1990 ACM International Conference on Supercomputing,
June 11-15, 1990, Amsterdam, The Netherlands

A Method for Parallel Program Generation with an Application to the *Booster* Language†

Edwin M. Paalvast, Arjan J. van Gemund
TNO Institute of Applied Computer Science (ITI-TNO)
P.O. Box 214, NL-2600 AE Delft, The Netherlands

Henk J. Sips
Delft University of Technology
P.O. Box 5046, NL-2600 GA Delft, The Netherlands

Abstract

This paper describes a translation method for the automatic parallelization of programs based on a separately specified representation of the data. The method unifies the concept of data-representation on the algorithm-level as well as machine-level, based on the so-called view concept. It is shown that given a decomposition of the data, application of the translation method to the view-based *Booster* programming language results in efficient SPMD-code for distributed- as well as shared-memory architectures. It will be argued that the method is not restricted to *Booster*, but can also be applied to other languages.

1. Introduction

In programming either shared- or distributed-memory parallel computers, programmers would like to consider them as being uni-processors and supply as little extra information as possible on code parallelization and data partitioning. On the other hand, maximum speed-up is desired, without loss of portability. This trade-off is reflected in the existence of a variety of parallel language paradigms, which, regarding to the decomposition method, can be divided into two categories: implicit and explicit. Languages based on implicit descriptions, like functional [Hudak89, Chen88] and dataflow languages [Arvind88], leave the detection of parallelism and mapping onto a parallel machine to the compiler. Unfortunately, contemporary compilers do not produce efficient translations for arbitrary algorithm-machine combinations. In turn, if a programmer would know the optimal mapping of an algorithm onto a certain architecture most implicit description languages do not provide facilities to express this mapping explicitly. An exception to this is described in [Hudak88].

†This research is partially funded by SPIN

Most languages based on explicit descriptions specify parallelism c.q. communication and synchronization as integral part of the algorithm. This has the disadvantage that one has to program multiple threads of control, which are generally very hard to debug [Karp88]. Hence, experimentation with different versions of the same parallel algorithm, for example different decompositions, is in general rather cumbersome. Comparably small changes may require major program restructuring.

In this paper, we describe a different explicit approach, that pairs the flexibility of the implicit with the expressiveness of the explicit. In the approach algorithm description and algorithm decomposition are described separately. Efficient SPMD (Single Program Multiple Data) [Karp87] code, in particular communication and synchronization, is generated automatically by the compiler. Furthermore, the compiler uses a model base of target architectures in order to optimize computation and communication efficiency.

The approach of inducing parallelism by explicitly decomposing the data is not new. In [Callahan88, Gerndt89, Kennedy89] applications to Fortran are described, in [Rogers89] to Id Nouveau, in [Koelbel87] to BLAZE, and in [Quinn89] to C*. In particular application to Fortran is limited, because of equivalencing, passing of array-subsections to subroutine calls, and any form of indirect addressing cannot be translated efficiently. A second limitation is that the description of complex decompositions and especially dynamic decompositions, i.e. a redistribution of the data at run-time, is not feasible either. An exception is [Kennedy89] where a method is presented to describe redistribution. However, this method still has the drawback that redistribution statements are intermingled with the program code, which limits portability.

A more fundamental problem to these approaches is that distinctive formalisms are used for the description of algorithm and decomposition. Hence, a unified formal system to reason about optimizing transformations at compile-time is not possible. Furthermore, the approaches do not address the issue in the general context of data representation. In our approach the way in which we "view" data on the algorithm level is essential. The approach is illustrated in Fig. 1. We specify the program P and the involved set of data structures D. By adding the data representation $D \varnothing$

D', the compiler automatically generates an equivalent program *P'* with the set of data structures *D'*.

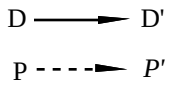

Fig 1. The approach to program translation.

Especially for sparse data structures the possibility to 'view' them as a compact structure is very convenient from the programmers point of view.

To illustrate our method, a high-level parallel programming language called *Booster* is introduced in Section 2. Programs written in *Booster* are translated to imperative languages, like (parallel) Fortran, C, ADA, or OCCAM. For Fortran and C, the code-generation is not restricted to distributed-memory computers, but code can also be generated for shared-memory- and vector computers. Section 3 describes the translation of *Booster* to SPMD-programs in terms of a so-called view calculus. In addition, a number of optimizations with respect to the translation are discussed. In Section 4, the generation of a number of alternative send/receive schemes are elaborated on and an example translation of a *Booster* program to both a distributed- and shared memory architecture is given. Finally, in Section 5 a brief description is given of the architecture of the *Booster* translator.

2. The *Booster* Language

In this section the *Booster* language is discussed informally. For a more extensive treatment the reader is referred to [Paalvast89a,b].

2.1. Basic concepts

In a conventional programming language (such as Fortran) the array is used as the basic data structure for storing large amount of related data elements. These elements can be accessed by the use of indices into 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 inserting (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 an array 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 may be executed in parallel. A comparable approach is followed in a language like ACTUS [Perrot87].

Data- and index domain
In *Booster*, these observations have resulted in a strict distinction between *data- and index domain*. 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.

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, accessed through indices. Unlike arrays, shapes need not necessarily be rectangular (for convenience we will, for the moment, assume that they are). The ordered set of all indices in the shape is called an index set. 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 returns 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 perspective, hence the name. Shapes are defined in *Booster* as follows:

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

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*}. 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)*}.

Content statements
In so-called *content statements* we can move and modify 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 10^{th} element of A, and so on.

Arithmetic Operators
Apart from standard scalar operators *Booster* also supports their multi-dimensional equivalents. For example, a vector A times a scalar 2 is written as:

```
A := A*2;
```

Application of these multi-dimensional operators is restricted to pairs of scalars and higher dimensional structures. Other operators can be specified with help of the `function` construct, which is discussed shortly.

View Statements
We 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. After this view statement the three content statements will have exactly the same effect.

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

Note that no additional data structure is created. This is typical of all views. Also note the different assignment symbols for view- and content-statements; '<-' for view statements and ':=' for content statements.

Modules and Functions
A *Booster* program consists of a set of modules, where each module has a number of input- and output-arguments. Within a module it is possible to encapsulate a number of view- and content statements in functions. *Booster* functions, like modules, do not have side effects and when a function has only one output argument and at most two input arguments it may be used as an infix operator in content statements. An example of a *Booster* function is the following:

```
FUNCTION
Vector_Mult PRIORITY 7 (V, W) -> (U);

V,W (n) OF REAL;
U (n # n) OF REAL;
BEGIN
   U[i,j] := V[i] * W[j];
END;
```

Here a new language construct is introduced, the *free variable* `i` and `j`. A treatment of the exact syntax and semantics of this construct is deferred to the next section. This function assigns the vector-vector product $V_i \cdot W_j$ to $U_{i,j}$. The keyword **`priority`** assigns a precedence to the function `Vector_Mult` which is used to decide evaluation order when the function is itself used as binary operator. Priority 7 corresponds with the priority of the '*' operator. E.g., `2 + X Vector_Mult Y = 2 + (X Vector_Mult Y)`

Control Flow
In addition to the content- and view-statements, *Booster* also offers several control-flow constructs similar to those found in conventional languages. Available are the **`IF`** statement for conditional execution, and **`WHILE`** and **`ITER`** statements for iteration purposes (**`ITER`**-loops execute a fixed number of times, **`WHILE`**-loops execute as long as a boolean condition evaluates *true*).

2.2. View classes

Views basically come in three flavours, each of which will be illustrated with a simple example.

Selection Views
The first non-trivial type of view we illustrate is the *selection* view:

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

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

Again, the two content statements are equivalent, given the view statement that precedes them. The *index expression* `5:15` selects the subset or *range* of indices *5* through *15* of `A`. After this statement, the identifier `V` can be used to access `A` through the index set {*0*, *1*, … *14*}. 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, like the original index set does. `A` itself is never affected by any view statement.

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

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

The following content statements are equivalent:

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

The free variable `i` is used to access the values of `A` through `V` in reverse order. In fact, permutation views are an efficient alternative to create high level indirect addressing.

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:

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

The following content statements are equivalent:

```
V[0,0] := V[2,3];
A[0]   := A[11];
```

The construct `(4#5)` explicitly specifies the index set that should result from the view, or, put in another way, the domains for the free variables `i` and `j`. In the permutation view statement given in the previous sections, the domain of `i` and hence the resulting index set could be deduced by the compiler from the declaration of `A`. Here, the compiler needs to be told how to partition the *20* elements of `A` over the two dimensions of `V`. The identifier `V` now becomes for all application purposes identical to a matrix shape. The fact that the index set of this matrix refers, *via* a view function, to a one-dimensional vector is completely hidden from the 'user' of `V`.

View Functions
Another example illustrates that selection views need not always select consecutive ranges. At this time we introduce the concept of a *view function*:

```
VIEW FUNCTION Even (V) -> (E);
V (n);
E (n div 2 + n mod 2);
BEGIN
   E[i] <- V[2*i];
```

```
END;
```

The view function is a way of encapsulating related view statements. Input and output arguments are specified and their index sets declared. 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.

Below a more complex view function is given, which uses the previously defined function `Even` and returns the even and odd elements of `A`.

```
VIEW FUNCTION Unzip (V) -> (E, O);
V (n);
E (n div 2 + n mod 2);
O (n div 2);
BEGIN
   E    <- V[Even];
   O[i] <- V[2*i+1];
END;

(E, O) <- Unzip(A); // Call in main program
```

Consequently, the three following content statements are equivalent:

```
E[5] := O[5];
A[10] := A[11];
A[Even][5] := A[11];
```

2.3. Example program

We will illustrate some of the concepts of *Booster* by means of the well-known Gaussian elimination algorithm (without pivoting). This example is also used to illustrate the decomposition technique in Section 2.4. Only those parts of the algorithm are described which are relevant to the discussion. The algorithm takes a previously declared non-singular *n x n* matrix `A` as input. The algorithm eliminates in successive steps each column until an upper triangular matrix results. In each step four selections are involved: the pivot element `E`, the pivot row `R`, the pivot column `C`, and the remainder `B` (Fig.2). These selections of the matrix `A` are defined by the view function `pivoting`. The generic `$`-symbol denotes the upperbound of the index set the view is applied to. In this case `$` equals `m-1`.

```
VIEW FUNCTION pivoting (Q) -> (E,R,C,B)
Q (m # m);
BEGIN
   E  <-  Q[0,0];
   R  <-  Q[0,1:$];
   C  <-  Q[1:$,0];
   B  <-  Q[1:$,1:$];
END;
```

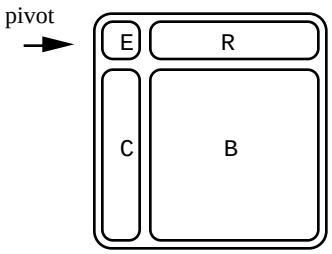

Fig.2 Representation of the view function `pivoting`

The algorithm is described in terms of these views:

```
H <- A;
WHILE  size(H) > 1 DO
   (E, R, C, B) <- pivoting(H);
   B := B - C Vector_Mult R / E;
   H <- H[1:$,1:$];
END;
```

In the initial step of the algorithm, the index set of `A` is assigned to the view identifier `H`. The remainder of the algorithm is coded as a conditional loop with three statements. In the first statement the views `E`, `R`, `C`, and `B` are defined by application of the view function `pivoting` to `H`. The content statement describes computation associated with the current views `E`, `R`, `C`, and `B`. The last statement re-defines `H` in terms of the previous view. This *view recursion* is illustrated in Fig. 3. The algorithm terminates if the size of view `H` is equal to 1.

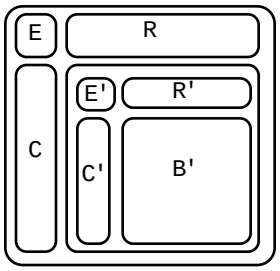

Fig.3 Representation of the repeated application of the view function `pivoting`

2.4. Mapping data and algorithms to (parallel) machines

Before introducing the mapping description formalism of *Booster*, we return to the shape and view concepts that have been introduced in Section 2. Shapes define the total amount of data-space needed for representing the values the algorithm operates upon. Shapes, however, need not necessarily be translated directly to equally dimensionalized data structures in the target languages. The programmer may influence the representation of shapes, by *relating* the actual representation in a virtual machine and the shape in question through a *view*. This principle is illustrated in Fig.4.

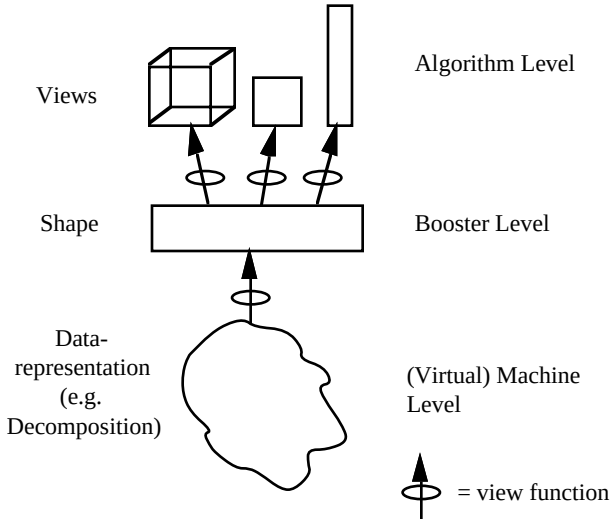

Fig. 4 Data representation

As an example, consider a shape `A` with index set `n#m` and two different mappings of this shape on a virtual machine:

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

Note that the shape `A` is regarded as a view on the resource `Lin_Mem` of the virtual machine. The 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
To obtain a partitioning of a shape, again the view concept can be used. If, for example, a two-dimensional shape is to be decomposed in a row-wise fashion for a parallel machine with *p* processors, this is described using the following view function depicted in Fig. 5.

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

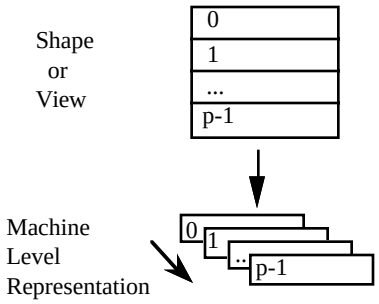

Fig. 5 Row-wise decomposition

Returning to the Gaussian elimination example possible mappings are:

```
    A <- row_decompose(M,p);
or
    H <- row_decompose(M,p);
```

The first view statement assigns parts of shape `A` to model `M` with `p` processors, effectively resulting in a *static* decomposition. The second view statement assigns parts of the *view* identifier `H` to the processor-model `M`. Note that with the decomposition on the view identifier we surpass the shape-level (see Fig 3.). The effect of this dynamic decomposition on `H`, on which all the current calculation is performed, is load-balancing. This in contrast to the first decomposition scheme in which an increasing number processors will become idle as the iteration proceeds. We will return to this in Section 4. The mapping of the algorithm to a machine-model can be specified on different levels of abstraction. These range from a global partitioning of shapes or views (as was shown above) or to a detailed, machine specific, mapping onto processors and memories.

3. The translation of *Booster*

3.1. Basic view calculus

Due to the inherent complexity of the operations supported by the *Booster* language, a so-called view calculus has been developed, providing a formal foundation for many types of compile-time optimizing transformations, necessary to ensure an efficient execution. Key aspect of this functional calculus is expression-level view arithmetic, by which the operational semantics of *Booster* expressions are defined through rewrite rules. In Section 3.1 only those subjects and techniques will be discussed which are necessary for a full understanding of the principles behind the data decomposition driven SPMD code generation method and its application to the Gaussian elimination example of Section 2.3. A more comprehensive treatment including e.g., vectorization issues can be found in [Gemund89].

Projection
The unary *projection function*, denoted $[i]$, $0 \le i \le n-1$, selects the element v_i when applied to the *1*-dimensional variable $V = (v_0, v_1, ..., v_{n-1})$. In contrast to the convention used in *Booster*, $[i]$ is now applied as a *function* and consequently we write $[i]V = v_i$, rather than $V[i]$. In the *d*-dimensional case, a projection is defined for each dimension. Let $A = ((a_{0,0}, ..., a_{0,m-1}), ..., (a_{n-1,0}, ..., a_{n-1,m-1}))$ be a *2*-dimensional variable. The two projection functions are $[i,\_]A = (a_{i,0}, ..., a_{i,m-1})$, and $[\_,j]A = (a_{0,j}, ..., a_{n-1,j})$.

Construction
The *vector construction function*, denoted *vec(i,a:b)*, is a function which, when applied to $[f(i)]V$, returns a *vector* of instances $([f(a)]V, [f(a+1)]V, ..., [f(b)]V)$, $b \ge a$, where $i,a,b,f(i) \in \mathbf{N}$. Observe, that the usual parentheses to denote the application of *vec(i,a:b)* to $[f(i)]V$ are omitted. This will be done throughout the sequel unless ambiguity arises. Multi-dimensional variables are constructed by vector nesting, like in the following $2 \infty 2$ matrix:

vec(i,0:1) vec(j,0:1) [*2*i+j*]*V* =
vec(i,0:1) ([*2*i*]*V*, [*2*i+1*]*V) =*
(([0]V, [1]V), ([2]V, [3]V))

Construction functions can also have predicates. The function *vec(i,a:b | P_i)* means that the instances parameterized by *i* are only generated if P_i evaluates *TRUE*. For instance, *vec(i,3:5) | i ≥ 4)* [*i*]*V* = *([4]V,[5]V)*.

Functional composition of vector constructors and projectors allows for the representation of any view- (or shape) expression. Both constructs form the basic elements in terms of which any compile-time transformation is defined.

View reduction

Consider the variable $V' = (v'_0, v'_1, v'_2)$, defined by *V'* = [*3:5*]*V*. Hence, $V' = (v_3, v_4, v_5)$, which is formally represented by *V'* = *vec(i,0:2)* [*i+3*]*V*. In effect, this implies the definition of [*3:5*]*V* by the rewrite rule

[*3:5*]*V* = *vec(i,0:2)* [*i+3*]*V*

Since $v'_j = v_{j+3}$, *j = 0,1,2*, the reference to an element [*j*]*V'* corresponds to the reference [*j*][*3:5*]*V* = [*j+3*]*V*. The *reduction* of the *composite vie*w [*j*][*3:5*]*V* = [*j+3*]*V* is formally stated by the following rewrite rule (where *f(i)=i+3*), i.e.

[*j*] *vec(i,a:b)* [*f(i)*]*V*= [*f(j+a)*]*V* , *j = 0, ..., b-a* *(3.1)*

Although (3.1) states the reduction rule for the *1*-dimensional case, similar rules for higher dimensions are easily derived. Consider the following *2*-dimensional reduction

[*j*,_] *vec(i,a:b)* [*f(i)*,_]*A* = [*f(j+a)*,_]*A* *(3.2)*

The view reduction in compile-time is fundamental to *Booster's* view programming concept: views can be thought of as *virtual data structures* on which, like shapes, any of the usual arithmetic operations are defined. The operational semantics of a *Booster* expression requires any *view* reference to be resolved by this technique to an irreducible *shape* reference.

In the multi-dimensional case, element referencing is formally expressed by composite projection. For example $[i,j]A = a_{i,j}$ can be written as [*j*][*i*,_]*A*. Such rewrite rules are useful in multi-dimensional reductions. For instance, consider the view [*transpose*]$A = A^T$, where [*transpose*] = *vec(i,0:$) vec(j,0:$)* [*j*,*i*] (as in *Booster*, *$* binds at application). Reduction of the *2-composite* $[transpose]^2$ = [*transpose*][*transpose*] yields (terms subject to reduction are bold faced)

vec(i,0:$) vec(j,0:$) **[*j,i*]***vec(u,0:$) vec(v,0:$)* [*v*,*u*] =
vec(i,0:$) vec(j,0:$) [*i*][*j*,_]*vec(u,0:$) vec(v,0:$)* [*v*,*u*]

Application of (3.2), followed by application of (3.1), yields

vec(i,0:$) vec(j,0:$)[*i*]**[*j*,_]*vec(u,0:$)v***ec(v,0:$)[*v*,***u***]=
vec(i,0:$) vec(j,0:$) **[*i*] *vec(v,0:$)*** [***v***,*j*] =

vec(i,0:$) vec(j,0:$) [*i*,*j*] *(3.3)*

Expression (3.3) equals the identity view, which can be cancelled altogether. Thus, the general reduction technique is to shift projections to the *right*, leading to the cancellation of constructors, as occurred twice in the previous derivation.

As reduction of *2*-composites might be performed as previously discussed, reduction of *k-composites* necessitates a more generic approach if *k* is not known at compile-time. For instance, the translation of the Gaussian elimination program given in Section 2.3 involves compile-time reduction of such a *k*-composite. In the process of resolving view recursion, the view statement

```
H <- H[1:$,1:$];
```

inside the loop is replaced by the non-recursive (intermediary language) sequence

```
H <- [1:$,1:$]^kA; k:=k+1;
```

where *k = 0, 1, ...* denotes an iteration counter generated in the process. As [*1:$*][*1:$*] = [*2:$*], it can easily be seen by induction that the *k*-composite $[1:\$,1:\$]^k$ reduces to [*k:$,k:$*]. Hence, the views *B*, *C*, *R*, and *E* in the content statement are eventually reduced to

B = [*k+1:$,k+1:$*]*A*, *C* = [*k+1:$,k*]*A*,
R = [*k,k+1:$*]*A*, *E* = [*k,k*]*A* *(3.4)*

Operator reduction

As mentioned in Section 2.1, *Booster* supports multi-dimensional operators. Their semantics are defined in terms of basic scalar operations through *operator reduction* rules. This type of rewrite rules are applied to transform expressions in terms of operators which are directly supported by the actual target architecture (e.g., scalar or vector operators). For example, consider the vector expression [*Even*] *(V + W)*, where *V* and *W* are *n* element vectors and [*Even*] is defined according to Section 2.2, i.e. [*Even*] = *vec(i,0:∪n/2⌉-1)* [*i*]. The following reduction applies

vec(i,0:∪n/2⌉-1) [*i*] *(V + W) =*
vec(i,0:∪n/2⌉-1) ([*i*]*V* + [*i*]*W)*

which *reduces* the vector addition to a scalar addition. Apart from multi-dimensional versions of the usual built-in operators (*'+'*, *'-'*, *'*'*, *'/'*, etc.), *Booster* supports the definition of user-defined operators (e.g., the `Vector_Mult` operator in Section 2.1). Note, that operator reduction automatically implements *drag-through*, i.e. sub-expressions like *(V + W)* in the previous example need not be evaluated prior to the application of [*Even*], since, by the same type of rewrite rule, it holds [*Even*] *(V + W)* = [*Even*]*V* + [*Even*]*W*, thus eliminating excessive computation and the need for intermediate storage.

Translation of *Booster* content statements

Formally, translation of a content statement *S* is based on a sequence of transformations applied to its corresponding *post condition* expression *P(S)*. As a typical example of

the transformation process, we consider translation of the content statement *S* inside the loop of the Gaussian elimination algorithm (Section 2.3), i.e.

```
B := B - C Vector_Mult R / E;
```

The post condition corresponding to this *2*-dimensional assignment is obtained by adding a *2*-dimensional identity view *vec(i,0:n-k-1) vec(j,0:n-k-1)* [*i,j*] (see (3.3)), and replacing the assignment by a relational *'='* operator (effectively turning the content statement into a *2*-dimensional expression with each element returning *true*), i.e.

$$vec(i,0{:}\$)\ vec(j,0{:}\$)\quad [i,j]\ (B = B - C\ Vector\_Mult\ R / E) \qquad (3.5)$$

Transformation of *P(S)* into *P(S')* is achieved by application of both view- and operator reductions, based on expression equivalence. The order of reductions is arbitrary. After transformation, code is generated for *S'* in which each vector constructor maps to an index loop constructor. The following sequence of operator reductions applies in case of scalar processing (operators subject to reduction are '=', '-', and *Vector_Mult*, respectively)

$$[i,j]\ (B = B - C\ Vector\_Mult\ R / E)$$
$$[i,j]\ B = [i,j]\ (B - C\ Vector\_Mult\ R / E)$$
$$[i,j]\ B = [i,j]\ B - [i,j]\ (C\ Vector\_Mult\ R / E)$$

Operator reduction of `Vector_Mult`, according to the operational *Booster* definition (Section 2.1), is given by

$$[i,j]\ (C\ Vector\_Mult\ R) = [i]C * [j]R$$

Hence, (3.5) reduces to

$$vec(i,0{:}\$)\ vec(j,0{:}\$)\quad [i,j]\ B = [i,j]\ B - [i]C * [j]R / E \qquad (3.6)$$

Substitution of (3.4) and subsequent *view* reduction finally yields the following post condition *P(S')*, i.e.

$$vec(i,0{:}n-k-2)\ vec(j,0{:}n-k-2)\quad \{[i+k+1,j+k+1]A = [i+k+1,j+k+1]A - [i+k+1,k]A * [k,j+k+1]A / [k,k]A\ \} \qquad (3.7)$$

which is mapped to the following imperative style pseudo code S':

```
for i := 0 to n-k-2 do
  for j := 0 to n-k-2 do
    A[i+k+1,j+k+1] = A[i+k+1,j+k+1] -
          A[i+k+1,k] * A[k,j+k+1] / A[k,k];
```

In general, the course of transformation depends on the available operations supported by the actual target architecture, and is guided by execution cost minimization. An example of such an optimizing transformation is vectorization, i.e. rewriting *P(S)* in terms of vector operators. Data decomposition, i.e. rewriting *P(S)* in terms of the actual memory structure of the target machine, affects the transformation process in a similar way. This is the main subject of the following sections.

3.2. Data decomposition

In *Booster* the shape is the basic data structure on which the algorithmic operations are performed, typically through the use of views. As discussed in Section 2.4 however, a general *d*-dimensional *shape*, in turn, is nothing but an abstraction of its actual *memory map*, which, in a uniprocessor, is 1-dimensional. Thus, a shape itself can be regarded as a *view* of its memory map. Consider the 2-dimensional shape *A* with index set `n # m`. Let A^M denote its memory map. Then the storage scheme can be expressed by the view function [*M*] according to $A = [M]A^M$. For instance, a column-wise storage scheme is represented by [*M*] = *vec(i,0:n-1) vec(j,0:m-1)* [$j \cdot m + i$]. If, for any reason, code generation is desired in terms of A^M, rather than *A*, the translation scheme will include *substitution* of *A* by $[M]A^M$ and subsequent view reduction, e.g., $[u,v]A = [u,v][M]A^M = [v \cdot m + u]A^M$.

The same kind of substitution and reduction scheme is used for the generation of SPMD code. Consider a distributed-memory machine with p_{max} *virtual processors*. Quite similar to the previous discussion, decomposition of a shape *A* is essentially a specification of its *distributed map* A^{DM} through specification of a decomposition scheme [*D*], i.e. $A = [D]A^{DM}$. For instance, consider the view function `row_decompose` (Section 2.4), where

$$[D] = vec(i,0{:}n-1)\ vec(j{:}0,n-1)\ [i\ div\ (n/p_{max}),\ i\ mod\ (n/p_{max}),\ j]$$

Contrary to the uniprocessor case, A^{DM} now represents a vector of p_{max} local memory maps, effectively defining the first index axis as *processor* axis and the remaining index space local to each processor. In principle, automatic parallel program generation is essentially based on a similar translation scheme as previously discussed, i.e. substitution of *A* by $[D]A^{DM}$ and subsequent reduction.

3.3. SPMD generation concept

We present the concept behind the automatic SPMD code generation scheme based on the translation of a simple 1-dimensional *Booster* content statement. The scheme is easily generalized to more complex operations. Consider translation of the following, irreducible, post condition

$$vec(q,q_{min}{:}q_{max})\ \{\ [f(q)]V = Expr([g(q)]W)\ \} \qquad (3.8)$$

where *V* and *W* are 1-dimensional shapes and *Expr* is some arithmetic expression of which $[g(q)]W$ is a term. Let *V* and *W* be partitioned according to the data decompositions (vector lengths $|V|$ and $|W|$)

$$V = vec(i,0{:}|V|-1)\ [\pi_V(i),\lambda_V(i)]V^{DM}$$

$$W = vec(i,0:|W|-1)\ [\pi_W(i),\lambda_W(i)]W^{DM} \qquad (3.9)$$

where $\pi_V(i)$, $\pi_W(i)$, *and* $\lambda_V(i)$, $\lambda_W(i)$ denote processor indices and local memory indices, respectively. After substitution in (3.8), the post condition becomes

$$vec(q,q_{min}:q_{max})$$
$$\{\ [\pi_V(f(q)),\lambda_V(f(q))]V^{DM} =$$
$$Expr([\pi_W(g(q)),\lambda_W(g(q))]W^{DM})\ \} \qquad (3.10)$$

The most straightforward translation scheme would be directly based on (3.10). Optimizations will be discussed in the next section. The usual convention that each processor is responsible for the production of its *own* local data, implies that each processor p traverses the entire index space q_{min} ... q_{max} and *only* performs the calculation on the condition $\pi_V(f(q)) = p$. The corresponding transformation of (3.10) is realized by addition of a $vec(p,0:p_{max}-1)$ constructor (which reflects the parallel computation) combined with a predicate $\pi_V(f(q)) = p$, i.e.

$$vec(p,0:p_{max}-1)\ vec(q,q_{min}:q_{max} \mid \pi_V(f(q)) = p)$$
$$\{\ [p,\lambda_V(f(q))]V^{DM} =$$
$$Expr([\pi_W(g(q)),\lambda_W(g(q))]W^{DM})\ \} \qquad (3.11)$$

Note, that, despite the increase in dimension, the semantics of (3.11) remains the same as exactly the same index references are generated.

For a distributed-memory model of computation, the next step is to express (3.11) in terms of local data. Let $V^L{}_p$ denote the partition of V^{DM} local to processor p, such that $V^{DM} = vec(p,0:p_{max}-1)V^L{}_p$. Let $W^L{}_p$ be defined similarly. Then

$$[p,\lambda_V(f(q))]V^{DM} = [\lambda_V(f(q))]V^L{}_p \qquad (3.12)$$

and (using a common functional notation)

$$[\pi_W(g(q)),\lambda_W(g(q))]W^{DM} =$$
$$\quad if\ (\pi_W(g(q)) = p)$$
$$\quad then\ [\lambda_W(g(q))]W^L{}_p$$
$$\quad else\ fetch(\pi_W(g(q)),\lambda_W(g(q))) \qquad (3.13)$$

where the *fetch* function returns the element with local index $\lambda_W(g(q))$ residing at processor $\pi_W(g(q))$. Note, that for the purpose of this introductory discussion a message-passing scheme is assumed in which communication is initiated by calculating processors issuing the *fetch* call. A full discussion on the generation of message-passing primitives will be deferred until Section 4.

By substitution of (3.12) and (3.13), (3.11) maps to the following imperative style SPMD pseudo code for all processors $p = 0, \ldots, p_{max}-1$, where p is assumed to be provided by the run-time function `myself`:

```
p := myself;
for q := qmin to qmax do
  if πV(f(q)) = p then
    if πW(g(q)) = p then
      VL[λV(f(q))] := Expr(WL[λW(g(q))]);
    else
      VL[λV(f(q))] :=
           Expr(fetch(πW(g(q)),λW(g(q))));
```

Note, that transformation to a *shared-memory* model of computation is even more straightforward. Back-substitution of (3.9) in (3.10) yields the following (worker) code

```
p := myself;
for q := qmin to qmax do
  if πV(f(q)) = p then
    V[f(q)] := Expr(W[g(q)]);
```

The applicability of the SPMD generation method presented to shared-memory machines, remains throughout the sequel.

3.4. Optimization

As mentioned in the previous section, code generation based on (3.11) is straightforward but not very efficient. All processors traverse the entire index space testing the condition $\pi_V(f(q)) = p$, whereas the number of references per individual processor might be reduced by a factor p_{max}, assuming a fair partitioning. Let Q_p denote the exact set of indices to be covered by processor p, i.e.

$$Q_p = \{\ q \mid \pi_V(f(q)) = p,\ q_{min} \le q \le q_{max}\ \}$$

Let the *distribution function* $\theta_p(t)$ be chosen such, that it maps a *consecutive* index domain $t = t_{p,min}$ ... $t_{p,max}$ to Q_p. Then (3.11) can be reduced to

$$vec(p,0:p_{max}-1)\ vec(t,t_{p,min}:t_{p,max})$$
$$\{\ [p,\lambda_V(f(\theta_p(t)))]V^{DM} =$$
$$Expr([\pi_W(g(\theta_p(t))),\lambda_W(g(\theta_p(t)))]W^{DM})\ \} \quad (3.14)$$

Thus, all the run-time evaluation overhead for generating the proper index set Q_p can be avoided if the *distribution parameters* θ_p, $t_{p,min}$, $t_{p,max}$ are known at compile-time. For block-decompositions and scatter-decompositions the following two theorems illustrate the conditions for which such an optimization is obtained.

Theorem 1:
Let V^{DM} be a block-decomposition of V according to

$$V = vec(i,0:|V|-1)\ [\pi_V(i),\lambda_V(i)]V^{DM}$$

where $\pi_V(i) = i\ div\ c$, $\lambda_V(i) = i\ mod\ c$, and c denotes the maximum block size per processor. If $f(q)$ is a monotonic

increasing function in q, then the distribution parameters are given by

$$\theta_p(t) = t,$$
$$t_{p,min} = \text{MAX}\{q_{min}, \cup f^{-1}(c \cdot p)\rceil\},$$
$$t_{p,max} = \text{MIN}\{q_{max}, \in f^{-1}(c \cdot p + c - 1)\rfloor\} \qquad (3.15)$$

Proof: The condition $\pi_V(f(q)) = p$ implies $f(q)\ div\ c = p$. Hence, the range for $f(q)$ is given by $c \cdot p \le f(q) \le c \cdot p + c - 1$, which yields $q = \cup f^{-1}(c \cdot p)\rceil \ldots \in f^{-1}(c \cdot p + c - 1)\rfloor$ iff f is monotonic. Since this set of indices is consecutive, it simply follows that $\theta_p(t) = t$. Hence, $t_{p,min}$ and $t_{p,max}$ directly follow from $q_{min} \le t \le q_{max}$.
[]

The theorem is also valid for monotonic decreasing functions $f(q)$, provided that the arguments of f^{-1} are exchanged for $t_{p,min}$ and $t_{p,max}$.

Theorem 2:
Let V^{DM} be a scatter-decomposition of V according to

$$V = vec(i,0{:}|V|-1)\ [\pi_V(i), \lambda_V(i)]V^{DM}$$

where $\pi_V(i) = i\ mod\ p_{max}$, $\lambda_V(i) = i\ div\ p_{max}$. If f is given by $f(q) = a \cdot q + b$, $a \ge 0$, the distribution parameters are given by

$$\theta_p(t) = \psi_{min} + (p_{max}/gcd(a,p_{max})) \cdot t,$$
$$t_{p,min} = 0,$$
$$t_{p,max} = \in (q_{max} - \psi_{min})/(p_{max}/gcd(a,p_{max}))\rfloor \qquad (3.16)$$

where ψ_{min} is the minimal solution in q of the linear diophantine equation $a.q - p_{max} \cdot k = p - b$, given the constraint $q \ge q_{min}$ and gcd denotes the greatest common divisor. If no solution to the diophantine equation exists, then no optimization can be achieved.

Proof: The condition $\pi_V(f(q)) = p$ implies $f(q)\ mod\ p_{max} = p$. Since $f(q) = a \cdot q + b$ this yields the linear diophantine equation $a \cdot q - p_{max} \cdot k = p - b$ in the variables q and k. Let ψ_{min} denote the minimal solution in q given the constraint $q \ge q_{min}$ (to be obtained by some usual iterative technique). Then the general solution $q_t = \theta_p(t)$ is given by $\theta_p(t) = \psi_{min} + (p_{max}/gcd(a,p_{max})) \cdot t$. The range $t_{p,min} \le t \le t_{p,max}$ follows from the constraint $q_{min} \le \theta_p(t) \le q_{max}$. By definition of ψ_{min} and $\theta_p(t)$ it follows that $t_{p,min} = 0$. Finally, with respect to $t_{p,max}$ the constraint $\theta_p \le q_{max}$ yields $t_{p,max} = \in (q_{max} - \psi_{min})/(p_{max}/gcd(a,p_{max}))\rfloor$.
[]

For cases in which $f(q) = q$, ψ_{min} has an analytical solution, i.e. $\psi_{min} = \cup (q_{min} - p)/p_{max}\rceil$. Hence, it holds that $\theta_p(r) = p + p_{max} \cdot t$ as $a = 1$ and it also holds that $t_{p,min} = 0$ and $t_{p,max} = \in (q_{max} - p)/p_{max}\rfloor$.

As discussed in Section 3.3, the convention that each processor is responsible for the production of its own local data, i.e. the *produces* $[\pi_V(f(q)), \lambda_V(f(q))]V^{DM}$, implies testing the *produce condition*, i.e. $\pi_V(f(q)) = p$. As illustrated by Theorem 1 and 2, under certain conditions this test could be performed at compile-time. Clearly the same technique applies to the test whether the *uses*, i.e. $[\pi_W(g(q)), \lambda_W(g(q))]W^{DM}$, are stored local or are to be fetched from another processor. Again, under the same conditions, the *use condition*, i.e. $\pi_W(g(q)) = p$, can be evaluated at compile-time, resulting in a simple index set membership test. This point will be elaborated in Section 4.4.

4. Generation of send/receive statements

For the purpose of the introductory discussion in Section 3.3, the *fetch* primitive has been introduced in which communication is solely initiated by the calculating processor when the *use* condition $\pi_W(f(q)) = p$ evaluates false. In general however, message-passing architectures usually provide means of communication through a *send/receive* scheme. Starting point for the following discussion is (3.10), i.e.

$$vec(q, q_{min}{:}q_{max})$$
$$\{\ [\pi_V(f(q)), \lambda_V(f(q))]V^{DM} =$$
$$Expr([\pi_W(g(q)), \lambda_W(g(q))]W^{DM})\ \} \qquad (4.1)$$

where for simplicity it is assumed that V and W are decomposed in the same number of partitions, i.e. p_{max}.

4.1. Receive scheme

Similar to a *fetch* scheme described in Section 3.3, generation of *receive* calls is based on testing the *use* condition, i.e. $\pi_W(g(q)) \ne p$, as a result of the convention that each processor is responsible for a *produce* condition, i.e. updating those elements of V^{DM} for which $\pi_V(f(q)) = p$. Hence, the transformation of (4.1) equals the post condition described by (3.11), where in (3.13) the *fetch* call is replaced by a *receive* call returning the element with local index $\lambda_W(g(q))$ residing at processor $\pi_W(g(q))$, i.e.

$$[\pi_W(g(q)), \lambda_W(g(q))]W^{DM} =$$
$$if\ (\pi_W(g(q)) = p)$$
$$then\ [\lambda_W(g(q))]W^L{}_p$$
$$else\ receive(\pi_W(g(q)), \lambda_W(g(q))) \qquad (4.2)$$

4.2. Send scheme

Contrary to the above scheme, however, one might have considered an alternative convention, in which each proces-

sor is responsible for a *use* condition, i.e. processing those indices q for which $\pi_W(g(q)) = p$ (those elements of W^{DM}, which are local to p). In such a complementary scheme, (4.1) would have become

$$vec(p,0{:}p_{max}\text{-}1)\ vec(q,q_{min}{:}q_{max} \mid \pi_W(g(q)) = p)$$
$$\{\ [\pi_V(f(q)),\lambda_V(f(q))]V^{DM} = Expr([p,\lambda_W(g(q))]W^{DM})\ \} \qquad (4.3)$$

in which the V^{DM} and W^{DM} references would be expressed in local terms according to

$$[\pi_V(f(q)),\lambda_V(f(q))]V^{DM} = \text{if } (\pi_V(f(q)) = p) \text{ then } [\lambda_V(f(q))]V^L{}_p \text{ else } send(\pi_V(f(q)),\lambda_V(f(q))) \qquad (4.4)$$

$$[p,\lambda_W(g(q))]W^{DM} = [\lambda_W(g(q))]W^L{}_p \qquad (4.5)$$

This corresponds with the fact that *send* calls would have to be issued when the *produce* condition $\pi_V(f(q)) = p$ evaluates false. With respect to the semantics of the *send function* in (4.4) it holds, that a post condition

$$send\ (\pi_V(f(q)),\lambda_V(f(q))) = \lambda_W(g(q))$$

implies $\lambda_W(g(q))$ to be stored at local element $\lambda_V(f(q))$ residing at processor $\pi_V(f(q))$, (i.e. *send* may be looked upon as an inter-processor "access" function).

4.3. Transformation for send/receive scheme

Consequently, a *send/receive* scheme the indices which are to be processed according to the original convention (Section 4.1) must now *include* the index range derived according to the alternative scheme (Section 4.2). Each processor must issue a *receive* call on a false *use* condition and a *send* call on a false *produce* condition, i.e. when the complementary processor will issue the corresponding *receive* call. As a result, the post condition for a combined *send/receive* scheme becomes

$$vec(p,0{:}p_{max}\text{-}1)\ vec(q,q_{min}{:}q_{max} \mid \pi_V(f(q)) = p \approx \pi_W(g(q)) = p)$$
$$\{\ [\pi_V(f(q)),\lambda_V(f(q))]V^{DM} = Expr([\pi_W(g(q)),\lambda_W(g(q))]W^{DM})\ \} \qquad (4.6)$$

in which the V^{DM} and W^{DM} references are to be substituted according to (4.4) and (4.2), respectively. The communication system is assumed to provide non-blocking `send` and blocking `receive` primitives. Furthermore, we will assume existence of a *procedural* version for the *send* primitive, i.e. `send(`π_V`(f(q)),`λ_V`(f(q)),`λ_W`(g(q)))`, which sends $\lambda_W(g(q))$ to processor $\pi_V(f(q))$ where it is received as element $\lambda_V(f(q))$. As a result, (4.6) maps to the following pseudo code

```
p := myself;
for q := qmin to qmax do begin
  if πV(f(q)) = p then
    if πW(g(q)) = p then
      VL[λV(f(q))] := Expr(WL[λW(g(q))]);
    else
      VL[λV(f(q))] :=
          Expr(receive(πW(g(q)),λW(g(q))));
  else
    if πW(g(q)) = p then
      send(πV(f(q)),λV(f(q)),WL[λW(g(q))]);
    else
      /* idle */ ;
end;
```

4.4. Optimization

Similar to the discussion in Section 3.4, optimizations are possible in which the index range per individual processor is minimized, and, as an additional optimization, the *use* and *produce* conditions for receives and sends are reduced at compile-time to index set membership tests. In a *send/receive* scheme however, additional require-ments arise with respect to the *order* in which the individual indices are to be generated if deadlock is to be avoided.

Let $R_p = \{\ q \mid \pi_V(f(q)) = p,\ q_{min} \le q \le q_{max}\ \}$ and $S_p = \{\ q \mid \pi_W(g(q)) = p,\ q_{min} \le q \le q_{max}\ \}$ denote the minimum set of indices q be processed by processor p in a receive- or send scheme, respectively. Note, that the members as well as the size of R_p may differ significantly from S_p, as the decompositions of V and W and the functions f and g will generally not be the same. Then for each processor p, execution is according to the following scheme:

for all indices $q \in (R_p \approx S_p)$
- issue *send* call: if $q \in (S_p \setminus R_p)$
- issue *receive* call: if $q \in (R_p \setminus S_p)$
- compute: if $q \in R_p$ (4.7)

Let the distribution parameters $\theta_{R,p}$, $r_{p,min}$, $r_{p,max}$ and $\theta_{S,p}$, $s_{p,min}$, $s_{p,max}$ be given with regard to the generation of R_p and S_p respectively. In order to avoid deadlock, an additional requirement with respect to the functions θ is, that the sequence in which R_p and S_p are generated, must equal the order by which R_p and S_p are defined (set members are added on a *use* or *produce* condition as q varies from q_{min} to q_{max}). Note, that the code presented so-far is free of deadlock, as for each index q for which the *produce* and *use* do not reside at the same processor, there always exist two complementary processors issuing the corresponding *send* and *receive* call. With respect to code generation, a second requirement to prevent deadlock is that (non-blocking) *send* tests be generated before (blocking) *receive* tests.

If the above distribution parameters can be derived at compile-time, code generation is straightforward. For example, consider code generation for block-decomposition, where f and g are monotonic increasing functions. From Theorem 1 it can be easily verified that the distribution function θ satisfies the requirements to prevent deadlock. The following code is based on scheme (4.7) where u implements the union $R_p \approx S_p$:

```
p := myself;
for u := 0 to MAX(r_p,max-r_p,min, s_p,max-s_p,min)
do begin
  r := u + r_p,min;
  s := u + s_p,min;
  if s_p,min ≤ s ≤ s_p,max and not
      r_p,min ≤ s ≤ r_p,max then
      send(π_V(f(s)),λ_V(f(s)),W[λ_W(g(s))]);
  if r_p,min ≤ r ≤ r_p,max then
    if not s_p,min ≤ r ≤ s_p,max then
      V[λ_V(f(r))] :=
           Expr(receive(π_W(g(r)),λ_W(g(r))));
    else
      V[λ_V(f(r))] := Expr(W[λ_W(g(r))]);
  end;
end;
```

4.5. Alternative communication schema

Although correct, the previous scheme has a serious flaw: it may lead to pure sequential processing. To see this, one must notice that given a loop instance, not all required *sends* are issued. If, for instance, all *uses* are allocated to the same processor, only one *send* is issued per cycle. Hence, only a single production can proceed per computation cycle. Three alternative send/receive schemes will be briefly discussed. Without loss of generality, we assume block-decomposition on both V and W and monotonic increasing functions f and g.

Buffered receive scheme
In such a scheme, all possible *sends* are issued first. The *receives* are issued when actually needed in the computation. The following demonstrates the principle

for all indices $s \in (S_p \setminus R_p)$
  - issue *send* call
for all indices $r \in R_p$
  - issue *receive* call when $r \notin S_p$
  - compute

Since all possible *sends* are issued before any computation takes place, this can place a burden on the communication system, as the *sends* which are not yet received take up buffer space. Note, that all *receives* could also be executed before any actual computation takes place. This however still implies the necessity of local buffer space.

Unbuffered receive scheme
In the previous section all possible *sends* are issued before any computation takes place. To avoid the necessity of buffering, transmission of data can be deferred to the time when it is actually needed for computation. To solve this we must find out which processor needs a *use*. Hence, we must search for other processors at the same relative position of r, i.e. r' and test *their use* conditions. This scheme is accomplished by solving $y = \pi_V(f(r'))$ from the equation $p = g((r_{y,min} + r - r_{p,min}) \text{ div } c_V$, either symbolically at compile-time or with a loop at run-time. When $y \neq p$ the *send* must be issued.

Block-send and -receive scheme
Some message-passing systems support block-send and -receive primitives in order to speed up data transfers. Let `blk_send(x,λ_V(f(s)),W[λ_W(g(s))],s|s_x,min:s_x,max)` denote block-send, where $s_{x,min}{:}s_{x,max}$ denotes the local range of the block, which is to be transmitted to processor x. For block-receive a similar primitive applies. Let c_V, c_W denote the decomposition block sizes for V and W, respectively. Similar to the proof of Theorem 1, the index range $s = s_{x,min} \ldots s_{x,max}$ for which $\pi_V(f(s)) = x$, is given by $s_{x,min} = \cup f^{-1}(c_V \cdot x)\rceil$, $s_{x,max} = \in f^{-1}(c_V \cdot x + c_V - 1)\rfloor$. The range of processors to which block-sends apply is given by $f(s_{p,min}) \text{ div } c_V \leq x \leq f(s_{p,max}) \text{ div } c_V$. For the block-receive a similar derivation holds. Program generation is then straightforward. However, often block-sends and block-receives are more easily realized when a variable has a higher dimension, as will be shown in the Gaussian elimination example in the next section.

4.6. The Gaussian elimination example

As an example, the SPMD generation method is applied to the translation of the *content statement* of the Gaussian elimination algorithm for a scalar parallel architecture. First a translation is shown for a distributed-memory model with the static block-decomposition (Section 2.4)

```
A <- row_decompose(M,p_max);
```

Second, a translation is shown for a shared-memory model with the dynamic block-decomposition

```
H <- row_decompose(M,p_max);
```

Distributed-memory with static decomposition
Recall post condition (3.7), i.e.

$$vec(i,0{:}n-k-2)\ vec(j,0{:}n-k-2)\ \{[i+k+1,j+k+1]A = [i+k+1,j+k+1]A - [i+k+1,k]A * [k,j+k+1]A / [k,k]A\ \} \quad (4.8)$$

Substituting $f_B(i) = i+k+1$, $g_B(i) = g_C(i) = i+k+1$, $g_R(i) = g_E(i) = k$, we consider the post condition only in terms of these functions in the variable i, i.e.

$$vec(i,0{:}n-k-2)$$

$\{... f_B(i) ... = ... Expr(g_B(i), g_C(i), g_R(i), g_E(i)) ... \}$

Application of `row_decompose` is expressed as

$A = vec(i,0:n\text{-}1)\ [i \text{ div } c, i \bmod c, \_]M,\ c = n \text{ div } p_{max}$

where $M = vec(p,0:pmax\text{-}1)\ L$ (L being a local data partition). Translation is given for the buffered *receive* scheme (Section 4.5). Although Theorem 1 is defined for the one-dimensional case, extension to the two-dimensional case is trivial, which is demonstrated by applying the theorem to the derivation of R_p (note that f_B is monotonic), this yields

$r_{p,min} = MAX(0, c{\cdot}p\text{-}k\text{-}1),$
$r_{p,max} = MIN(n\text{-}k\text{-}2, c{\cdot}p+c\text{-}1\text{-}k\text{-}1)$

With regard to the *use* terms g_B, g_C, g_R, g_E, the appropriate index sets are given by

g_B, g_C: $S_p = R_p$ (as $g_B, g_C = f_B$) *(4.9)*
g_R, g_E: $S_p = 0{:}n\text{-}k\text{-}2$ (no optimization) *(4.10)*

(4.9) implies that no sends have to be generated for these terms ($S_p \setminus R_p = \varnothing$). The full index range in (4.10) comes from the fact that Theorem 1 does not apply to constant functions. Hence, with respect to the sends, in principle the full index range is generated, i.e. $s_{p,min} = 0$, $s_{p,max} = n\text{-}k\text{-}2$, combined with the usual predicate test. As an additional (further) optimization, block-sends and -receives, are used to implement the j range (combining the E and R data). This leads to the following pseudo code (ignoring various further optimizations):

```
for s := s_p,min to s_p,max do   // full range
  if p = k div c then            // --> S_p
    if p ≠ s+k+1 div c then      // --> S_p \ R_p
      blk_send(s div c,
               L[s mod c,j],j|k:n);

for r := r_p,min to r_p,max do   // --> R_p
  if p ≠ k div c then            // --> R_p \ S_p
    begin
    Temp := blk_recv(k div c,
                     L[k mod c,j],j|k:n);
    for j := 0 to n-k-2 do
      L[(r+k+1) mod c,j+k+1] :=
      L[(r+k+1) mod c,j+k+1] -
      L[(r+k+1) mod c,k] *
      Temp[r+k+1] / Temp[0];
    end
  else
    for j := 0 to n-k-2 do
      L[(r+k+1)mod c,j+k+1] :=
      L[(r+k+1)mod c,j+k+1] -
      L[(r+k+1)mod c,k] *
      L[k mod c,j+k+1] /
      L[k mod c,k];
```

Shared-memory with dynamic decomposition

As the row-decomposition is now in terms of the (dynamic) *view* H, rather than the shape A, we will consider post condition (3.6), which is not yet reduced in terms of A, i.e.

$vec(i,0:\$)\ vec(j,0:\$)$
$[i,j]\ B = [i,j]\ B - [i]C * [j]R / E$ *(4.11)*

As H is not reduced in terms of A, the recursive view statement `H <- H[1:$,1:$];` inside the loop (see Section 3.1) is directly reduced in terms of M, effectively resulting in the k-parameterized view definition

$H_k = vec(i,0:n\text{-}k\text{-}1)\ [i \text{ div } c_k, i \bmod c_k, \_]M$ *(4.12)*

where k denotes the iteration counter and $c_k = (n\text{-}k)/p_{max}$ denotes the block size, now decreasing with k. Since $B = [1{:}\$,1{:}\$]H_k$, $C = [1{:}\$,0]H_k$, $R = [0,1{:}\$]H_k$, and $E = [0,0]H_k$, (4.11) becomes

$vec(i,0:n\text{-}k\text{-}2)\ vec(j,0:n\text{-}k\text{-}2)$
$\{ [i+1,j+1]H_k = [i+1,j+1]H_k - [i+1,0]H_k *$
$[0,j+1]H_k / [0,0]H_k \}$ *(4.13)*

Transformation is similar to the one discussed in the previous section. For a shared-memory model, the post condition is expressed in terms of H_k rather than M as discussed in Section 3.3. Optimization by application of Theorem 1 yields the (k-dependent) index set $R_{p,k}$, i.e.

$r_{p,k,min} = MAX(0, c_k{\cdot}p\text{-}1),$
$r_{p,k,max} = MIN(n\text{-}k\text{-}2, c_k{\cdot}p+c_k\text{-}1)$

and (4.13) transforms to:

$vec(p,0:p_{max}\text{-}1)vec(r,r_{p,k,min}{:}r_{p,k,max})\ vec(j,0:n\text{-}k\text{-}2)$
$\{ [r+1,j+1]H_k = [r+1,j+1]H_k - [r+1,0]H_k *$
$[0,j+1]H_k / [0,0]H_k \}$ *(4.14)*

Since $H_k = [k{:}\$,k{:}\$]A$, the following pseudo code is generated, where a barrier-call is used to synchronize between the iterations k:

```
for r := r_p,k,min to r_p,k,max do  // --> R_p
    for j := 0 to n-k-2 do
      A[r+k+1,j] := A[r+k+1,j] - A[r+k+1,k]*
                    A[k,j]/A[k,k];
barrier;
```

5. Implementation of the Translator

We are currently working on a prototype implementation of the translator for *Booster*, which is based on the principles discussed above. As translation of *Booster* programs is in fact a target architecture dependend reduction sequence, AI-techniques are used to implement the translator. A similar approach is described in [Wang89]. The course of inference, i.e. reduction, within this rule-based system is di-

rected by a performance measures derived from the target architecture. Each architectural model is a virtual machine description (e.g. Fortran-level) including high level operators, such as library-calls and high level communication functions, all with associated performance models. The last step of the translation process is the generation of a parallel program in one of the target languages with addition of the correct language extension appropriate to the specific target dialect (for examples see [Karp88]).

6. Conclusion

We have presented a general method for automatically deriving efficient SPMD-code for distributed- as well as shared-memory processors, given an algorithm and data decomposition description. The translation method as well as the algorithm and decomposition description formalisms are based on the application of the so-called view concept with its associated calculus. To this purpose, the high-level view programming language *Booster* is introduced, illustrating its merits by an example of its translation to distributed- and shared-memory architectures.

Our method can also be applied to other languages as well. As can be seen from the method, a number of criteria can be formulated with respect to maximum achievable speed-up when translating programming languages, based on data partitioning. If the functions *f(q)* and *g(q)* and their properties, such as linearity and monotonicity, are known, efficient translations can be generated for block- and scatter-decompositions. Traditional languages such as Fortran introduce serious problems regarding general decompositions. From a given Fortran program the functions *f(q), g(q),* etc., including the appropriate ranges for *q,* have to be extracted from the source code, which can be quite complex, if not impossible. Dynamic decompositions appear even more cumbersome, since all releveant information on the dynamic decomposition is scattered throughout the program. In this respect, functional languages are more promising, provided that a strict separation between index- and data manipulations can be made (possibly through annotations). The *Booster* language, as described in this paper, meets both approaches half in between.

Further research will be directed to extending the calculus to a true intermediate formal framework for the description of optimizations like vectorization and parallelization, based on machine models described within the same framework. Although in the current approach, data decomposition is supplied by the user, future research will also focus on incorporating decompositions as integral part of the architecture driven translation process which is described in Section 5.

References

[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.

[Chen88] M.C. Chen, Y. Choo, J. Li, "Compiling Parallel Programs by Optimizing Performance," *Journal of Parallel and Distributed Computing*, Vol. 5, No. 5, October 1988.

[Gemund89] A.J.C. van Gemund, *A View Language and Calculus*, Report no. 89 ITI B 46, TNO Institute of Applied Computer Science (ITI-TNO), Delft, The Netherlands, February 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, pp. 54-61.

[Hudak89] Hudak P. "Conception, Evolution, and Application of Funtional Programming Languages," *ACM Computing Surveys*, Volume 21, Number 3, September 1989, pp. 359-411.

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

[Karp88] A.H. Karp, R.G. Babb II, "A Comparison of 12 Parallel Fortran Dialects," *IEEE Software*, September 1988, pp. 52-67.

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

[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.

[Paalvast89a] E.M.R.M. Paalvast, *The Booster Language*, Technical Report, no. 89 ITI B18, TNO Institute of Applied Computer Science (ITI-TNO), July 1989, Delft, The Netherlands.

[Paalvast89b] E.M.R.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.

[Perrott87] R.H. Perrott, R.W. Lyttle, P.S. Dhillon, "The Design and Implementation of a Pascal-Based Language for Array Processor Architectures,"

Journal of Parallel and Distributed Computing, 1987, pp. 266 - 287.

[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.

[Quinn89] M.J. Quinn, P.J. Hatcher, *Data Parallel Programming on Multicomputers*, Parallel Computing Laboratory, Department of Computer Science, University of New Hampshire, report number PCL-89-18, March 1989, 16 pp.

[Wang89] K-Y. Wang , D. Gannon, "Applying AI Techniques to Program Optimization for Parallel Computers," In: *Parallel Processing for Super-computers and Artificial Intelligence*, H. Kwang, D. DeGroot, Eds. McGraw-Hill, 1989, pp. 441-485.