedings of the First Conference on Hypercube Multiprocessors*, Knoxville, Tennessee, August 26-27, 1985, pp. 39 - 47.

[Duff89] Duff I.S., Erisman A.M., Reid J.K., *Direct Methods for Sparse Matrices*, Oxford Science Publications, 1989.

[Foster89] I. Foster, S. Taylor, *Strand: New Concepts in Parallel Programming*, Prentice Hall, 1989.

[Gehani86] N.H. Gehani, W.D. Roome, "Concurrent C," *Software-Practice and Experience,* Volume 16(9), September 1986, pp. 821 - 844.

[Gemund89] A. van Gemund, "A view language and calculus," *Report no. 89 ITI B 46,* Institute of Applied Computer Science, Delft, 1989.

[Gerndt89] M. Gerndt, "Array distribution in SUPERB," *Proceedings of the Third International Conference on Supercomputing*, Crete, Greece, June 1989, pp. 164-174.

[Hudak88] P. Hudak, "Exploring parafunctional programming: separating the What from the How," *IEEE Software*, January 1988.

[Hudak89] P. Hudak, "Conception, evolution, and application of functional programming languages," *ACM Computing Surveys*, Vol. 21, no.3, September 1989.

[Karp87] A.H. Karp, "Programming for parallelism," *IEEE Computer*, May 1987, pp. 43-57.

[Kennedy89] K. Kennedy, H.P. Zima, "Virtual Shaped Memory for Distributed-Memory Machines," *Proceedings of the Fourth Hypercube Conference, Monterey, California*, March 1989, 6 pp.

[Koelbel87] C. Koelbel, P. Mehrotra, J. Van Rosendale, "Semi-automatic domain decomposition in BLAZE," *Proceedings of the 1987 International Conference on Parallel Processing*, August 17-21, 1987, pp. 521 - 524.

[May86] D. May, D. Pountain, "A tutorial introduction to OCCAM programming," *INMOS OCCAM tutorial*, 1986.

[Mühl88] H. Mühlenbein, Th. Schieder, S. Streitz, "Network programming with Muppet," *Journal of Parallel and Distributed Computing*, Vol. 5, (5), October 1988.

[Paalvast89] E.M. Paalvast, H.J. Sips, "A High-Level Language for the Description of Parallel Algorithms," *Proceedings of Parallel Computing '89*, August 1989, Leiden, The Netherlands, North-Holland publ. co.

[Paalvast90] E.M. Paalvast, A.J. van Gemund, H.J. Sips, "A method for parallel program generation with an application to the Booster Language", *Proceedings 1990 ACM International Conference on Supercomputers*, June 1990, Amsterdam.

[Padua80] D. Padua, D. Kuck, D. Lawrie, "High-Speed multiprocessors and compilation techniques," *IEEE Trans. on Comput.,* Vol C-29 (9) , September 1980, pp. 763 - 776.

[Padua86] D. A. Padua, M. J. Wolfe, "Advanced compiler optimizations for supercomputers," *Communications of the ACM*, Vol 29 (12), December, 1986, pp. 1184 - 1201.

[Perrott87] R.H. Perrott, R.W. Little, P.S. Dhillon, "The design and implementation of a Pascal-based language for array processor architectures," *Journal of Parallel and Distributed Computing*, June 1987.

[Peyton87] S.L. Peyton-Jones, "The implementation of functional programming languages," *Prentice-Hall international, Series in Computer Science*, 1987.

[Quinn89] M.J. Quinn, P.J. Hatcher, "Data parallel programming on multicomputers," *Report no. PLC-89-18*, Dept. of Computer Science, University of New Hampshire, March 1989.

[Rogers89] A. Rogers, K. Pingali, "Process decomposition through locality of reference," *ACM Sigplan '89 Conference on Programming Language Design and Implementation*, June 1989, Portland Oregon.

[Ruppelt88] Th. Ruppelt, G. Wirtz, "From Mathematical Specifications to Parallel Programs on a Message-Based System," *Proceedings of the International Conference on Supercomputing*, 1988, St. Malo, France.

[Sked85] S. Skedzielewski, J. McGraw, S. Allen, R. Oldehoeft, J. Glauert, "SISAL : Streams and Iteration in a Single Assignment Language," *Report Lawrence Livermore National Laboratory* , M - 146, March 1985.

[Turner85] D.A. Turner, "Miranda: A non-strict functional language with polymorphic types," *Lecture Notes in Computer Science*, Springer Verlag, Vol. 201, 1985, pp. 1 - 16.