Solving Large Systems of Differential Equations in Parallel Using Covers and Skeletons

M. Südholt[†], *C. Piepenbrock*[‡], *K. Obermayer*[‡] *and P. Pepper*[‡]

[†]*Projet Lande, INRIA/IRISA-Rennes, France,* **sudholt@irisa.fr**
[‡]*Fachbereich Informatik, Institut für Kommunikations- und Softwaretechnik, TU Berlin, Germany,* **{piep, oby, pepper}@cs.tu-berlin.de**

Abstract
The design and implementation of parallel algorithms for distributed memory architectures is much harder than the development of sequential algorithms. This is mainly due to the communication and synchronization that is necessary to manage distributed data correctly. This paper applies a methodology for the transformational derivation of parallel programs using data distribution algebras that enable an abstract description of data distribution issues. Algorithms are formulated using skeletons, that is, specialized higher-order functions with particular parallel implementations. The methodology is applied to a the solution of a system of ordinary differential equations where convolutions can be computed using the Fast Fourier transformation. The example illustrates the practical optimization problems for a development model of the visual system that involves large scale neural network simulations. Finally, this algorithm is compared to an implementation of the same system of equations in the programming language C* on a CM-5.

Keywords
Functional programming, parallel programming, numerical algorithm, program transformation, skeleton, data distribution algebra

1 INTRODUCTION

The numerical solution of large systems of differential equations is one of the most challenging problems in the field of scientific computing. For many problems such systems are so large that only parallel computers possess the computational power necessary to solve the equations. Unfortunately, efficient parallel algorithms cannot simply be developed as extensions of their sequential counterparts. The spatial aspect of parallel algorithms—that is, the distribution of processes and data across the processing units of the parallel computer—constitutes a salient characteristic that is not encountered in the design of sequential algorithms.

Parallel algorithms that operate on distributed data have to communicate data at suitable synchronization points, which means that the temporal aspects of parallel algorithms are also much more complex than in the sequential case. Parallel programming in general is therefore much more error-prone, and it is considerably more difficult to verify and validate parallel than sequential algorithms. Moreover, the (commercial) availability of a multitude of parallel architectures complicates the development of portable parallel methodologies. Automatic program derivation by program transformation promises to yield a solution: transform high-level abstract specifications manually/automatically via less abstract intermediate specifications to efficient executable programs.

We see our approach as a step towards a compiler that performs an automatic transformational development of parallel algorithms in a functional setting. The essential ideas can be characterized as follows. We emphasize the distribution aspect from the very beginning of the design. The basic catchwords for this part of our approach are **data distribution algebras** describing so-called **covers**. Both the spatial distribution and the temporal evaluation are specified by abstract, high-level, functional language concepts, and the pertinent catchword here is **skeletons**. Finally, there is an underlying concept of program deduction that leads from high-level specifications to lower-level implementations. Instead of interactive program synthesis systems, however, we see powerful compilation techniques that can take over at a very early stage of the deduction process.

In contrast to imperative approaches, equational reasoning can be applied in a functional setting (semi-)automatically—a procedure that ensures the correctness of an implementation by the use of correctness-preserving transformations. Moreover, it supports portability across different target architectures by architecture-dependent transformations.

Our approach is based on a number of design decisions. Before dwelling on the technical details, we briefly discuss the considerations that motivate them.

The programmer must be in control of the distribution of the data over the local memories. The predominant parallel architectures on the market nowadays are MIMD machines with distributed memory (such as the CRAY T3E or the IBM SP2). There are strong indications that they will remain the forerunners in the future. In these architectures the distribution of the data over the local memories is the main factor that determines the communication amount. Experience has shown that in most cases the really "good" distributions rely on the properties of the application at hand—which a compiler cannot detect.

The programmer's control over the data distribution must be expressible in abstract mathematical terms, (ideally) without any explicit reference to communication features. As a consequence, we develop a description technique for data structures that amalgamates two viewpoints:

- In one view, data structures are standard mathematical objects that are amenable to standard functional treatment.
- But at the same time, data structures induce another view, which reflects a distribution over processors and the corresponding communication.

This requirement leads us to the introduction of so-called "covers", which are treated in detail in Section 3.2.

The programming model has to reflect the homogeneous nature of the underlying processors. Parallel (MIMD) computers are homogeneous in the sense that they consist of processors of the same kind, which, moreover, are connected in a regular way. Even though the technicalities of the processors and their connections should not concern the programmer, but be left to the compiler, the distinction of a homogeneous environment (as opposed to a network of heterogeneous machines) should be reflected in the programming model. As a consequence, our "covers" have to be described in terms of regular data structures, thus leading to the notion of "data distribution algebras" (discussed in Section 3.3) which are used to define skeletons implementing a SPMD programming model. In particular, this regular design explicitly supports the scalability of algorithms defined on such data structures.

1.1 Notation

Constructive calculi for program development in a functional setting feature quantifier- and index-free notations that rely on suitable concrete representations of data structures. For instance, the geometric representation of matrices used throughout this paper is defined as follows (cf. Pepper & Möller 1991, Bird 1989).

DEFINITION 1
The geometric representation of two-dimensional matrices is defined as

```
STRUCTURE Matrix[α]
  SORT α
  SORT matrix
  FUN □ : matrix                                  -- empty matrix
  FUN ⊡ _ : α → matrix                            -- one-element matrix
  FUN _ ◫ _ : matrix × matrix → matrix            -- horizontal composition
  FUN _ ⊟ _ : matrix × matrix → matrix            -- vertical composition
  FUN _ _ ⊞ _ _ : matrix × matrix × matrix × matrix → matrix
  AXM (a ⊟ b) ◫ (c ⊟ d) = a c ⊞ b d = (a ◫ c) ⊟ (b ◫ d)
    ...
  GENERATE matrix BY □ ⊡ ◫ ⊟
  GENERATE matrix BY □ ⊡ ⊞
```

The notation is based on concepts from the functional language OPAL. It should be essentially self-explanatory.

The above structure introduces a module for matrices over elements of a (parameter) sort α (to argue about the height, width and dimension of matrices, we use the functions `height`, `width` and `dim`, respectively.) Functionalities are introduced by the keyword FUN. Notations such as _ ◫ _ introduce infix operators, where the placeholders '_' indicate the parameter positions. The corresponding function definitions marked by DEF are given in this article using only standard constructs that are customary in functional programming, such as function composition (denoted by the composition operator '∘') and the like. We will also state some derivable equivalences, which are introduced by the keyword THM.(Free) type definitions that are introduced by TYPE are used to define product types and to define constructor und selector functions on the free type.

The structures used in our approach are defined by (parametrized) algebraic specifications because this formalism is best suited for transformations based on equational reasoning. The above definition enables the formulation of laws in a much simpler and comprehensive way than with other formulations:

THM **Monoid[matrix**, □, ◫]

THM $\mathtt{height}(\mathtt{a} \boxminus \mathtt{b}) = \mathtt{height}(\mathtt{a}) + \mathtt{height}(\mathtt{b})$

The specification function `cost` is used to express properties about the communication costs of parallel algorithms.

Our remaining notations are standard: functions are specified using first-order predicate formulae. Function application — curried and/or uncurried — is bracketed, and standard operations on lists and related data types (e.g. vectors) are used: '#' denotes the length of lists or vectors, '::' prepends an element, 'ft' yields the first element, 'rt' the rest, and 'front' all elements except the last one.

2 MODELLING RECEPTIVE FIELD DEVELOPMENT IN THE VISUAL CORTEX

2.1 Visual Cortex

We now introduce a problem that involves solving large systems of differential equations. It is a massively parallel neural network simulation for the development of simple cell receptive fields in the visual cortex. Models and simulations in the neurosciences often involve large networks of neurons that require powerful tools for parallel computation. In the following sections we will demonstrate our approach of data distribution algebras using the mathematical model underlying this example.

In this section we briefly describe (*i*) how the brain can extract simple features from visual images; (*ii*) how the neurons involved in this process may have initally developed proper connections between each other; and (*iii*) how this development process of a large number of neurons may be modelled mathematically. The resulting model (see equation 1) is the basis for our implementations.

Images in the visual field are picked up by photoreceptor cells in the retina. Neurons transmit the image information to the cortex where it is interpreted. In a first step, the cortex extracts simple features from the images such as the position and orientation of edges and the eye from which a stimulus originates. Such feature detector cells can be found in layer IVc of the visual cortex V1. Hubel & Wiesel (1962) characterized them as simple cells and proposed a model for the neuronal connections in the primary visual pathways leading from the eyes via the lateral geniculate nucleus (LGN) to the visual cortex (see figure 1, left). They hypothesize that a cortical neuron becomes **orientation selective** if it receives input from alternating patches of light contrast (ON) and dark contrast (OFF) sensitive cells. In consequence, the neuron would respond maximally to an oriented light bar pattern that matches these receptive field patches (see Figure 1 left). At the same time each cortical neuron receives input signals mostly from only one of the two eyes and thus exhibits **ocular dominance**.

It is remarkable that for light bar stimuli at all angles that are flashed anywhere in one of the two eyes, a neuron can be found in V1 that responds optimally to this stimulus. Furthermore, neighbouring simple cells specialize on similar stimuli and form ordered feature maps (e.g. an ocular dominance map and an orientation selectivity map). Such **cortical maps** are a fundamental principle of cortical organization and can be found in many parts of the brain.

The complex neuronal wiring patterns from figure 1 are not completely genetically determined. Instead, their development depends on cell activity—they are "learned from experience". Based on a simple Hebbian learning mechanism a number of models have been put forward to explain the emergence of orientation selectivity and ocular dominance in simple cells and cortical map formation (Linsker 1986, Miller 1994, Obermayer et al. 1992, Erwin et al. 1995). The Hebbian rule simple states that a synaptic connection between two neurons becomes stronger if they are concurrently active. Here, we introduce a successful **correlation based learning model**.

2.2 Mathematical Model

The model uses simple connectionist-type neuron models: their activity is represented by a value that corresponds to the mean firing rate, i.e. a short time average over its cell membrane potential. This cell output depends on the activity of all the connected input neurons weighted by their connection strength. Neurons connect via synapses, and the connection strength is a value that includes the number of synapses between two neurons as well as their efficiency.

In our model (see figure 1, right), a neuron of LGN population i at location α connects to a cortical neuron at location x with synaptic strength $S^i_{\alpha,x}$. Activity patterns from the retinas (V^i_α) are projected to the LGN (activities P^i_α) and from there on to the cortex (O_x). All locations α, x are measured in retinal coordinates because all the projections are at least roughly topographic.

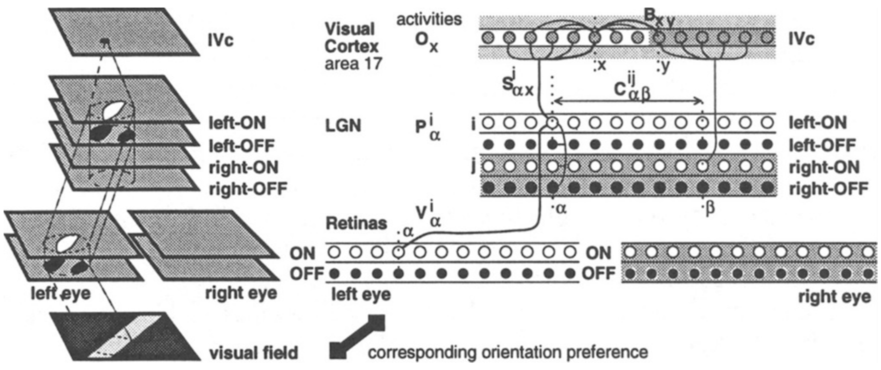

Figure 1 Left: Schematic drawing of the primary visual pathways and the Hubel and Wiesel model for simple cells. The synaptic connections for one cortical cell are shown as black and white patches. The neuron has left-eye ocular dominance and responds best to a light bar stimulus of the shown orientation. **Right:** The correlation-based learning model for simple cells (as on the left, the one-dimensional layers of neurons have to be implemented as two-dimensional sheets).

The correlation-based learning model assumes that orientation selectivity and ocular dominance of cortical simple cells are properties of the LGN-to-cortex connection strengths $S^i_{\alpha,x}$ (Linsker 1986, Miller 1994). Learning starts after the retina, the LGN and the cortex have developed their layered structure and after the formation of topographic but otherwise unspecific connections $S^i_{\alpha,x}$. Then the connection strengths $S^i_{\alpha,x}$ change from the stimuli in the retina by a Hebbian learning rule and orientation selectivity and ocular dominance emerges. The Hebbian principle yields the (linearized) correlation-based learning model (Miller 1994, Piepenbrock et al. 1996, Erwin & Miller 1995).

$$\frac{d}{dt}S^i x_{\alpha,x}(t) = \eta A_{\alpha,x} \sum_{j,\beta,y} I_{x,y} C^{ij}_{\alpha,\beta} S^j_{\beta,y}(t) - \gamma S^i_{\alpha,x}(t), \quad 0 \leq S^i_{\alpha,x}(t) \geq 1 \tag{1}$$

$$\text{where} \quad C^{ij}_{\alpha,\beta} = \langle P^i_\alpha P^j_\beta \rangle \qquad I_{x,y} = \sum_{n=0}^{\infty} (B^n)_{x,y} \tag{2}$$

The parameter η is the **learning rate**. The synaptic arbor function $A_{\alpha,x}$ conserves the **topographic mapping** and represents the maximum number of synapses between an LGN neuron at α and a cortical neuron at x (it is localized and thus zero for large distances $|\alpha - x|$). The intra-cortical interaction function I includes the effects of all connections among the cortex cells and ensures the emergence of the **cortical maps** (the cortical cells excite each other at short and inhibit each other at larger distances $|x - y|$). The two-point correlation functions C incorporates the driving force of the learning process: the **activity patterns** in the LGN that originate in the retinas. The term $-\gamma S^i_{\alpha,x}(t)$ constrains synaptic growth and keeps the $S^i_{\alpha,x}$ non-negative and the total synaptic connection strength for each cortical neuron constant (a nonlinear operation).

To implement this model, we choose constant functions A and I and initialize the synaptic connection strengths $S^i_{\alpha,x}$ unspecifically. (This means random values multiplied with A to establish the initial topographic mapping.) Then we compute the correlation functions C by generating a sequence of sample stimuli V^i_α on the retina and propagating them to the LGN (Piepenbrock et al. 1996). The retinal stimuli V^i_α may consist of spontaneous noise-like activity (Miller 1994) or wave-like patterns (Piepenbrock et al. 1996).

To simulate the development of the simple-cell receptive field, we only have to integrate Equation 1 numerically, in the simplest case using the Euler method. Note, however, that the location indexes α and x each denote two-dimensional position vectors, i.e. $S^i_{\alpha,x}$ is a five-dimensional data object.

The problem of numerically integrating a system of differential equations like Equation 1 is a very common problem in neural network modelling as well as in engineering in general. It is the size of the system that makes the problem difficult in practice. To simulate a large number of cortical neurons (e.g. 128 by 128), we need a parallel computer to make the computations feasible.

Equation 1 has been implemented in C* on a CM-5 parallel computer. In Section 4.4 we discuss this implementation and show that the most difficult part of a traditional imperative solution consists of the management of distributed data and algorithms. These data distribution issues are the motivation for this paper and an efficient solution may be derived using our methodology. This is the main reason for employing the methodology presented in this paper.

2.3 Fast Fourier Transformations and Data Distribution

To simulate the neuronal development, we numerically integrate a difference version of Equation 1 step by step. This can be quite straightforwardly implemented using Fast Fourier transformations because the summation at the core of Equation 1 simply represents a high-dimensional convolution (over the indexes β, y, and j). Here, we use the convolution theorem which states that a convolution of two functions is equal to the inverse Fourier transform of the product of their Fourier transforms. This reduces the number of necessary operations from $O(n^2)$ for a convolution by simple summation to $O(n \log n)$ for an implementation using the Fast Fourier Transform (FFT) (where n is the number of cortical or LGN neurons). Using Fourier transformations (denoted by subscription) and a data-parallel multiplication operation (denoted by '$\cdot$'), this sum can therefore be implemented in a functional setting as a matrix expression

$$\boxed{(I_x \cdot ((C_\alpha \cdot S_\alpha)_{\alpha^{-1}})_x)_{x^{-1}} \qquad (3)}$$

where the subscripts denote the axes along which the annotated matrix expressions are transformed — i.e. C_α ($C_{\alpha^{-1}}$) denotes the (inverse) Fourier transformation of C along (the dimension represented by the) axis α. In the remainder of this paper we consider expressions of this kind as functional programs.

In this paper we are not interested in the functional algorithm for the parallel computation of Fourier transformations. Instead, we assume that a suitable parallel algorithm (called FFT below) for Fourier transformations is part of a library of basic skeletons, which in turn is a part of the so-called skeleton hierarchy (cf. Darlington et al. 1993, Pepper & Südholt 1997). What we are interested in, however, is the interplay between the distribution of the matrix data and the computation of convolutions using Fourier transformations.

Typical implementations of Fast Fourier transformations on MIMD architectures expose several characteristics that interact with data distribution issues. As a paradigmatic example for our methodology, we use the characteristics of the Fast Fourier sub-routines of the library of Thinking Machine's CM-5 (TMC 1993). Some aspects of these characteristics depend on the underlying machine architecture. A skeleton-based approach is quite suitable in this case because it enables the portable implementation of machine-dependent features using architecture-specific transformations.

The efficiency of the computation of convolutions using Fourier transformations on the CM-5 critically depends on two properties that interact with the underlying data distribution (Here we only consider the parameters of the Fourier transformation that are necessary for the derivation of the communication statements. A complete implementation of the FFT library function would obviously necessitate taking into account scaling, multiple instances along different axes, etc.):

1. The matrix elements belonging to an axis along which the matrix is transformed should be local to a processor because the Fourier transformation can then be performed as a purely local operation. The composition of transformations of different matrices and along different axes can therefore be sped up by suitable data redistribution operations that ensure that all transformations can operate locally.
2. The elements of the input and output matrix of a Fourier transformation may be ordered using two different address orderings. A Fourier transformation is executed more efficiently if the address orderings of elements of an axis along which the transformation takes place are different. The address orderings of elements of an axis that is not transformed, however, must be equal.

We account for the first condition by specifying the distribution of matrices using covers and deriving low-level skeletons that implement cover transformations used to "localize" data elements. This issue is the subject of Sections 3.4 and 4.3, respectively.

For referring in implementations to the axes along which Fourier transformations take place, we introduce three auxiliary types:

```
TYPE control   == control(op: operation, inOr: adrOrd, outOr: adrOrd)
TYPE operation == nop forward inverse
TYPE adrOrd    == normal bitreversed
```

The types **control**, **operation** and **adrOrd** represent the parameters that define the transformation along an axis, i.e. which kind of Fourier transformation has to be performed along an axis and the address ordering of the input and output data elements. Control parameters are thus the programming means used to "implement" the axis annotations $\_\alpha, \_x$ in Equation 3.

The second condition can be formalized algebraically using a function `fft` which embeds the library function FFT and conceals the details of its low-level implementation.

DEFINITION 2 (FOURIER TRANSFORMATION)
The Fourier transformation is specified by

```
FUN fft : matrix[α] × seq[control] → matrix[α]                          1
SPC fft(M, c) == M'                        -- FFT (M , c ) == M '        2
    PRE (dim(M) = #(c) ⇒                                                 3
        (fft(M, c') == M' ∧ c ≠ c' ⇒                                     4
         ∀0 ≤ i ≤ #(c) :  (op(c!i) ≠ nop ⇒                               5
                            inOr(c!i) ≠ outOr(c!i) ∨ inOr(c'!i) = outOr(c'!i))   6
                        ∧ (op(c!i) = nop ⇒                               7
                            inOr(c!i) = outOr(c!i) ∨ inOr(c'!i) ≠ outOr(c'!i))   8
        )) ⟹ cost(fft(M, c)) ≪ cost(fft(M, c'))                          9
    POST M' = FFT(M, c)                                                  10
```

The first antecedent of the precondition (Line 3) states that there has to be a control parameter in `c` for each dimension of the matrix `M`. The second antecedent (Lines 4–8) formalizes the restriction imposed above on address orderings: the Fourier transformation `fft(M, c)` is computed more efficiently than a different transformation `fft(M, c')` (computing the same result matrix) if the input and output orders of all axes along which a tranformation takes place are different (Lines 5–6). All elements that belong to dimensions whose axes are not transformed have to use the same input and output ordering (Lines 7–8). Note that the relation between the different costs of the evaluation of `fft(M, c)` and `fft(M, c')` can be quantified — we only use the relation '$\ll$' here for simplicity — based on the implementation of FFT. The postcondition (Line 10), finally, states that `fft(M, c)` actually computes a Fourier transformation using (a specification of) the library function FFT.

3 DATA DISTRIBUTION

As outlined in the introduction, it is mandatory that our programming model takes distribution issues into consideration. This is already the case today to some extent in certain extensions of FORTRAN (such as HPF, HPF Forum

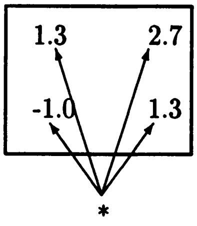

Figure 2 A matrix as a bundle of arrows

1993) and C/C^{++} (such as Split-C, Culler et al. 1993) and C* (Tichy et al. 1992). But we feel that most of these additions are not very systematic and that they are frequently introduced at much too low a level (see C* example in Section 4.4). The goal of our approach is to provide a programming model whereby distribution issues can be described at a high and abstract level geared towards the needs of application domains as opposed to low-level hardware features. In accordance with the prevailing paradigm for the design of parallel software, we restrict ourselves (in this paper at least) to regular communication structures and do not consider arbitrarily communicating processing systems.

3.1 Representing Data Structures

As a first problem, we encounter the necessity for an abstract notion of what we mean by "data structures". Intuitively, we may characterize this idea by pointing to typical examples such as sequences, trees, matrices, vectors and graphs. On the basis of a given programming language, such as Pascal or FORTRAN, we may characterize the available data structures by considering all type constructors of the language. But in generic settings, such as algebraic specifications, it is less clear how data structures like lists can be distinguished from atomic data like floating-point numbers.

In mathematical terms, we sketch our viewpoint as follows:

DEFINITION 3 (DATA STRUCTURE)
*A **data structure** (such as a matrix or a tree) over a set S of elements is a bundle of arrows from a one-element set into S (see Figure 2).*

This slightly clumsy use of arrows (The terminology we employ here is borrowed from category theory. The word "bundle" is used as a synonym for set. At first sight this looks very much like we are talking about pointers, but — as will be seen in a moment — this is only one possible implementation of the abstract concept.) is necessitated by the well-known difficulty of distinguishing multiple occurrences of the same value in a data structure. Using this abstraction, we can extract representation issues. For example, in the matrix in Figure 2 the arrow $* \rightarrow 2.7$ would be represented classically by the index pair (1,2). But in a more geometric setting, it might be represented by a selector north-east, or in a recursive definition (similar to the specification `Matrix` in Section 1.1) by the composition of selectors `top` ∘ `right`.

3.2 Covers

In connection with data distributions, the concept of **substructures** is obviously fundamental. In our setting, its definition is trivial.

Definition 4 (Subobject)
A ***subobject*** *B of a structure A is a subset of the bundle of arrows. We denote the corresponding inclusion morphism by $B \hookrightarrow A$.*

Our goal is, of course, to split a given data structure into subobjects and to distribute these subobjects over the available processors. We therefore have to provide the means for expressing such splittings. Traditionally, one uses partitionings (that is, all subobjects are disjoint) for this purpose. However, we came to the conclusion that this restriction is inadequate. Indeed, allowing the subobjects to overlap is — as will be seen later on — the clue to a much more comprehensible programming style. This motivates the introduction of structures of overlapping substructures, the so-called covers.

Definition 5 (Cover)
Semantically, a ***cover*** *C of a structure A is a set $C = \{B_i \mid i \in I\}$ of subobjects $B_i \hookrightarrow A$ of A such that their union yields A again, that is, $\bigcup C = \bigcup_{i \in I} B_i = A$. (Recall that structures are sets of arrows.) Moreover, the following requirements have to be met:*

- *Every subobject $B_i \in C$ is partitioned into an own part and a foreign part.*
- *The own parts of the B_i form a partitioning of A.*

Forming the union of two subobjects is often called gluing, thus emphasizing the fact that their overlapping parts are identified. Gluing will be denoted here as $A⧉B$.

A partitioning, therefore, is a special case of a cover where the subobjects $B_i \in C$ are pairwise disjoint; in other words, the foreign parts are all empty.

These definitions also entail that the foreign part of each B_i is contained in the own part of some (possibly several) B_j.

Before we proceed with the presentation of our basic concepts, we will briefly review the motivation behind the approach: if we have to compute some function f on a data structure A, that is, $B = f(A)$, then we want to find covers C_A and C_B such that the corresponding subobjects B_i and A_i

can be computed by $B_i = f'(A_i)$ for some suitable function f'. (Actually, the situation is slightly more complex, because there are variations of this simplistic paradigm. But this is exactly what skeletons will be used for later on.)

The purpose of this design is evident: we expect to have B_i and the own part of A_i on the same processor. But for calculating B_i we also need the foreign part of A_i — which determines the communication overhead. So the concept of covers enables an abstract specification of communication requirements.

Note the restriction we apply here: write access can only be applied to the own data of a processor, i.e. remote accesses are only read commands. (If concurrent write accesses were allowed, the consistency of independent computations performed on overlapping covers could be ensured — regardless of any restrictions — by a sheaf-theoretic semantics (Pepper & Südholt 1997).)

But — as mentioned earlier — this restriction is fulfilled in many applications. Moreover, it is weakened to a large extent by the flexibility of the underlying transformational approach to program development: during different stages of the derivation of a parallel program different covers (and domains of possible write access with them) can be considered.

Semantical Framework. Although a complete definition of the formal semantics of data structures, covers and cover operators is beyond the scope of this paper, a short sketch should be helpful for the reader.

The methodology is based on three concepts that are specified algebraically: data structures, covers and skeletons. The algebraic specifications are given a loose semantics.The constituent properties of covers thus cannot be ensured by construction, but are treated as proof obligations. Because loose specifications are often unwieldy, we use the abstract representation introduced in Section 3.1 to distinguish particular models of the specifications. Besides, this model can be naturally used to define other notions that are important in constructive program development, such as the shape and contents of a data structure or the equivalence of different concrete representations of the same data structure (see Pepper & Südholt 1997).

3.3 Cover Operators and Data Distribution Algebras

Covers do not necessarily support homogeneous computations. Arbitrary collections of arbitrarily shaped subobjects may form a cover, but clearly we can only work decently with well-structured covers that exhibit some homogeneity.

Example: A typical situation is illustrated in Figure 3. A matrix is to be distributed over (in this example) $q = 3$ processors. In the light of the properties of the application, we opt for an overlapping row-block cover consisting of $q - 2$ (hence, in our illustration, only 1) inner row blocks plus a top and a bottom borderline block. Each of the inner row blocks consists of

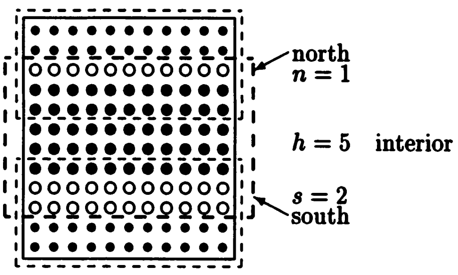

Figure 3 An overlapping row-block cover with 3 blocks

- the own part, that is, of h interior rows, and
- the foreign part, that is, of n "northern" and s "southern" rows. (Such situations, where there are foreign parts both to the north and the south, only work deadlock-free, when certain computational patterns hold for the detailed algorithm.)

For the top and the bottom block, the northern and southern rows, respectively, are missing.

As mentioned earlier, the idea behind this design is that the own part will be assigned to the processor's local memory, whereas the foreign parts provide the compiler with the information necessary to derive the appropriate communication patterns. (In the situation shown in Figure 3, where we assume $n = 1, s = 2$ and $h = 5$, two rows need to be communicated from the bottom processor to the middle processor, and one row from the top processor to the middle processor.) The actual computation of the middle processor is described as if it were working on a matrix with $n + h + s = 8$ rows — without any mention of communication. (End of example)

This example clearly illustrates the basic ingredients needed for defining covers. We are dealing with three kinds of data structure:

- the original structure: `matrix[real]`;
- the structure of the substructures: `block` $\overset{\text{def}}{=}$ `matrix[real]`
- the covering structure: `vector[block]` = `vector[matrix[real]]`

In addition, we have to specify the way in which the original matrix is viewed as a vector of matrices, defining in particular the size of the vector and the overlapping parts. But before we tackle this in detail, we want to settle the question of the overall framework.

Recall that a data structure is a bundle of arrows into some base type; hence, all our data structures are generic which we denote, for example, by `matrix`[α] or `vector`[β], or simply by `matrix[_]` when the name of the parameter sort is not needed. As explained above, the definition of a cover involves three generic structures: the original sort `obj[_]`, the subobject sort `sub[_]` and the cover sort **cover[_]**. (Remember that the different occurrences of the placeholder '_' may refer to different expressions.)

Moreover, we employ a pair of functions split and glue for defining the relationship of the cover to the original object. Semantically speaking,

- split defines how the original bundle of arrows is mapped onto a set of (not necessarily disjoint) bundles which cover the original bundle, and how this set is made into a data structure itself.
- glue defines how the original object can be recovered from its parts.

DEFINITION 6 (COVER SPECIFICATION)
A cover is a refinement of the following specification

```
COVER Cover
  SORT obj[_]                              -- the original object
  SORT sub[_]                              -- the subobjects in the cover
  SORT cover[_]                            -- the structure of the cover
  FUN split : obj[_] → cover[sub[_]]
  FUN glue : cover[sub[_]] → obj[_]
  AXM glue ∘ split = id
```

The row-block cover, for example, can be defined by two functions
FUN **split**: **matrix[α] → vector[matrix[α]]**
FUN **glue**: **vector[matrix[α]] → matrix[α]**

The geometric representation of matrices defined in Section 1.1 enables us to specify distributions in a very concise manner, as illustrated by the following definition (cf. Figure 3):

DEFINITION 7 (ROW-BLOCK COVER)
The (overlapping) row-block cover is defined by

```
COVER RowBlock[q, n, s]
  FUN q, n, s : nat
  REFINES Cover USING obj[_]   == matrix[_]
                      sub[_]   == matrix[_]
                      cover[_] == vector[_]
  AXM split(Mat) = Vec  ⇒  #(Vec) = q
                           ∧ height(Vec.i) = height(Vec.j) ± 1
                           ∧ top(Vec) COVERED  BY  IS[s]
                           ∧ (B ∈ inner(Vec) ⇒  B COVERED BY NIS[n, s])
                           ∧ bottom(Vec) COVERED  BY  NI[n]
  AXM glue(Vec) = ⧉/Vec
  FUN _⧉_ : matrix[α] × matrix[α] → matrix[α]
  AXM height(N) = n ∧ height(S) = s
      ⇒ (A ⊟ N ⊟ S) ⧉ (N ⊟ S ⊟ B) = (A ⊟ N ⊟ S ⊟ B)
```

where '/' denotes the skeleton "reduce" introduced in Section 4.

Because this definition should be mostly self-explanatory, we only mention the following details: the row-block cover covers matrices with vectors of matrices (Lines 3–5) and has as many components as there are processors (Line 6). Consequently, the size of the components is not fixed, but the components are approximately the same size (Line 7). The inner blocks are partitioned as specified by NIS[n, s] (see below).

This specification characterizes all relevant aspects of the concept of row-block covers. Together with the axiom **glue ∘ split = id**, we can follow, for instance, that two adjacent blocks of the cover consist of contiguous parts of the original matrix. But, for the sake of readability, we have refrained from excluding all pathological border cases. Assume, for instance, that the size n of the northern part is larger than the inner size h (see Figure 3). In this case the communication of the elements of a northern foreign part would involve more than the southern neighbouring component.

For the sake of completeness, we also show the cover specification of NIS (the other two are defined analogously).

```
COVER NIS[n, s]
  FUN n, s : nat
  REFINES Cover USING obj[_]    == matrix[_]
                      sub[_]    == matrix[_]
                      cover[_]  == triple[_]
  TYPE triple[_] == (FOREIGN north : matrix[_],
                     OWN     inner : matrix[_],
                     FOREIGN south : matrix[_])
  AXM split(Mat) = triple(N, I, S) ⇒ height(N) = n ∧ height(S) = s
  AXM glue(triple(N, I, S)) = N ⊟ I ⊟ S
```

The covers of a given data structure (such as matrices in this paper) form an algebra, which we call the **data distribution algebra**. Such an algebra introduces further operations on covers, such as "mapping" covers over covers. (This definition only makes sense if the covers, i.e. the splitting and gluing functions, have suitable types. This condition can be met by defining covers using partially defined subobjects. With this technique, all covers of a data type $D[\alpha]$ can be defined to be of type $D[D[\alpha]]$ (cf. Pepper & Südholt 1997).)

Definition 8 (Cover composition)
Let C_1, C_2 be covers, such that $obj_2[\_] = sub_1[\_]$. Then the ***composed cover*** *$C_2 * C_1$ is defined by (where '$*$' below denotes the map skeleton introduced in Section 4.2)*

$\mathtt{split} = (\mathtt{split}_2\ *) \circ \mathtt{split}_1$

$\mathtt{glue} = \mathtt{glue}_2 \circ (\mathtt{glue}_1\ *)$

The foreign parts of the composed cover are determined by

$\mathtt{foreign}(C_2\ *\ C_1) = \mathtt{foreign}(C_1) \cup \mathtt{foreign}(C_2)$

Here, the set of arrows belonging to the foreign part of a cover is defined by $\mathtt{foreign(C)} = \bigcup_{s \in C} \mathtt{foreign(s)}$. The condition on the foreign part implicitly also determines the own part of the composed cover.

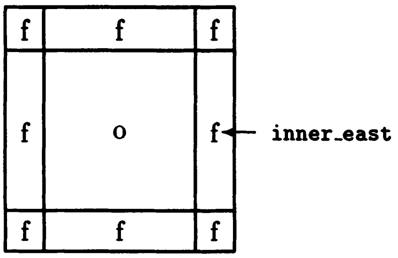

Figure 4 Own and foreign parts of a tile

3.4 Higher-Dimensional Matrix Covers

The cortex equation (Equation 1 on page 6) uses four-dimensional coefficient matrices, because only the position vectors denoted by α and x are relevant here. In Section 2.3 we indicated that we capture some of the characteristics concerning the efficiency of the Fourier transformations using suitable covers, i.e. we examine the following functional algorithm

$$\boxed{dS = \mathsf{norm} * (I_x^{\mathcal{I}} \cdot \; (C_\alpha^{\mathcal{C}} \cdot \; S_\alpha^{\mathcal{S}})_{\alpha^{-1}})_x)_{x^{-1}} \qquad (4)}$$

where the calligraphic letters $\mathcal{I}$, $\mathcal{C}$ and $\mathcal{S}$ denote the covers of the corresponding matrices. norm is the functional implementation corresponding to the term γS in Equation 1 which normalizes the matrix elements as described in Section 2.2.

An important observation, motivated by the deliberations in Section 2.3, is that the Fourier transformations are only performed along two dimensions, instead of along four dimensions. Consequently, we do not define genuine four-dimensional covers, but "lift" suitable two-dimensional covers to four dimensions. This is also the reason why we do not need constructors for higher-dimensional matrices here. All relevant aspects can be specified using the constructors for the two-dimensional case defined in Section 1.1 and the lifting operator defined below.

For simplicity, we choose tiling covers as the basic two-dimensional covers. Instead of defining the tiling cover in an ad hoc manner, we "superimpose" a row-block cover on a column-block cover. The column-block cover is structurally equivalent to a row-block cover and can be defined algebraically using a specification isomorphism:

```
COVER ColBlock[p, w, e]  ==  im(RowBlock[q, n, s])                               1
      WHERE im(p) == q,             im(n) == w,          im(s) == e,              2
            im(height) == width, im(⊟) == ◫,                                      3
            im(north) == west,    im(south) == east                               4
```

In Lines 2–3, the specification isomorphism im maps the parameters and operations of the cover RowBlock onto those of ColBlock (note that the objects, subobjects and cover sort remain the same). In Line 4, the selectors of the local covers are renamed so as to better fit identifiers.

From the axiom (a b ⊞ c d) = (a ◫ b)⊟(c◫d) that holds for the geometric representation of matrices, the theorem

```
THM RowBlock[r,n,s] *∪ ColBlock[c,w,e] = ColBlock[c,w,e] *∪ RowBlock[r,n,s]
```

follows. From the definition of the composition operator $*_\cup$ (Definition 8) we can derive that each subobject here consists of a matrix (its own part) surrounded by matrices forming the foreign part (see Figure 4). We use the previous theorem to define the tiling cover.

DEFINITION 9 (TILING COVER)

```
COVER Tile[r · c,n,s,w,e]  ==  RowBlock[r,n,s] *∪ ColBlock[c,w,e]     1
  FUN inner _ east: matrix[α] → matrix[α]                             2
  DEF inner _ east  ==  inner ∘ east                                  3
      ...                                                             4
```

On Lines 2–3, **inner _ east** defines a selector used to access the tile forming the middle-eastern edge of the foreign part. (The selectors **inner** and **east** are defined as part of the covers **RowBlock** and **ColBlock**, respectively; the analogous definitions for the north-eastern foreign part etc. are missing.) The selectors are used in functions that are defined on the cover, such as the Fourier transformation.

A suitable four-dimensional cover can be defined by lifting the tiling cover to four dimensions using the (product) cover operator ×. This operator can easily be defined algebraically using projections on two dimensions:

Let C_1, C_2 be two-dimensional matrix covers. Then the **product cover** $C_1 \times C_2$ is defined by the requirement
AXM $(C_1 \times C_2)|_{\mathtt{dim}(1,2)} = C_1 \;\wedge\; (C_1 \times C_2)|_{\mathtt{dim}(3,4)} = C_2$

We are now ready to define the covers used in Equation 4. Assuming that the identifier x there refers to the plane determined by the first and second, and α to the third and fourth dimension, the four-dimensional covers are defined as the product,

DEFINITION 10
Let $\mathtt{id}_C$ *be the identity cover defined by* split = glue = id.[1]

```
DEF 𝒳  ==  id × Tile[t₁,n₁,s₁,w₁,e₁]
DEF 𝒴  ==  Tile[t₂,n₂,s₂,w₂,e₂] × id
```

where id denotes the identity cover, $t_1 \cdot t_2$ processors are available and the parameters $n_1, \ldots, e_2$ denote the overlap that depends on the underlying algorithm computing the Fast Fourier transformation. (We come back to this issue in Section 4.3.) The choice of identity cover $\mathtt{id}_C$ is motivated by the observation made in Section 2.3 that (the parallel implementation of the) Fourier transformations are computed most efficiently if the elements along the axes to be transformed are stored locally at a processor. Data distribution (indicated by the tiling covers) should therefore only be specified for the dimensions that are not transformed.

The basic covers used in Equation 4 can then be defined by

DEFINITION 11 (BASIC COVERS)
Let $\mathcal{X}, \mathcal{Y}$ be the covers defined in the last definition.

```
DEF 𝒮 = 𝒞    ==  𝒳
DEF ℐ        ==  𝒴
```

4 SKELETONS

As mentioned earlier, skeletons are certain higher-order functions that are amenable to "good" parallel implementations. Such skeletons exist for all kinds of data structures, but to give a first intuitive flavour we present some of the simplest ones for the case of lists:

- The most basic skeleton is "map", which applies a function to all elements of a list. This function is usually written in infix notation using the symbol '$*$'.

  FUN $* : (\alpha \to \beta) \to \mathbf{seq}[\alpha] \to \mathbf{seq}[\beta]$
  DEF $\mathtt{f} * \langle \mathtt{a}_1, \ldots, \mathtt{a}_n \rangle = \langle \mathtt{f}(\mathtt{a}_1), \ldots, \mathtt{f}(\mathtt{a}_n) \rangle$

  Its parallel implementation is trivial: the sequence elements are distributed across the processing units and the function is applied separately at each processor. Note that the number of processors can be smaller than the number of sequence elements without additional complications. This parallel computation thus induces no communication (except for the initial distribution) at all.
- A close relative of "map" is "zip", which combines two argument lists into a result list by applying an operation to each pair of corresponding elements:

  FUN $\Upsilon : (\alpha \times \beta \to \gamma) \to \mathbf{seq}[\alpha] \times \mathbf{seq}[\beta] \to \mathbf{seq}[\gamma]$
  DEF $\langle \mathtt{a}_1, \ldots \mathtt{a}_n \rangle \Upsilon_{\oplus} \langle \mathtt{b}_1, \ldots \mathtt{b}_n \rangle = \langle \mathtt{a}_1 \oplus \mathtt{b}_1, \ldots, \mathtt{a}_n \oplus \mathtt{b}_n \rangle$

  This skeleton also requires no communication if the elements of the two sequences have been distributed accordingly beforehand.
- Another basic skeleton is the "reduce" skeleton (denoted by the infix symbol '/'), which reduces a non-empty list to a single value:

  FUN $/ : (\alpha \times \alpha \to \alpha) \to \mathbf{seq}[\alpha] \to \alpha$
  DEF $\oplus/\langle \mathtt{a}_1, \ldots, \mathtt{a}_n \rangle = \mathtt{a}_1 \oplus \ldots \oplus \mathtt{a}_n$

  This skeleton can be implemented efficiently (on suitable architectures) in parallel by distributing the data logically in a tree-like fashion and computing the reduction bottom-up from the leaves to the root. Note that $\oplus$ has to be associative.

Besides these general skeletons, more specific ones have been derived from algorithm design tactics (e.g. skeletons for divide-and-conquer or approximation algorithms) or lower-level characteristics such as near-neighbour communication.

Our approach to the transformational development of parallel programs is in line with most other modern skeleton-based approaches in that it should provide a universal set of skeletons that can be used (in principle) to express all other skeletons. This is in contrast to the first approaches to programming by skeletons (Cole 1989), where each skeleton was intended to be some sort of indivisible entity not expressible by a combination of other skeletons — if skeleton composition was not forbidden altogether.

4.1 Implementing the Cortex Equation Functionally

We are now ready to present the functional algorithm that solves the cortex equation (Equation 1). The algorithm shown in Equation 4 introduces appropriate covers for the element matrices. The efficient computation of Fourier transformations, however, necessitates data redistributions as discussed in Section 2.3. We take this into account by inserting cover transformations (denoted by expressions of the form $\mathcal{C}_2 \leftarrow \mathcal{C}_1$) into the final algorithm (remember that $\mathcal{X}$, $\mathcal{Y}$ denote the basic covers developed in Section 3.4).

$$\boxed{dS = \mathsf{norm} * ((\mathcal{X} \leftarrow \mathcal{Y}) \circ (I_x^{\mathcal{Y}} \cdot \ ((\mathcal{Y} \leftarrow \mathcal{X}) \circ (C_\alpha^{\mathcal{X}} \cdot \ S_\alpha^{\mathcal{X}})_{\alpha^{-1}}))_x)_{x^{-1}} \qquad (5)}$$

To complete this functional program, we have three things to do: define the cover-parallel multiplication '·', define the map-skeleton '*' on (overlapping) covers and implement the cover transformations $\mathcal{X} \leftarrow \mathcal{Y}$ and $\mathcal{Y} \leftarrow \mathcal{X}$.

The cover-parallel multiplication can be defined using the zip-skeleton as

```
FUN ·: matrix[real] × matrix[real] → matrix[real]
DEF ·  ==  ⋎(·real)
```

where '$\cdot_{\mathtt{real}}$' denotes the multiplication function on reals. In the next two sections, we address the remaining issues.

4.2 The Generalized Map Skeleton

The algorithm shown as Equation 5 contains functions that are mapped onto covered matrices. The function norm, for instance, is used to normalize all elements of matrices that are covered by a tiling cover. In Section 2.2, it is mentioned that the function norm has to constrain the total synaptic strength of the elements belonging to the dimension x. This operation is local to the dimension α: using the cover transformation $\mathcal{X} \leftarrow \mathcal{Y}$ in Equation 5, this becomes a localized operation.

In addition to constraining the dS, it is also necessary to clip the values of the S at every integration step, i.e. restrict them such that they fit in the interval [0 ... 1]. (This task is denoted by norm′ below.) This is obviously quite a simple case, because this computation can be done using a disjoint cover (although we have not introduced a partitioning in Equation 5 explicitly), given that no communication is necessary.

The definition of the tiling cover as defined in Section 3.4 consists (in our case) of matrices, of matrices of real values. The function norm should obviously normalize the "inner" real values, i.e. we have the following implementation in mind:

```
FUN norm′ : matrix[real] → matrix[real]
DEF norm′(M)  ==  normReal * M
```

where the function **normReal** is applied to the components of the covered matrix, that is, matrices parameterized by basic types and the operator * is the map-skeleton on matrices.

The map-skeleton can be defined as generalizing this pattern:

DEFINITION 12 (MAP SKELETON)
Realizing a function f *based on the* ***map skeleton*** *is defined as:*

```
SKELETON Map[f, g] OVER  Cover
   FUN f :  obj[α] → obj[β]
   FUN g :  sub[α] → sub[β]
   ENRICH Cover BY
   DEF f(A)   ==   glue(g * split(A))
```

As a prerequisite, the cover cover[_] *must possess a map operator '*'.*

Skeletons implicitly specify communication through the access (of g in Definition 12, for instance) to the foreign parts of substructures of the underlying cover. Thus, parameterizing skeletons with covers allows the derivation of cost information based on the cover (and skeleton) definitions (although it does not provide a solution to the monotonicity problem of transformational program derivation (cf. Skillicorn 1993)). This information can be used to caracterize communication free computations as illustrated in the next section, apply replication of foreign parts in order to avoid communication or directing equations such that they become cost reducing transformations (cf. Pepper & Südholt 1997).

The above example norm′ also illustrates a nice property that holds when the overall object and the subobjects have the same type, and when f is already defined using the map operator:

```
THM ∀A :  obj[α] . g * A  =  glue(g * split(A))
```

This theorem states that a function g is mapped over an object A by first splitting the original object, mapping g over the resulting cover and gluing the resulting subobjects together to form the result object.

Coming back to our overall algorithm, the main program in Equation 5 can be augmented by the following instantiation of the map-skeleton, which introduces the function norm′

```
IMPORT Map[norm′, normReal] OVER  Tile[p, 0, 0, 0, 0]
```

The zip-skeleton used in the definition of the function '·' above can easily be defined in terms of a map over pairs of covers.

4.3 Deriving Low-Level Communication Code

The last gap that has to be filled in order to implement Equation 5 is the definition of the cover transformations between the different convolution computations.

(Non-)Local Fast Fourier Transformations. In Section 2.3, an important property of Fast Fourier transformations (on the CM-5) was mentioned: they are computed most efficiently (which means without communication in this case) if the elements belonging to the dimensions that are transformed can be accessed locally. This can be expressed as a property of the underlying data distribution using a transposition function '$\_^{\mathtt{T}}$':

THEOREM 1 (LOCAL FFT)
Let M *be a matrix and* c *a control specification. Then we have*

$$\boxed{\begin{array}{ll}\text{THM} & \mathtt{M}\ \text{COVERED BY}\ \mathtt{C} \ \Rightarrow \\ & \quad (\mathtt{cost}(\mathtt{fft}(\mathtt{M},\mathtt{c})) = 0 \iff \\ & \qquad \forall 0 \le \mathtt{i} \le \#(\mathtt{c}) :\ \mathtt{op}(\mathtt{c}) \ne \mathtt{nop} \ \Rightarrow\ \exists \mathtt{s} \in \mathtt{C} :\ \mathtt{M}|_{\mathtt{dim(i)}} \subseteq \mathtt{own}(\mathtt{s}))\end{array}}$$

where the specification function own *yields the own part of a subobject.*

Performing a Fast Fourier transformation along a non-local axis is equivalent on the CM-5 to transposing the matrix such that the axis is local, computing the transformation and transposing the result. This gives rise to the equivalence

THEOREM 2 (TRANSPOSED FFT)
Let M *be a matrix and* c *a control specification. Then we have*

$$\boxed{\mathtt{fft}(\mathtt{M},\mathtt{c}) \ = \ (\mathtt{fft}(\mathtt{M}^{\mathtt{T}},\mathtt{c}))^{\mathtt{T}}}$$

The communication costs of the transpositions are obviously not negligible. While we illustrate below how algorithms defined as skeletons that operate on covers provide hints how to calculate these costs, let us only assume for the moment that $\mathtt{cost}(\_\hat{}\mathtt{T}) \gg 0$.

Deriving the Cover Transformations. Because convolutions are calculated using Fast Fourier transformations (cf. Equation 5), the previous two theorems can be used to derive properties of the implementation of convolutions. Theorem 2 yields an alternative method for computing convolutions:

THEOREM 3 (TRANSPOSED CONVOLUTION)
Let M, N *be matrices and* c *a control specification. Then we have*

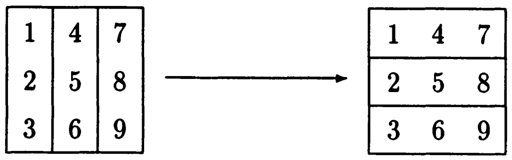

Figure 5 Two-dimensional transposition operation

THM $(M_c\ N_c)_{c^{-1}} = (((M^T)_c\ (N^T)_c)^T)_{c^{-1}}$

Furthermore, Theorem 1 and the fact that transposition involves communication
(`cost(_^T)` $\gg$ `0`) allow us to assign different costs to the lhs and rhs of Theorem 3. This is, however, only possible if the compiler knows how the input matrices are distributed. Since in our framework the covers defined in Definitions 10 and 11 yield exactly the necessary information, the compiler can choose between the two possible implementations. We can conclude that the transformations have to be implemented as

FUN $\mathcal{Y} \leftarrow \mathcal{X}$, $\mathcal{Y} \leftarrow \mathcal{X}$: `matrix` $\rightarrow$ `matrix`
DEF $\mathcal{Y} \leftarrow \mathcal{X}$ == `transpose(`α`)`
DEF $\mathcal{X} \leftarrow \mathcal{Y}$ == `transpose(x)`

where the function `transpose`: `control` $\rightarrow$ `matrix` $\rightarrow$ `matrix` transposes its argument matrix such that the axes that are to be transformed (as given in the control parameter) become local.

The definition of the low-level skeleton **`transpose`** shows how covers can be used to abstractly specify communication. Consider the two-dimensional transposition operation shown in Figure 5. This transposition can be implemented, for instance in a master-slave environment, by first collecting all elements at the master and then redistributing them in a second phase. The collection phase can be implemented by covering the argument matrix by the tiling cover and using an access to the right overlapping part to accumulate the elements that form a row:

```
trans * M WHERE M COVERED BY Tile(t,0,0,0,1)
FUN trans : matrix → matrix
DEF trans(t) == ...inner _ east(t)...
```

The function `trans` is defined on a tiling cover that overlaps one column to the east. It accesses the middle part of the overlap (the element to the right) using a call to the selector function `inner _ east`. If this operation is synchronized such that `trans` is applied to a tile `t` after it has been applied to the left neighbour of `t`, all elements of a row are accumulated on a processor. As shown in Pepper & Südholt (1997) covers allow the concise specification of this and similar classes of synchronized behaviour by recursive equations on covers.

Because the other operations of the implementation shown in Equation 5 (except the cover transformations) are localized operations, we can further derive that the basic tiling covers can be defined as partitionings, i.e. that we can define

```
DEF n1 = s1 = w1 = e1 = n2 = s2 = w2 = e2    ==    0
```

in Definition 10.

4.4 Comparison with C*

We implemented Equation 1 on Thinking Machine's CM-5 and the core code fragments are shown in table 1. The Connection Machine 5 is a massively parallel distributed-memory computer and the data-parallel programming language is C*, a dialect of C. Operations in C* are automatically performed in parallel if they use special parallel data objects—the **shapes**. The number of parallel processors, however, is not known until runtime when the data is distributed according to rules that should minimize the communication overhead.

This implementation is particularly well suited for a comparison with our approach because C* is an imperative programming language designed to provide some of the features that we suggest in our functional programming methodology such as parallel code, automatic data distribution, and implicit data communication between processing nodes.

The program uses four-dimensional shapes (Line 2) (and an array of such shapes for the fifth dimension). The CM Scientific Software Library provides a Fourier transformation that can be used to transform high dimensional data. We define controls (Lines 6–8) for the FFT for forward and backward transformation (*i*) along the two LGN dimensions and (*ii*) along the cortical dimensions.

First, we allocate the shape (Line 10–13), i.e. we tell the system how much data the parallel variables will hold ($32 \times 32 \times 32 \times 32$ elements) and how to distribute the data across the system. The default rule for data distribution assumes an equal amount of communication along each of the shape dimensions. This default distribution would be optimal for an FFT along all four dimensions. For our particular problem, however, it proved to be more efficient to keep the two data axes local to a processor along which most operations take place, and then transpose the data when necessary (just as we discussed in the previous sections). Thus, to increase the performance we request a data distribution where the the first two dimensions are evenly distributed according to the default rule and the third and fourth dimension are arranged in serial order (i.e. local to a processor). Once a shape is declared it can be used to generate data objects (Lines 3, 4, 14).

The main loop of the program (Lines 18–49) then consists of Fourier transforming the data along the α axes (Lines 20-24), then multiplying it with the FFT of C (Lines 27-34) and transforming it back (Line 35). Then we redistribute the data to make the x axes local (Line 38), convolve with I (Lines 39-42) and transpose back. Finally, the constraint normalizes the synaptic

```
/* declare parallel variable shape and pointers to parallel variables */
shape [][][][]ModelMap;
double:ModelMap *S[4], *dS[4], *C_fft[4], *I_fft, *A;
CMSSL_double_complex_t:ModelMap *S_fft[4], SUM_fft;

CMSSL_fft_control_t
  forward_lgn_ctrl[4] = {no_fft, no_fft, fwd_fft, fwd_fft},
  inverse_lgn_ctrl[4] = {no_fft, no_fft, inv_fft, inv_fft};

/* allocate parallel variable shape with serial LGN axes */
ModelMap = allocate_detailed_shape (&ModelMap, 4, {32, 32, 32, 32}, NULL,
  {CMC_news_order, CMC_news_order, CMC_serial_order, CMC_serial_order},
  NULL, NULL, NULL);
/* allocate all parallel variables; setup FFT */
/* compute A; randomly initialize S; compute I and C and FFT them */
... code missing ...

for (iter = 0; iter < no_iterations; iter++) {
  /* FFT S along LGN dimensions: S --FFT--> S_fft */
  for (i = 0; i < 4; i++) {
    re (*S_fft[i]) = *S[i];
    im (*S_fft[i]) = 0.0;
    CMSSL_fft_detailed (S_fft[i],CMSSL_cmpx_to_cmpx,forward_lgn_ctrl,id)
  }

  for (i = 0; i < 4; i++) {
    re (*SUM_fft) = 0.0;
    im (*SUM_fft) = 0.0;
    for (j = 0; j < 4; j++) {
      /* convolve S with C means multiply S_fft with C_fft */
      cind = ((i/2==j/2) ? 0 : 2) + (i+j)%2; /* choose 1 of the 4 C's */
      re (*SUM_fft) += re (*S_fft[j]) * C_fft[cind];
      im (*SUM_fft) += im (*S_fft[j]) * C_fft[cind];
    }
    CMSSL_fft_detailed (SUM_fft,CMSSL_cmpx_to_cmpx,inverse_lgn_ctrl,id);

    /* transpose data; FFT; multiply w. I_fft; FFT back; transp. back */
    [pcoord(2)][pcoord(3)][pcoord(0)][pcoord(1)]re(SUM_fft)= re(SUM_fft);
    CMSSL_fft_detailed (SUM_fft,CMSSL_cmpx_to_cmpx,forward_lgn_ctrl,id);
    re (*SUM_fft) *= I_fft;
    im (*SUM_fft) *= I_fft;
    CMSSL_fft_detailed (SUM_fft,CMSSL_cmpx_to_cmpx,inverse_lgn_ctrl,id);
    [pcoord(2)][pcoord(3)][pcoord(0)][pcoord(1)]*d_S[i] = re(*SUM_fft);
    *d_S[i] *= learn_step_size * *A;
  }

  /* compute constraint term (gamma) and update the weights */
  ... code missing ...
}
```

Table 1 C* program for the computation of Equation 1

weights for each cortical cell (Line 47/48; code not shown) which is local to the α axes.

C* tries to conceal as much of the parallel architecture from the programmer as possible. Data objects can be declared and used without knowing about the hardware which makes C* a portable language. However, the default shapes are designed for "average" communication needs, no mechanism for (semi-automatic) data rearrangement depending on the communication costs assessment is provided. Shapes have to be "hand made" (like in our example, Line 11) when the default data distribution is not optimal for the parallel problem at hand. Shape declaration, and by that means, data distribution is an a-priori decision that is independent of the rest of the C* program. Therefore it is not elegantly possible to redistribute data when appropriate (see e.g. the shape transposition in lines 38 and 43 that is easily possible only because all four shape dimensions have the same size. Alternatively, we can redistribute a data object in memory by declaring a second shape of the same size but with a different layout in memory and then casting the data.)

From this discussion it should be clear that the data-parallel constructs of C* support only a very restricted programming model in comparison to the methodology presented in this paper. Shapes, for instance, are a very restricted subset of partitioned covers. Hence, the communication and synchronization requirements have to be implemented explicitly and cannot be derived as with our methodology. Furthermore, C* does not provide any means to specify the constraints on address orderings as is done in Section 2.3 (These requirements are (rather informally) described in the library description (TMC 1993)).

5 CONCLUSION

In this paper we have presented an extension of the skeleton methodology by covers. This abstraction enables us to declaratively treat two important aspects of parallel programming that are commonly dealt with in an ad hoc manner. First, data distribution algebras enable a high-level and precise specification method for data distribution issues. They have been shown to be useful for providing the information necessary to derive communication statements for data redistribution. This feature has been shown to largely simplify the formulation of parallel algorithms in comparison to data-parallel imperative methodologies.

Second, the methodology enables the concise definition of architecture-specific features. This information can then be used for deriving the low-level communication code.

5.1 Related Work

The derivation of sequential programs by program transformation has long been a topic of research (Feather 1987, Bauer et al. 1985). More recently, this research has focused on the application of constructive calculi for the derivation of parallel programs. While Cole (1989) introduced **skeletons** in an imperative framework, most work — starting with the work of Darlington et al. (1993) — has concentrated on applicative calculi (Pelagatti 1993, Geerling 1996, Bratvold 1993), because the integration of skeletons into applicative programming languages is very smooth and leads to descriptions that are quite close to mathematical specifications. Considerable attention in the field of the transformational derivation of parallel programs has also been given to the so-called **Bird-Meertens Formalism** (Meertens 1986, Bird 1989). Skillicorn (1992) works on categorial data types that provide a set of polymorphic types together with a restricted class of functions, called catamorphisms, which are relatively easy to parallelize. Skeletons can, however, also be integrated into other calculi besides the Bird-Meertens Formalism. Geerling (1996), for instance, uses skeletons in an assertion-based calculus formulated in a Dijkstra-style logic framework. Most of this work — with the exception (to a certain degree) of Bratvold (1993) and Skillicorn (1993) — does not explain how to break down transformations for automatic compilation guided by appropriately chosen cost functions. However, the research did yield a proliferation of transformational derivations useful for practical problems (Pepper 1993, Pepper et al. 1993).

There are some approaches to the transformational development of parallel programs that rely on the concept of **shapes** (Jay & Cockett 1994). Shapes represent containers for data elements and can be manipulated using their polymorphic properties. While this approach requires analysis methods that are quite similar to ours, all the other approaches put forward up to now only deal with disjoint shapes. Shapes have also been applied to non-transformational imperative frameworks for parallel programming (see, for example, work concerning the object-oriented language Sather (Schmidt 1992)). Similar data distribution techniques are also diffusing (to a small extent) into conventional imperative programming languages. One such example are the distribution directives of High Performance Fortran (HPF Forum 1993).

REFERENCES

Bauer, F. et al. (1985), *The Munich Project CIP. Vol. I: The Wide Spectrum Language CIP-L*, number 183 *in* 'LNCS', Springer Verlag, Berlin.

Bird, R. S. (1989), Lectures on constructive functional programming, *in* M. Broy, ed., 'Constructive Methods in Computing Science', Vol. 55 of *NATO ASI, F*, Springer, pp. 151–216.

Bratvold, T. A. (1993), A skeleton-based parallelising compiler for ML, *in* R. Plasmeijer & M. van Eekelen, eds, 'Implementation of Functional Languages', U. Nijmegen. TR 93-21.

Cole, M. (1989), *Algorithmic Skeletons: Structured Management of Parallel Computation*, MIT Press.

Culler, D. E. et al. (1993), Parallel programming in Split-C, *in* 'Supercomputing'.

Darlington, J. et al. (1993), Parallel programming using skeleton functions, *in* A. Bode, M. Reeve & G. Wolf, eds, 'PARLE '93', pp. 146–160.

Erwin, E. & Miller, K. (1995), 'Modeling joint development of ocular dominance and orientation in primary visual cortex', *Proc. of the CNS.*

Erwin, E., Obermayer, K. & Schulten, K. (1995), 'Models of orientation and ocular dominance columns in the visual cortex: A critical comparison', *Neur. Comp.* **7**, 425–468.

Feather, M. S. (1987), A survey and classification of some program transformation approaches and techniques, *in* L. Meertens, ed., 'Program Specification & Transformation', North-Holland.

Geerling, M. (1996), Transformational Development of Data-Parallel Algorithms, PhD thesis, Katholieke Universiteit Nijmegen.

HPF Forum (1993), 'High Performance Fortran', *Scientific Programming.*

Hubel, D. H. & Wiesel, T. N. (1962), 'Receptive fields, binocular interaction and functional architecture in the cat's visual cortex', *Journal of Physiology London* **160**, 106–154.

Jay, C. & Cockett, J. (1994), Shapely types and shape polymorphism, *in* D. Sannella, ed., 'Programming Languages and Systems - ESOP '94', LNCS, Springer Verlag, pp. 302–316.

Linsker, R. (1986), 'From basic network principles to neural architecture: Emergence of orientation selective cells', *Proceedings of the National Academy of Science USA* **83**, 8390–8394.

Meertens, L. (1986), Algorithmics — towards programming as a mathematical activity, *in* 'CWI Symposium on Mathematics and Computer Science', North-Holland, pp. 289–334.

Miller, K. (1994), 'A model for the development of simple cell receptive fields and the ordered arrangements of orientation columns through activity-dependent competition between ON- and OFF-center inputs', *Journal of Neuroscience* **14**, 409–441.

Obermayer, K., Blasdel, G. G. & Schulten, K. (1992), 'A statistical mechanical analysis of self-organization and patte rn formation during the development of visual maps', *Phys. Rev. A* **45**, 7568–7589.

Pelagatti, S. (1993), A Methodology for the Development and the Support of Massively Parallel Programs, PhD thesis, Universita di Pisa-Genova-Udine.

Pepper, P. (1993), Deductive derivation of parallel programs, *in* R. Paige, J. Reif & R. Wachter, eds, 'Parallel Algorithm Derivation and Program Transformation', Kluwer Academic, chapter 1, pp. 1–53. Also as TR 92-23, TU Berlin.

Pepper, P. & Möller, B. (1991), Programming with (finite) mappings, *in* M. Broy, ed., 'Informatik und Mathematik', Springer Verlag, pp. 381–405.

Pepper, P. & Südholt, M. (1997), Deriving parallel numerical algorithms using data distribution algebras: Wang's algorithm, *in* 'Proc. of the 30rd Hawaii International Conference on System Sciences'.

Pepper, P., Exner, J. & Südholt, M. (1993), Functional development of massively parallel programs, *in* D. Bjorner et al., eds, 'Formal Methods in Programming and Their Applications. LNCS 735', Springer Verlag, pp. 217–238.

Piepenbrock, C., Ritter, H. & Obermayer, K. (1996), 'Linear correlation-based learning models require a two-stage process for the development of orientation and ocular dominance', *Neural Processing Letters* **3**, 31–37.

Schmidt, H. W. (1992), Data-parallel object-oriented programming, *in* 'Fifth Australian Supercomputer Conference, Melbourne', 263–272.

Skillicorn, D. B. (1992), The Bird-Meertens Formalism as a parallel model, *in* 'NATO ARW "Software for Parallel Computation"'.

Skillicorn, D. B. (1993), 'Deriving parallel programs from specifications using cost information', *Science of Computer Programming.*

Tichy, W. F., Philippsen, M. & Hatcher, P. (1992), 'A critique of the programming language C*', *Communications of the ACM* **35**(6), 21–24.

TMC (1993), *CMSSL for C**, Thinking Machines Corp.

6 BIOGRAPHY

Mario Südholt is a member of the declarative language group at the research institute in computer science Irisa/Inria-Rennes, France. He received his Masters degree in computer science from the University of Coblence in 1992 and will complete his PhD at the Technical University of Berlin, Germany, in June 1997. His research interests cover the formal derivation of sequential and parallel algorithms, (declarative) programming languages and formal models for the description of software architectures.

Christian Piepenbrock is a member of the Neural Information Processing Group at the Department of Computer Science at the Technical University of Berlin. He received his Masters degree in the interdisciplinary program in Computer Science and Biology from the University of Bielefeld in 1995. His research interests are models for the structure and development of neuronal networks in the visual system and the simulation of such parallel networks.

Dr. Klaus Obermayer is Professor for Neural Information Processing at the Department of Computer Science at the Technical University of Berlin. He received his Masters degree in physics in 1987 from the University of Stuttgart and his PhD in 1992 from the Technical University in Munich, Germany. His research interests cover the areas "computational neuroscience", statistical physics of neural networks and the application of artificial neural networks in signal processing and data analysis.

Peter Pepper received a Masters Degree in Mathematics and a PhD in Computer Science from the Technical University Munich. Since 1985 he has been Professor for Computer Science at the Technical University of Berlin, where he holds a chair in Compiler Construction and Programming Languages. He is a member of GI, ACM, EATCS, IFIP working groups WG 2.1 and WG 14.3, and an affiliate of IEEE. His research interests cover functional programming, algebraic specification, transformational program development, formal derivation of parallel algorithms and software engineering for safety-critical systems.

Discussion Session
Monday afternoon

Jim Boyle: I would like to ask the last speaker, do you think that, assuming you had the manpower to complete the derivation to C^* code, would the result be more or less efficient than the hand-written program that you showed?

Mario Südholt: The efficiency of the final algorithm depends on the level at which you let the user interact with the system. Another question is what can we do with the transformation that accompanies the basic covers and the skeletons relating them. We have several case studies where we get almost optimal algorithms, but this depends on the specific communication behaviour assigned to skeletons, which is different than that used in our paper. The communication says, when access to a foreign part is required, does the owner have to perform the computations before or after the access? With explicit control over synchronization, you get optimal algorithms in many cases.

Peter Pepper: Parallel algorithms in the literature are usually presented by some high-level informal description, followed by a program in which each variable had probably two or three indices, followed by an efficiency analysis. It is clear to me, since many of these indices contain mistakes, that in most cases the efficiency analysis was based on the informal description, not on the algorithm as presented. Numerical analysts think in terms of matrices, and there is a huge gap between this level and the C^* or HP Fortran code. Our work is concerned to lift the level of the programming. At least in that area, I don't think we have new algorithms, just an easier way to describe good algorithms. At the moment I would be happy if we can come up with the best algorithms that peoples have invented already, but with a shorter description, and one that enables variations of the algorithms to be programmed more easily.

Jim Boyle: I want to direct the same question to Perrin, do you have any indication of how the derived implementation in PEI would compare with a hand-crafted one?

Guy-René Perrin: I would give the same answer as Peter Pepper. We are only able to abstract what practitioners do, and to understand how they do what they do, and how we can find variations on algorithms and data distributions.

Wolf Zimmermann: This is to both speakers on data parallelism. You all start from an initial data distribution. Why don't you assume initially that you have to share the memory, then you don't need to worry about data distribution? So, for example, people who consider implementation of parallel algorithms as a scheduling problem, first more or less schedule the computations and then derive from that the appropriate data distributions.

Mario Südholt: One answer to this question is that there are models for shared memory programming that can be embedded into distributed architectures so that the algorithms are equally efficient. However, there are algorithms that are not of this form. At present I have not seen a programming methodology restricted to these pro-shared memory architectures that can be applied in a generality that is sufficient for practical programming.

Guy-René Perrin: When you program a shared-memory machine, practitioners say that what you measure is the efficiency of the machine, not the algorithm. I think it is not useful to draw a strict difference between these two points because if you have a shared-memory machine, you have a real problem of locality if it is a non-uniform memory access machine. So its better to say that the real work is about the locality of data in order to achieve either a shared-memory machine with cache memory, or distributed-memory machines.

Wolf Zimmermann: I agree with your answers, but somehow you misunderstood my question. I asked about the starting point, and I would say that starting with a shared-memory program would be cheaper. What people in scheduling theory do is to look at the operations being performed and model them as a directed acyclic graph, and then schedule the operations so that the execution time is minimized. In several cost models, they can even give performance guarantees.

Peter Pepper: The initial specification shouldn't talk about machines at all. When you are talking about parallelism, it is already a first step toward implementation. Then you can make your mind up whether you first model it as a shared or distributed-memory problem. My experience is that once you start going towards the shared-memory model, its very hard to find the way back to the distributed-memory model later on. The point is that data locality is the decisive issue in distributed-memory machines. These are the predominant machines in the market and I am pretty sure that they will continue to be predominant in the future. I believe that one day we will just see networks of workstations linked together. So, knowledge about locality is application dependent. If you pretend that there is shared memory, the compiler or runtime system is unable in most cases to figure out where the data should be kept. My conclusion is that we should allow the user to express his knowledge about locality as soon as possible and on as high a level as possible. The trick, of course, is not to let this complicate the algorithmic thinking, but to keep the solutions simple enough while still being able to express locality issues.

Helmut Partsch: To change the topic, in previous working conferences we had the benefit of having people outside the environment of WG2.1 because they sometimes raised provocative questions. One of the referee's reports on a (non-accepted) paper for this conference was that it was a very nice calculation, but so what? If I interpret this right, there was a general kind of doubt about the things we are doing here. Any comments on that?

Richard Bird: It seems desperately important to me that we do not rehearse exactly the same worries and anxieties that we had 11 years ago at the Bad Tölz meeting. What we should be doing is looking at progress. Have we made any progress in 11 years? I think we have; there is a maturity about the presentations today that just wasn't there 11 years ago. New young people are coming into the field because the subject of program calculation is one that excites and interests them. As long as the topic remains stimulating and not stagnating for you, I don't think you should worry about relevance to industry or technology transfer. We shouldn't ignore it, though. If there is a knock on the door, then fling it open quickly. Interaction is extremely fruitful, but soul-searching is not.

Peter Pepper: I would like to agree. There has been progress, and there is clearer understanding of the things that remain to be done. It takes a while for this progress to get into the industrial community. We have to wait 10 or 15 years before industry knocks at our doors.

Doug Smith: Industry is not going to knock on our doors in 10 or 15 years, since most funding is for doing applied research now. No one is going to make the stuff work for us, we have to bite the bullet and do it ourselves. It is worthwhile conducting more elaborate experiments to provide evidence that there is going to be some payoff. I am not saying that everyone should do that, but we need more groups who can invest the effort in implementation. The downside is that it is very expensive from the research point of view to get immersed is a domain deeply enough to do something significant.

Tom Maibaum: I don't know how many of you remember Tapsoft '85 when David Parnas stood up and said that you theoreticians have not told us which languages and which methods to use to improve our software engineering. Sometime later in the conference, Maurice Nivat stood up and said, of course not, that's not what scientists do. There is a lot of evidence that over a suitable time scale theoretical ideas do become everyday technology used in industry. For example, consider relational database theory in the late 60s and early 70s; nowadays nobody needs to know about the mathematics, they just use it in management information systems. The same story is true of parsing. When I was an undergraduate all the theory courses were about parsing and automata theory. Nobody studies it any more because they can use the technology without having to know the mathematics behind it. One day others will use systems such as Kids not because they know the mathematics but because enough knowledge will be encapsulated in the system. This is where I think the algebraic specification people are going wrong. They won't affect industry now because industry is not going to invest an enormous amount of money in time and education with ignorant people to use these kinds of languages.

It is only slowly over time that people will have the tools, both conceptual – because of their education – and technical, to make it work.

Alberto Pettorossi: One reason we have made progress is that we have concentrated on the simple but significant idea of equational reasoning. What is lacking is the meta-level reasoning to drive such calculations. A challenge for the future is to expose useful kinds of meta-level reasoning.

Peter Pepper: Just to add one more example to what Tom Maibaum has said about ideas moving into areas where people just use the technology. Just recently, Bruce Shriver during the HICSS conference addressed key technologies for the future. I was surprised to see that number 2 on his list was optimizing compilers. His argument was that many speed-ups were due simply to better compiler technology. If I look at compilers I see many techniques developed by this community being moved into compilers. This is an example where people do not sit down and use our methods to calculate programs, but just write higher-level programs than before and use the compiler to do the work for them.