

Parallaxis-III

Architecture-Independent Data Parallel Processing

Thomas Bräunl
The University of Western Australia
Electrical and Electronic Engineering
Nedlands, Perth WA6907

www.ee.uwa.edu.au/~braunl/parallaxis

Abstract

Parallaxis-III is an architecture-independent data parallel programming language based on Modula-2. It has been designed for teaching data parallel concepts and is in use at a large number of institutions. Compilers exist for data parallel systems as well as for a sequential simulation system. A data parallel graphics debugger allows efficient source level analysis for parallel programs.

Keywords

data parallel, virtual architecture, virtual processors, parallel debugger, data visualizer, traffic simulation.

1. Introduction

Parallaxis [Bräunl 89, 91, 93] is based on Modula-2 [Wirth 83], extended by data parallel concepts. The language is fully machine-independent across data parallel architectures; therefore programs written in Parallaxis run on different parallel computer systems. For a large number of (single-processor) workstations and personal computers there is a Parallaxis simulation system with source level debugging and tools for visualization and timing. Parallel programs with small data sets can be developed, tested and debugged with this simulation system. Then, Parallaxis compilers can be used to generate parallel code for data parallel systems. The simulation environment allows both the study of data parallel fundamentals on simple computer systems and the development of parallel programs, which can later be executed on expensive parallel computer systems. The programming environment for Parallaxis is available as public domain software [Bräunl 96].

Our major goals in developing this new parallel programming language was to simplify data parallel programming and to build on an existing *good* programming language. We chose Modula-2 over C, since it has a much clearer structure and exhibits numerous advantages when used for teaching programming concepts. We made Parallaxis a superset of Modula-2, so sequential algorithms can easily be expressed as a standard Module-2 program. Since some concepts are easier to formulate in a parallel fashion than in an iterative way, some algorithms actually become simpler in the parallel version. Since we are interested in data parallel processing in general, we avoided any machine-specific references in the language. Parallaxis is therefore a completely machine independent data parallel programming language.

The central point of Parallaxis is programming on a level of abstraction with virtual PEs and virtual connections. In addition to the algorithmic description, every program includes a con-

nection declaration in functional form. This means that the desired connection topology is specified in advance for each program (or for each procedure) and can be addressed in the algorithmic section with symbolic names instead of complicated arithmetic index or pointer expressions. However, full-dynamic data exchange operations are possible as well.

2. Data Parallel Language Constructs

The hardware structure of data parallel systems has to be mapped into programming language constructs. As a result, the parallel features of all data parallel programming languages turn out to be quite similar and share the same groups of language constructs listed below:

- a. Declaration of virtual processors and connections
- b. Declaration of vector data
- c. Parallel execution
- d. Parallel data exchange
- e. Vector reduction
- f. Vector - scalar exchange

For discussing the individual groups of language constructs, we would like to point out that data parallel entities are rather similar to elements in a data array, which are being worked on in parallel. This turns out to be a good analogy when discussing language features and in fact is the model chosen for some data parallel languages such as Fortran-90 [Metcalf, Reid 90].

In the following, the term "**vector**" will be used to describe a data parallel data structure, while the term "**scalar**" describes data structures used within the sequential sequencer part of a data parallel system. E.g. a loop counter is usually a scalar variable in a data parallel program, while a whole image could be a single vector, distributed over all available processing elements (PEs). On the hardware side of a data parallel system, scalar and vector data usually reside in separate memory units and are accessed by different processors.

(a) Declaration of virtual processors and connections

Before any data parallel calculation can start, the number of data parallel entities, or the "vector length" of the parallel data structures involved has to be specified.

[In our analogy this corresponds to the array length in an array declaration.]

Several languages like C* [Rose, Steele 87] and Parallaxis support this language construct, e.g.:

```
CONFIGURATION p[0..99]; (* declare PE structure "p" with 100 PEs *)
```

Others, however, like MPL [MasPar 91], do not provide language constructs to address this issue. This results in a machine dependence, since the same program will behave differently (it will in fact produce different results or not run at all) on a machine with less PEs than intended. As a consequence, the application programmer has to deal with the machine size in all user programs. On the other hand, freely specifying the "vector data length" (or number of PEs in a system) creates an additional level of abstraction, often called "virtual processors". This enables machine-independent (or more precisely machine-size-independent) programming with only minimal run-time overhead.

While a number of languages support the specification of virtual processors, only a few allow them to be structured in dimensions (like two- or more-dimensional data arrays), and

only Parallaxis implements the specification of virtual connections. Keeping the array analogy, most languages allow data exchange using indices (see point d, below). However, we are going one step further by defining these index mappings up front, so symbolic names can be used instead of indices in the algorithm part. E.g.:

```
CONNECTION
  list: p[k] -> p[k+1];          (* connect PEs as list *)
  ring: p[k] -> p[(k+1) MOD 100]; (* connect PEs as ring *)
```

(b) Declaration of vector data

The same way that scalar data must be declared before it can be used, vector data has to be declared appropriately and has to be distinguished from scalar data. The simplest form for doing this can be by using different keywords, e.g. (from Parallaxis-2):

```
SCALAR i: INTEGER; (* this variable exists only once *)
VECTOR a: INTEGER; (* each PE defined previously has its local a *)
```

This form of declaration, however, would allow only one vector type in a program's block structure (i.e. a procedure), since there is no language construct to associate a configuration with a variable. Parallaxis-III treats the "vector-ness" and vector structure as part of the data type. This allows declaration of variables with different vector structures and different base types next to each other. E.g. (in Parallaxis-III):

```
VAR i: INTEGER;          (* sequential as in standard Modula-2 *)
    a: p of INTEGER;    (* vector of configuration "p" as def. above *)
```

(c) Parallel execution

Some languages provide special language constructs to indicate data parallel execution, e.g. in some task-level parallel languages:

```
"FORALL i in [1..100] DO ... (* in parallel *) END"
```

However, this is not necessary for languages like Parallaxis, which have proper vector data declaration. The execution style (data parallel or sequential) is implicitly defined, by assuming that all operations on vector data is to be performed data parallel, while all operations on scalar data is to be performed sequentially. E.g. using the previous example declarations:

```
FOR i := 1 TO 100 DO (* sequential loop iter., since i is scalar *)
  a := a + 1;        (* parallel operation, since a is vector *)
END;
```

(d) Parallel data exchange

This operation is required to do data shuffling inside a vector, e.g. a permutation of vector elements in case of a 1:1 exchange structure. A data parallel data exchange operation is in fact the most interesting, but often also the most confusing operation. Unlike communication between independent parallel tasks, data parallelism requires synchronous execution between all active PEs. This means that *all* PEs send/receive data using the *same* exchange pattern *concurrently*.

While some languages merely allow the specification of an index expression, Parallaxis encourages the use of previously declared directions. E.g.

```
a := MOVE.ring(a);
```

This will rotate all elements of vector a . Things get more complex, if the data exchange structure does not follow a 1:1 mapping. This means the language has to define what happens when source or destination indices do not exist. The former could lead to undefined data, while the latter could result in lost data. Both problem domains are demonstrated in the following data exchange example. They will be discussed in more detail later on:

```
a := MOVE.list(a);
```

(e) Vector reduction

All data parallel languages provide either special language constructs or special low-level system routines to deal with vector reduction. Assume we are interested in the sum of all components of a vector. Then - if syntax were to allow it - we could find out by sequentially adding all components. However, we would have difficulties describing the same process in parallel, because this depends on the actual parallel hardware structure (grid, tree, etc.) and its relative communication timings. Therefore, special routines or constructs are required in one way or another.

(f) Vector - scalar exchange

Finally, each data parallel language must provide a way to exchange values between vector and scalar variables. Since I/O is usually sequential it would otherwise be impossible to initialize a vector with stored data or to print the result of a vector operation. The exchange can be between vector components and simple scalar variables or between a whole vector and a scalar data array.

3. Parallaxis

We will now take a closer look at the data parallel language constructs in Parallaxis-III by following the outline of language constructs from the previous section.

3.1 Virtual Processors and Connections

One or several ‘virtual machines’ consisting of processors and a connection network may be defined for every Parallaxis program. This is done in two simple steps. First, the keyword `CONFIGURATION` is used to specify the number of PEs and their arrangement in analogy to an array declaration. However, at this point, no specification has been made as to the connection structure between the PEs. This follows by specifying mapping relations, introduced by the keyword `CONNECTION` (this second step may be omitted if no connections are required). Every connection has a symbolic name and defines a mapping from a PE (*any* PE) to the corresponding neighbor PE. The specification of this *relative* neighbor is accomplished by providing an arithmetic expression for the index of the destination PE. Data exchanges can now be carried out using these symbolic connection names in the parallel program.

In our array analogy this two step declaration process looks like:

- a. **Configuration:** defines number of PEs required and their n-dimensional arrangement
[resembling an array type declaration]
- b. **Connection:** defines links between PEs based on their position in a configuration
[resembling the definition of a set of array indices]

Configurations are data types - they only come to life when used in the declaration of a vector variable. Connections are attributes - they can only be used in data exchange for a vector.

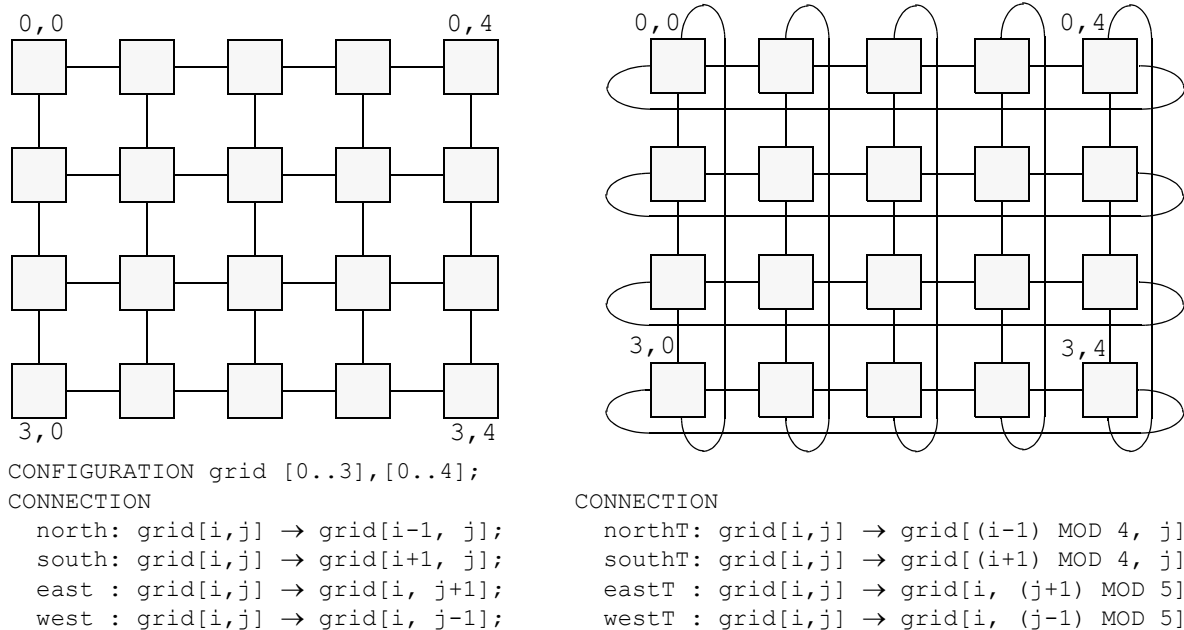


Figure 1: Two-dimensional grid and torus topology

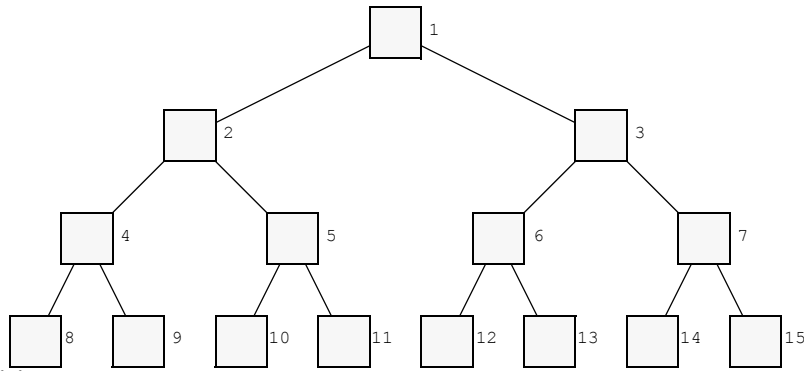
Figure 1 shows a PE arrangement as a two-dimensional grid structure in a simple Parallaxis example. The `CONFIGURATION` declaration provides 4×5 virtual processors, which are virtually connected to one another in the following `CONNECTION` declaration. The syntax for configurations follows roughly the (multi-dimensional) array declaration in Modula-2. Since homogeneous connection structures or topologies are easy to declare, four connection declarations are sufficient to construct a mesh of any size. One connection is defined for each cardinal direction. The connection to the north, for example, decrements the first index. Note that connections for some PEs result in non-existing PE positions. This is not an error - connections at border PEs are allowed to lead to ‘nowhere’, which means that these connections do not exist, and will not participate in any data exchange operation.

In case an application requires a torus instead of an open mesh, this can be easily accomplished by using the modulo-operator (see right-hand side of Figure 1). Here a new set of connections is defined on the same grid configuration (using connections `northT`, etc.). All connections of the torus point to valid neighbor PEs, so the previously discussed data exchange problem does not exist here.

There is a list of extensions to this simple process of defining virtual machine structures: Several destination expressions may be specified after the arrow symbol, separated by commas. Connections may be *parameterized*, as for a highly symmetric topology like the hypercube. With these, it is possible to perform a data exchange in a *computed* direction. For the definition of the binary tree network (see Figure 2), *bi-directional* connections (‘ \leftrightarrow ’ or ‘ $\langle - \rangle$ ’ in ASCII notation) can be used instead of *uni-directional* connections (‘ \rightarrow ’ or ‘ \rightarrow ’). A bi-directional connection is an abbreviation for two uni-directional connections and, therefore, requires a second connection name on the right hand side of the connection.

So-called ‘compound connections’ may be used to have a case distinction inside a connection. The following example connects local pairs of PEs (see Figure 3).

A case distinction is made for the `next` connection. If the PE-number is odd, a connection to the right neighbor is established, while if it is even, a connection to the left neighbor is estab-



Tree definition:

```
CONFIGURATION tree [1..15];
CONNECTION lchild: tree[i] → tree[2*i];
           rchild: tree[i] → tree[2*i+1];
           parent: tree[i] → tree[i DIV 2];
```

Identical tree connection with bidirectional operator:

```
CONNECTION lchild: tree[i] ↔ tree[2*i] :parent;
           rchild: tree[i] ↔ tree[2*i+1]:parent;
```

Identical tree connection with parameter:

```
CONNECTION child[1]: tree[i] ↔ tree[2*i] :parent;
           child[2]: tree[i] ↔ tree[2*i+1]:parent;
```

Figure 2: Alternative binary tree definitions



```
CONFIGURATION list [1..8];
CONNECTION next: list[i] → {ODD (i)} list[i+1],
                    {EVEN(i)} list[i-1];
```

Figure 3: Compound connections

lished. Using compound connections, arbitrary connection structures, even with irregularities, may be defined.

Connection structures do not have to be 1:1 connections. For 1: n connections, an implicit broadcast is executed. In the following example, the first element of each row is connected to all elements in its row.

```
CONFIGURATION grid [1..100],[1..100];
CONNECTION one2many: grid[i,1] → grid[i,1..100];
```

If a one-to-many connection is to be established to all PEs of a dimension, the range may be substituted by an asterisk '*'. So the following is an equivalent connection.

```
CONNECTION one2many: grid[i,1] → grid[i,*];
```

However, for $n:1$ (many-to-one) connections (and the general many-to-many $m:n$ connections) one must ensure that only a single value arrives at any PE's entry port. Therefore, each data exchange operation may include a vector reduction, as will be discussed later. In the following example, all elements of a row are connected to the first element in their column.

```
CONNECTION many2one: grid[i,j] → grid[i,1];
```

3.2 Multiple Configurations and Iterative Connections

In addition to the constructs shown so far, the definition of multiple topologies in a program is possible. These may be defined independently of each other on separate groups of PEs – in which case the topologies may have different vector data structures. Or the topologies can be defined as ‘different views’ of the same set of PEs with identical data structure. Furthermore, local topologies may be defined in procedures, thus allowing semi-dynamic connection structures.

Different configuration definitions denote different sets of PEs. For example, the following declaration defines two *distinct sets* of PEs:

```
CONFIGURATION grid [1..200], [1..50];  
tree [1..10000];
```

On the other hand, configurations may be defined as a *different view* of the *same set* of PEs. In this case, the numbers of PEs have to be identical.

```
CONFIGURATION grid [1..200], [1..50];  
tree [1..10000] = grid;
```

Connections may be specified between multiple configurations:

```
CONFIGURATION grid [0..199], [0..49];  
tree [1..10000];  
CONNECTION mix: grid[i,j] → tree[i*50 + j];
```

The final extension is the use of iterative connection functions, as in the following definition for a hypercube network of arbitrary size (‘**’ denotes exponentiation, n is a constant):

```
CONFIGURATION hyper [0..(2**n-1)];  
CONNECTION FOR k := 0 TO n-1 DO  
    dir[k]: hyper[i] ↔ {EVEN(i DIV 2**k)}  
                    hyper[i + 2**k] :dir[k];  
END;
```

If n equals 10, there are 1,024 PEs defined together with ten bi-directional connections. Expression `EVEN(i DIV 2**k)` tests, whether the k -th bit of i equals 0.

A large program may be split into several modules, which are compiled separately. So, e.g. for a module containing library functions, it may be desirable *not* to specify the size of a configuration. When writing routines for image processing, the size of the grid structure should be left unspecified and will be defined later by the module importing these routines. An open configuration is indicated by using an asterisk ‘*’ instead of a value range. The configuration size may be determined dynamically at run time, e.g. by passing a parameter that is subsequently used as an upper bound in the configuration declaration.

```

DEFINITION MODULE Open;
CONFIGURATION grid[*],[*];
CONNECTION left: grid[i,j] <-> grid[i ,j-1] :right;
           up  : grid[i,j] <-> grid[i+1,j ] :down;
PROCEDURE sum_3x3(input: grid OF INTEGER): grid OF INTEGER;
END Open.

```

```

MODULE Main;
FROM Open IMPORT grid, sum_3x3;
CONFIGURATION my_grid = grid[1..10],[1..10];
VAR a,b: my_grid OF INTEGER;
BEGIN
  a := 1;
  b := sum_3x3(a);
  WriteInt(b,5);
END Main.

```

Open configurations are needed when a procedure is to work on a vector of unspecified size, but has to make use of connections for data exchange or position data. If connections and position data are not required in a procedure which is to be used for different configurations (different size or arbitrary configuration), then the simpler concept of generic vector parameters may be used (see data declaration below).

3.3 Data Declaration

Parallaxis differentiates between scalar and vector variables in data declarations as well as in procedure parameters and results. Scalar data is placed on the control processor, while vectors are distributed component-wise among the virtual PEs (see Figure 4). The configuration name is used as part of the data type of a vector variable.

```

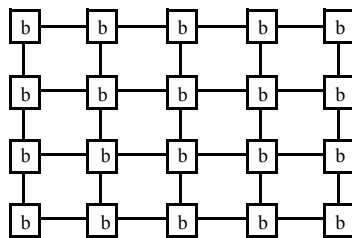
VAR a: INTEGER;      (* scalar *)
    b: grid OF REAL; (* vector *)
    c: tree OF CHAR; (* vector *)

```

on control
processor



distributed on PEs



distributed on PEs

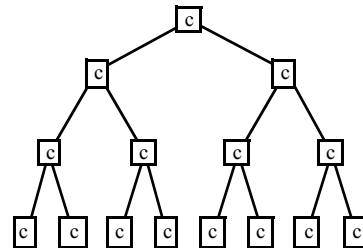


Figure 4: Declaration of scalar and vector data

Unfortunately, strict typing has an annoying effect on procedure arguments. Imagine, e.g. writing a function `factorial`, for computing the factorial value for an argument of type `INTEGER`. Now, a `factorial` function would have to be declared for scalar arguments, and for *every configuration defined* in a program (e.g. for `scalar`, `grid` and `tree` in Figure 4). Since there is no way of knowing them in advance, it would be impossible to write general library routines. To remedy this situation, parameters and variable declarations inside such a procedure may use the keyword `VECTOR` instead of a particular configuration name. This indicates that a parameter will be used in a parallel computation, without specifying a particular configuration (this

results in a *generic procedure*). All parameters declared as generic vectors or variables in such a procedure have to belong to the *same* configuration. Since no particular configuration has been specified, no data exchange may be performed in such a procedure.

```

PROCEDURE s_factorial(a: INTEGER): INTEGER;
VAR b: INTEGER; (* scalar *)
...
END s_factorial;

PROCEDURE v_factorial(a: VECTOR OF INTEGER): VECTOR OF INTEGER;
VAR b: VECTOR OF INTEGER; (* any vector *)
...
END v_factorial;

```

A procedure may contain several different open vectors or open configurations, and may also define local variables of the same "open type", as shown in the following example:

```

PROCEDURE vec(a: VECTOR OF INTEGER; b: VECTOR OF INTEGER);
VAR x: VECTOR a OF INTEGER; (* vector type corresponding to a *)
    y: VECTOR b OF INTEGER; (* vector type corresponding to b *)
...
END vec;

CONFIGURATION grid[*],[*];

PROCEDURE open(a: grid OF REAL; b: grid OF REAL);
VAR x: grid b OF REAL; (* vector type corresponding to b *)
...
END open;

```

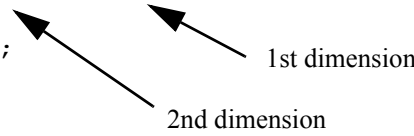
3.4 Processor Positions

There are two ways to determine a PE's current position. The first is by using the (vector-valued) standard function `ID`, which returns the virtual processor position as a single number in row-major ordering (or 'highest-dimension-major' for more than two dimensions):

```

CONFIGURATION grid [1..4],[ -2..+2];
...
VAR x: grid OF INTEGER;
...
x := ID(grid);

```



This results in each component of x being assigned the number of its virtual PE, always starting with 1, independent of configuration range and number of dimensions:

$$ID(\text{grid}) = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 6 & 7 & 8 & 9 & 10 \\ 11 & 12 & 13 & 14 & 15 \\ 16 & 17 & 18 & 19 & 20 \end{bmatrix}$$

The second way of determining the position of a virtual PE, now with respect to its configuration declaration, is to use the standard function `DIM`. This function takes the configuration name and the number of the dimension as arguments and returns the position of a PE within this dimension. Dimensions are numbered from right to left, that is, the highest dimension is at the left-most position.

The position values returned by function DIM match the ranges of the configuration definition. The following shows position data for rows and columns of the grid example:

$$\text{DIM}(\text{grid}, 2) = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 2 & 2 & 2 & 2 & 2 \\ 3 & 3 & 3 & 3 & 3 \\ 4 & 4 & 4 & 4 & 4 \end{bmatrix} \quad \text{DIM}(\text{grid}, 1) = \begin{bmatrix} -2 & -1 & 0 & +1 & +2 \\ -2 & -1 & 0 & +1 & +2 \\ -2 & -1 & 0 & +1 & +2 \\ -2 & -1 & 0 & +1 & +2 \end{bmatrix}$$

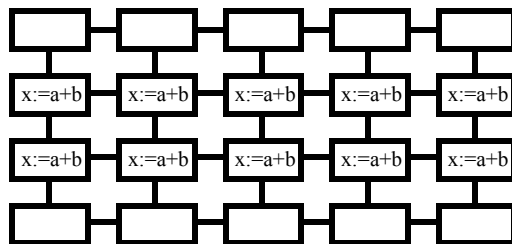
Functions ID and DIM are complemented by functions LEN (size of a dimension), UPPER/LOWER (upper or lower bounds of a dimension), and mapping functions DIM2ID/ID2DIM.

An individual PE may be selected by using a component expression with scalar arguments after the identifier of a vector variable:

```
VAR x : grid OF INTEGER; (* 2-dim. *)
    s,t: INTEGER;        (* scalar *)
...
s := x <<12>>;          (* get the value of the PE with ID 12 *)
x <<12>> := s;          (* set the value of the PE with ID 12 *)
s := x <:3,t+1:>;      (* get value of PE in row 3 and col. t+1,
                       according to CONFIGURATION ranges *)
x <:t,1:> := s;        (* set value of PE in row t and column 1 *)
```

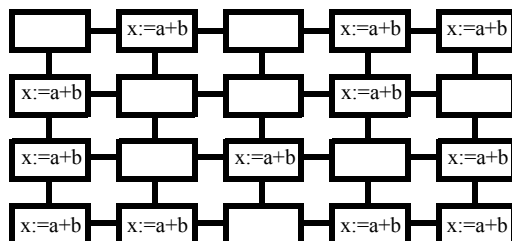
3.5 Parallel Execution

Parallel execution is implicit in Parallaxis-III, depending on the declaration of variables involved in a statement or expression. PE-selection (determining which PEs will be active during a certain statement) is also implicit. Any selection or iteration instruction (IF, FOR, WHILE, REPEAT, CASE, LOOP) with a vector argument may be used. Figure 5 shows the data parallel execution of a statement on a selected group of PEs. The selection criteria can involve PE positions or local data.



selection depends on PE position

```
VAR x,a,b: grid OF REAL;
...
IF DIM(grid,2) IN {2,3} THEN
  x := a+b
END;
```



selection depends on local data

```
VAR x,a,b: grid OF REAL;
...
IF x>0 THEN
  x := a+b
END;
```

Figure 5: Data parallel instruction

So whenever a selection is performed, e.g. by an `IF` statement with vector condition, only those PEs are active during execution of the `THEN` branch, whose local condition evaluates to `TRUE`. A `THEN` branch or an `ELSE` branch will only be executed if the condition (or its negation, respectively) will be satisfied by at least one PE. In the general case, when the condition evaluates to `TRUE` for some PEs, but evaluates to `FALSE` for some other PEs, then both `THEN` branch and `ELSE` branch will be executed subsequently (first `THEN`, afterwards `ELSE`) with the appropriate group of PEs being activated. This also holds for any scalar statements that may be contained in these branches. If vector `IF` statements are nested, then in the inner level only a subset of the PEs of the corresponding outer level can be active.

Example:

```
VAR x: grid OF INTEGER;
...
IF x>5 THEN x := x - 3
        ELSE x := 2 * x
END;
```

Execution:

<i>PE-ID:</i>	<i>1</i>	<i>2</i>	<i>3</i>	<i>4</i>	<i>5</i>	<i>(‘-’ denotes inactive)</i>
initial values of x:	10	4	17	1	20	
starting <i>then</i> -branch:	10	–	17	–	20	
after <i>then</i> -branch:	7	–	14	–	17	
starting <i>else</i> -branch:	–	4	–	1	–	
after <i>else</i> -branch:	–	8	–	2	–	
<i>selection done</i>						
after <i>if</i> -selection:	7	8	14	2	17	

The possible subsequent execution of `THEN`- and `ELSE`-branches may lead to unexpected side effect, which are shown in the following program.

Example:

```
VAR x: grid OF INTEGER;
    s: INTEGER, (* scalar *)
...
IF x>5 THEN x := x - 3; INC(s);
        ELSE x := 2 * x; INC(s);
END;
```

Execution:

<i>PE-ID:</i>	<i>s</i>	<i>1</i>	<i>2</i>	<i>3</i>	<i>4</i>	<i>5</i>	(<i>'*</i> ' active, <i>'-</i> ' inactive)
initial values of <i>s</i> :	1	*	*	*	*	*	
after <i>then</i> -branch:	2	*	-	*	-	*	
after <i>else</i> -branch / <i>if</i> :	3	-	*	-	*	-	

When entering a loop with vector condition (e.g. `WHILE` loop), only those PEs are active which satisfy the condition. In subsequent iterations of this loop, the number of PEs is *always decreasing*. The loop iterates until no PE is left to satisfy the loop condition.

Example:

```

VAR x: grid OF INTEGER;
...
WHILE x>5 DO
  x:= x DIV 2;
END;

```

Execution:

<i>PE-ID:</i>	<i>1</i>	<i>2</i>	<i>3</i>	<i>4</i>	<i>5</i>	(<i>'-</i> ' denotes inactive)
initial values of <i>x</i> :	10	4	17	1	20	
starting 1st iteration:	10	-	17	-	20	
after 1st iteration:	5	-	8	-	10	
starting 2nd iteration:	-	-	8	-	10	
after 2nd iteration:	-	-	4	-	5	
starting 3rd iteration:	-	-	-	-	-	
<i>loop terminates</i>						
after loop:	5	4	4	1	5	

The parallel `WHILE`-loop requires an implicit reduction operation - otherwise it would not be possible to determine when the loop terminates. This information has to be relayed from the vector side to the scalar side, which deals with the actual loop iteration. So the same `WHILE`-loop could be rewritten with an explicit reduction to make the argument a scalar and thereby use a standard scalar `WHILE`-loop. In that case, however, an additional vector `IF`-selection has to take care of the correct PE activation:

```

WHILE REDUCE.OR(x>5) DO      (* OR-reduction: continue loop iteration *)
  IF(x>5) THEN x:= x DIV 2 END (* while at least one PE is left *)
END;

```

Other control structures, known from sequential Modula-2 may be used in vector context as well. The `CASE`-selection can be treated as a nested chain of `IF-THEN-ELSIF`-selections, while

FOR- and REPEAT-loops can be regarded as modifications of a WHILE-loop. An ALL-statement allows to reactivate all elements of a configuration within a selection or loop.

3.6 Structured Data Exchange

Data exchanges between processors can be accomplished with simple symbolic names, thanks to the network declaration described earlier. Data exchange of a local vector variable between all or just a group of PEs can be invoked by calling system function MOVE with the name of a previously defined connection. Only active PEs participate in a data exchange operation. Figure 6 shows an example of a data exchange in the grid structure defined previously. The expression returns vector variable x shifted one position to the east.

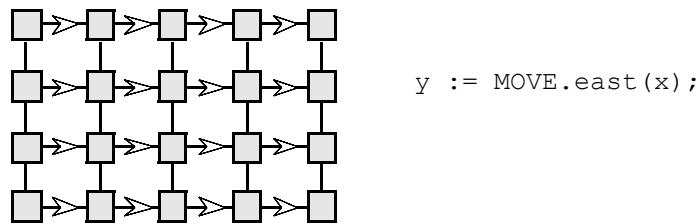


Figure 6: Synchronous data exchange

For the data exchange operation shown above, sender-PE *and* receiver-PE of a data exchange have to be active. For the operations SEND and RECEIVE shown below, it is sufficient for *only* the sender (or *only* the receiver, respectively) to be active. These operations are especially needed for the data exchange between different topologies. Unlike the other data exchange operations, SEND is a procedure (not returning a value) and therefore takes two arguments, first the expression to be sent, and second the variable to receive the expression.

```

SEND.east(4*x, y);
y := RECEIVE.north(x);

```

Additional data exchange modifiers may be specified for some of the data exchange operations. Data can be moved several steps at once along a defined connection, and incoming data can be reduced to a single value for $n:1$ connections. For details see [Bräunl 96].

```

SEND.right:2 (x,y);          (* move data two steps to the east *)
w := MOVE.parent:#SUM (u);  (* in tree: move data from children to
                             parent; reduce data by adding *)

```

Configuration boundaries often cause trouble in data parallel programming, for they frequently require special treatment to avoid undefined data. This is not the case for Parallaxis. Here, it is allowed to send data outside a configuration boundary and try to receive data from beyond the boundary. After initializing the send-expression with the vector parameter value supplied, data sent outside a configuration is deleted, while an attempt to read from outside leaves the particular PE's data unchanged. This approach avoids undefined values during a data exchange operation, while deletion of boundary data is intentional for many applications.

3.7 Unstructured Data Exchange

Structured data exchange makes application programs easy to write and understand. In some cases it also makes them faster, when better use can be made of the physical connection struc-

ture of a particular parallel system. However, it may be desirable to perform an unstructured data exchange. This reflects an arbitrary permutation of the components of a vector variable, which may be difficult to write down using structured data exchanges only.

For example, each component of a two-dimensional vector (a matrix) is to be sent to a destination address, which is being computed at run-time. When only structured data exchange is possible, e.g. via a grid, one has to program a communication procedure which shifts the matrix elements in several steps over the grid. This approach will work, however, some parallel computer systems have a global connection structure, which allows an arbitrary unstructured data exchange. In this case, specifying direct destination addresses for each component of a vector variable may result in a faster program. Despite the availability of specialized commands for unstructured data exchange, execution may be quite expensive. For example, a grid operation at the MasPar MP-1 requires about the same time as a simple arithmetic operation (addition), but a non-grid data exchange takes about 100 times longer to execute.

In our programming language syntax, the unstructured data exchange is still a machine-independent operation. If a certain data parallel architecture does not provide a general communication structure, then this data exchange will be routed transparently over the simpler network provided (e.g. a grid or a ring) taking several execution steps.

In Parallaxis, the `SEND` and the `RECEIVE` operations may take an index expression instead of a connection name. As before, when using `SEND`, only active PEs send data, and when using `RECEIVE`, only active PEs receive data. However, these two operations differ in their index semantics, as is shown for an example in Figure 7. In order to avoid confusion, operation `MOVE` may not be used with an index expression.

```
VAR x,y,index: grid OF INTEGER;
...
SEND.<<index>>(x,y);      sends data from all components of x to a destina-
                           tion, determined by vector index

y := RECEIVE.<<index>>(x); receives data from all components of x to a desti-
                           nation, determined by vector index, however, on
                           the receiver's side
```

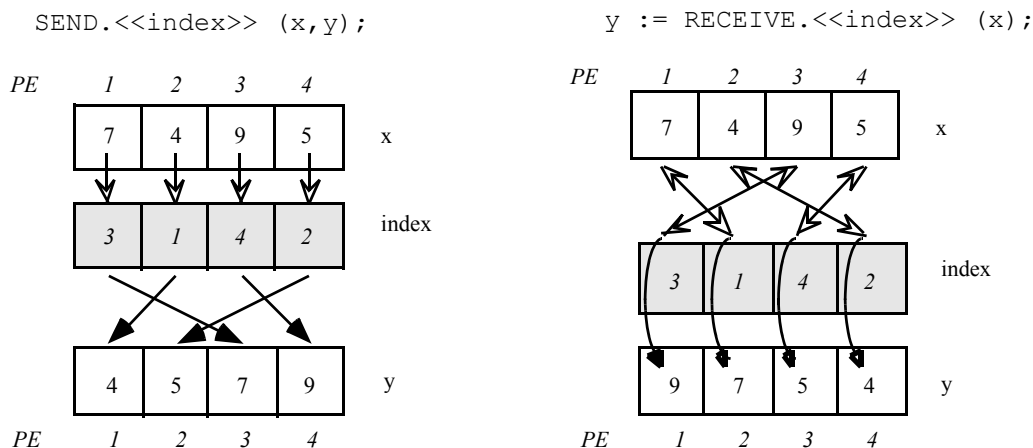


Figure 7: Unstructured data exchange

Besides using a single index, referring to the `ID` position of PEs, several indices referring to `DIM` positions may be used as well. Also, this kind of data exchange does not have to be a one-

to-one correspondence. If several indices refer to the same PE position, `RECEIVE` (one-to-many) becomes a broadcast, while `SEND` (many-to-one) has to specify a reduction operation for resolving collisions or an arbitrary component will be selected. For details see [Bräunl 96].

3.8 Reduction

The reduction of a vector to a scalar is another important operation. The `REDUCE` operation handles this task in conjunction with a system-defined or user-defined (programmable) reduction operation (see Figure 8). System-defined operators are:

`SUM, PRODUCT, MAX, MIN, AND, OR, FIRST, LAST`

The operators `FIRST` and `LAST` return the value of the first or last currently active PE, respectively, according to its identification number (`ID`). All other reduction operators' functions can easily be deduced from their names. In the optimal case the execution of a reduction operation requires about $\log_2 n$ time steps for a vector with n active components. However, this time estimation depends on the physical connection structure of the PEs.

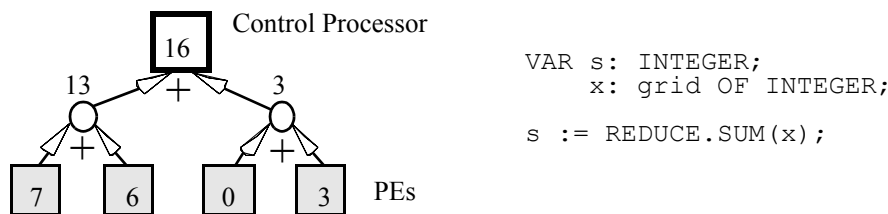


Figure 8: Vector reduction in Parallaxis

The `REDUCE` operation can also be called with a user-defined function. Such a function has to have two vector input parameters and has to return a vector value of the same type. Note that the reduction function implemented by the user should be associative and commutative, or unpredictable results may occur, e.g. $(1 - 2) - 3 \neq 1 - (2 - 3)$.

There are a few places, where substituting a scalar constant in lieu of a vector variable makes sense, but lacks information about the configuration to be used. Consider the problem of counting the number of active PEs for some configuration. Instead of using a vector variable, the constant 1 can be used for each PE, however, it has to be type cast to the appropriate configuration:

```
s := REDUCE.SUM( grid(1) ); (* return no. of currently active PEs *)
```

3.9 Exchange between Scalar and Vector Data

Communication between the control processor and the parallel PEs is done via system procedures in Parallaxis. Transferring a scalar field into a parallel vector is invoked with procedure `LOAD`, while transferring data back into a scalar field from a vector is accomplished with `STORE` (see Figure 9). Only active PEs participate in this sequential data exchange. `STORE` with inactive PEs does not result in gaps in the scalar array, but data elements are stored sequentially. `LOAD` with inactive PEs assigns the next array value to the next active PE, no scalar array elements will be skipped. Surplus elements will not be used, too few elements leave the corresponding array elements (or vector components, respectively) unchanged. The execution of this operation usually requires n time steps for a data array with n elements. A scalar integer

variable may be specified as an optional third parameter for `LOAD` and `STORE`, which limits the number of data items transferred and also receives the number of data items actually transferred after the operation.

```

CONFIGURATION list[1..n];
VAR  s: ARRAY[1..n] OF INTEGER;
     t: INTEGER;
     v: list OF INTEGER;
...
LOAD (v, s);      (* from scalar to vector *)
STORE(v, s);     (* from vector to scalar *)
STORE(v, s, t);  (* here, t becomes num. active PEs *)
                  (* require n steps each *)

```

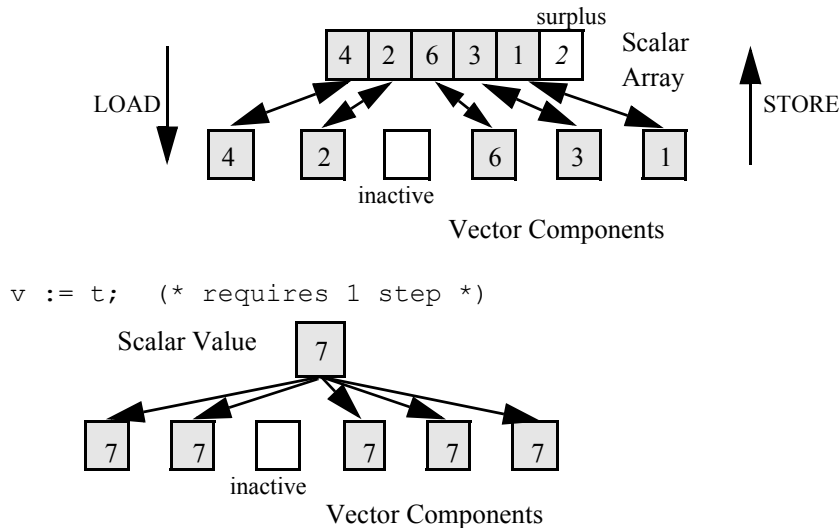


Figure 9: Data exchanges between PEs and control processor

Figure 9 (bottom) also shows an assignment in which a (constant or variable) scalar data value is copied into all or a group of PEs. Every component of the vector contains the same value as the scalar. This operation is implemented by an implicit *broadcast* and therefore requires only a single time step.

4. Programming Tools

The environment for Parallaxis-III comprises several compilers and a source-level debugger. The compilers generate code for parallel and sequential systems (the latter in simulation mode). The parallel debugger includes features for vector data visualization and for performance analysis.

Figure 10 shows the interaction of the Parallaxis tools (shaded boxes) with standard Unix tools (white boxes) on workstations and the MasPar massively parallel system.

4.1 Compiler P3

Here, the compiler for generating sequential C-code (simulation system) is discussed, which is complemented by compilers generating parallel code for MasPar MP-1/MP-2 [MasPar 91] and Connection Machine CM-2 [Thinking Machines 89]. We have also experimented with further code generators not discussed in this paper for Intel Paragon and workstation clusters using

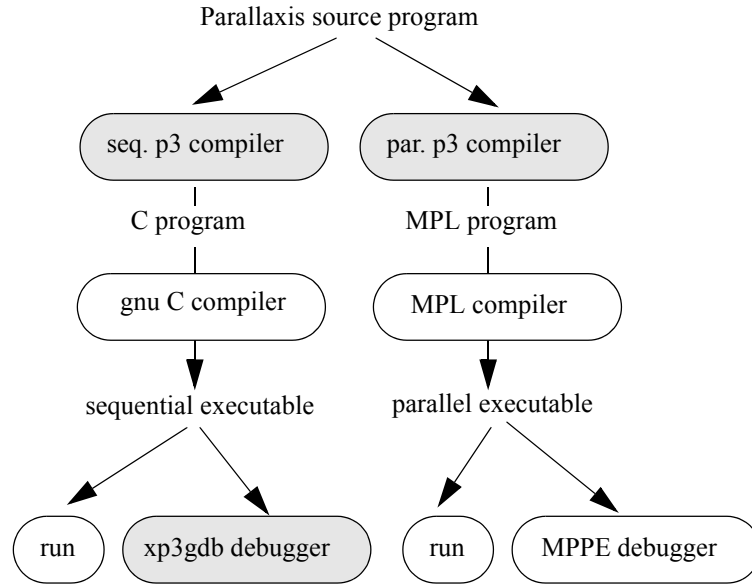


Figure 10: Parallaxis Tools

PVM [Geist et al. 94] (parallel virtual machine). The compilers generate C code, so a subsequent compilation step is necessary to generate object code.

The *Cocktail* compiler construction tools from GMD/Univ. Karlsruhe [Grosch 95] have been used to build the Parallaxis-III compilers. The compiler option list is shown in Figure 11.

The configurations of Parallaxis, i.e. the PEs, are implemented by linear arrays. Each configuration keeps track about which virtual PEs are active. This is called the "active-set" of a configuration.

4.2 Graphics Source Level Debugger xp3gdb

A compiler by itself is neither sufficient for parallel program development nor for educational purposes. Therefore, we developed a source level debugger for Parallaxis. Rather than starting from scratch, we used the gnu debugger *gdb* and its graphics interface *xxgdb* as a base. This standard C debugger had to be adapted to behave as if being a Parallaxis source level debugger. This affects not only the source line window and the positioning of break points, but also (and more difficult) the presentation of Parallaxis data types, especially vector data. Figure 12 shows a typical sample debugging session.

We added a number of graphics facilities. Especially for large vectors (e.g. two-dimensional images or simulation data), it is not very entertaining to examine large lists of data. Instead we provided the possibility to look at vector data *directly* in a graphics window. One- or two-dimensional data is displayed in a window with little boxes representing individual PEs (Figure 13). Each box is colored and contains two items of information:

- PE activity
if a PE is active it is represented by a filled square,
if it is inactive it is drawn as a hollow square
- PE data
each PE is drawn in a color representing its data value
(according to the value range bar on top, either as rainbow color or gray scale range).

```

NAME
    p3 -- Parallaxis-III Compiler User Interface V0.5

DESCRIPTION
    Compile some Parallaxis-III programs and call backend compiler.

SYNOPSIS
    p3 [options] [file] ...

OPTIONS
    -C                Generate C-code for simulation (default)
    -casts            Generate type casts to make C-programs lint free
    -cc name          Name of the backend compiler to use
    -g                Generate debug code (also passed to backend compiler)
    -h, -H, -help    Print this usage
    -headers          Generate header files for imported modules
    -Ipath            Add path to import/include list (Par. and backend)
    -indent i        Set indent of generated code to i blanks
    -koption          Pass option directly to backend compiler
    -Lpath            Add path to library path (backend only)
    -m, -mem          Print statistics about used memory
    -MPL, -mpl        Generate MPL-code for MasPar
    -n, -nocompile   Don't compile, just show commands (implies -v)
    -nop3inc          Don't use standard include paths
    -nop3lib          Don't use standard library paths
    -nodefaults      Same as -nop3inc -nop3lib
    -o name           Name of the generated executable
    -p                Parallaxis compile only, don't call backend compiler
    -c                Paral. and backend compile only, don't call linker
    -Ppath            Add path to import list (Parallaxis only)
    -PVM, -pvm        Generate PVM-code for Paragon
    -r, -rchecks     Don't generate runtime range checks
    -s, -small        Generate small MPL-only model (max. 128KB)
    -t, -time         Print statistics about used time (Parallaxis only)
    -tt, -total       Like -t, but also for backend compiler
    -v                Print version of p3 and the resulting compiler calls
    -vv              Like -v, passes also -v to backend compiler
    -w                Don't generate warnings
    -ww              Like -w, passes also -w to backend compiler

OPTIONS ONLY AVAILABLE DURING DEVELOPMENT
    -Zw              Write code tree
    -Zs              Write symbols tree
    -Zq              Query code tree
    -Zc              Check code tree
    -Z1              Run parser only, no semantic check
    -Z2              Run parser and semantic check only, no code generation

    Every other option is passed unchanged to the backend compiler.

ENVIRONMENT
    P3CC              Name of the backend compiler
    P3INC             ":"-separated list of paths where to find sources
    P3LIB             ":"-separated list of paths where to find libraries
    P3OPT             Default options always to set

```

Figure 11: Compiler options

Position numbers may be added and the data range may be specified. The vector window can display a static state (command *print*) or adapt dynamically to changing data (command *display*).

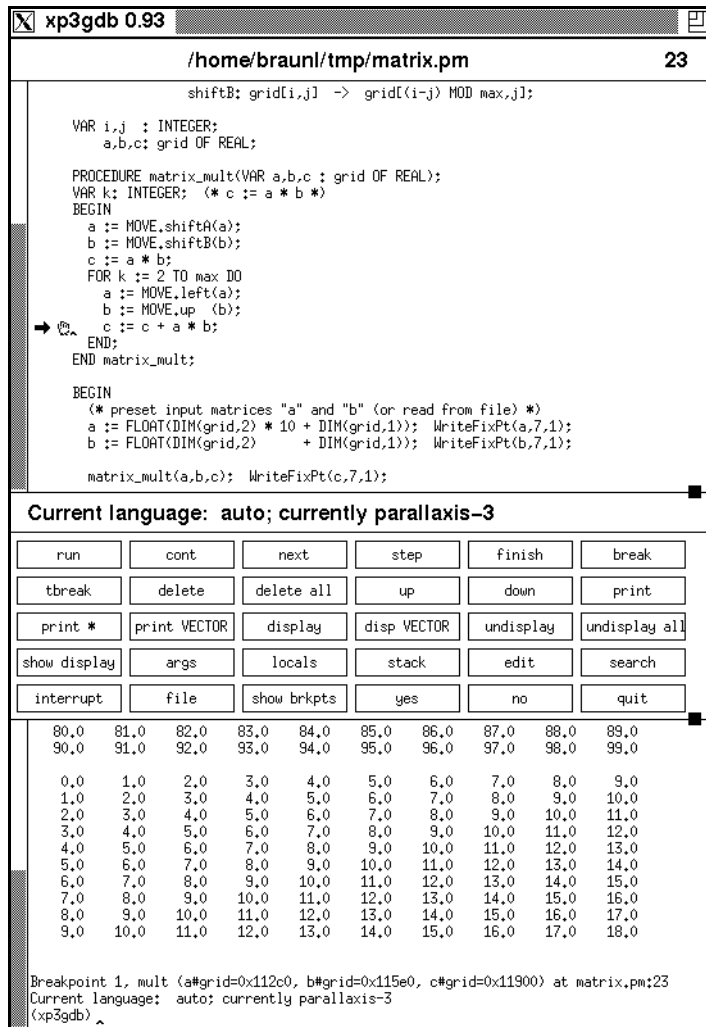


Figure 12: Debugger Control Window

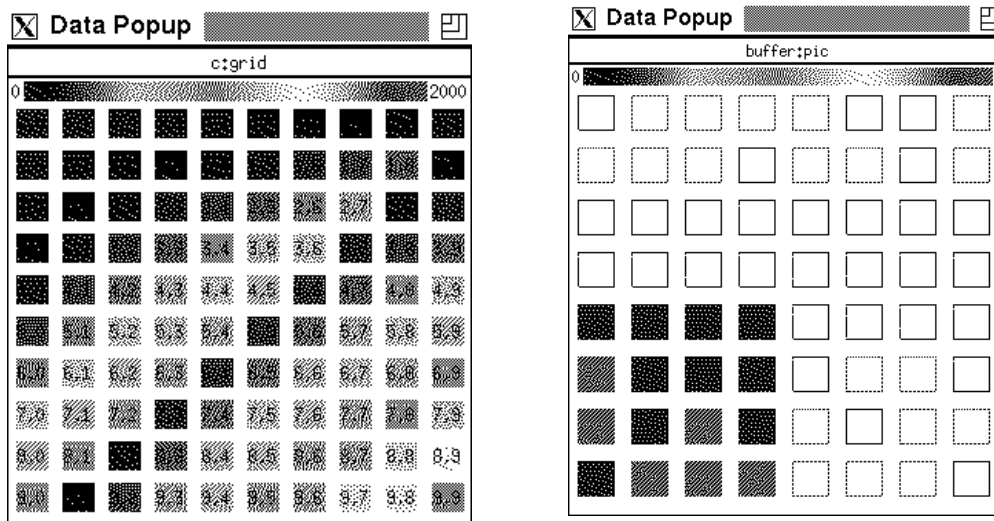


Figure 13: Vector data display

The PE usage may also be displayed graphically. Here, the program is executed in single step mode and the number of active PEs is determined at each step. Due to the overhead of stepwise evaluation, execution time slows down when using this feature. The PE usage values produce a tell-tale curve of the application program's parallel characteristics and are a valuable help in localizing critical program regions for optimization of the execution time. Figure 14 shows the PE usage curve for the prime sieve sample program.

```

MODULE prime;
CONFIGURATION list [2..200];
CONNECTION (* none *);
VAR next_prime: INTEGER;
    removed : list OF BOOLEAN;
BEGIN
  REPEAT
    next_prime:= REDUCE.FIRST(DIM(list,1));
    WriteInt(next_prime,10); WriteLn;
    removed := DIM(list,1) MOD next_prime =0
  UNTIL removed
END prime.

```

This tiny program represents the parallel version of the sieve of Eratosthenes. The list of active PEs resembles the candidates for prime numbers not yet removed. In the beginning all PEs are active, which is reflected by the initial peak in Figure 14. But in each step of the REPEAT loop, variable `removed` becomes true for all multiples of the next prime found, so corresponding PEs will no longer be active in the next iteration of the loop. This explains the rapid decrease in the PE usage diagram.

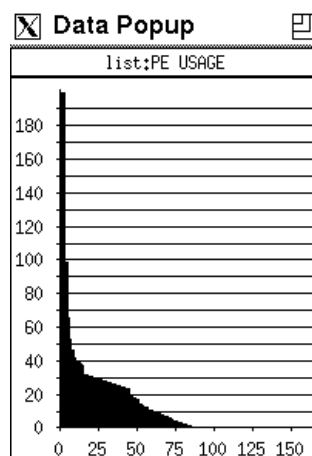


Figure 14: PE usage diagram

The xp3gdb debugger comprises the most important concurrent debugging and visualization features:

- Breakpoints and single stepping through data parallel program
- Color visualization of vector and matrix data
- Graphical PE activity / usage efficiency display

These features are to some extent similar to those in commercial parallel debugging tools, e.g. Prism for Connection Machine (now owned by Sun Microsystems) and MasPar's MPPE, which is a complete programming environment.

5. Sample Application

A typical data parallel application is discrete time simulation. Here, we present a model of very simple behavior of cars on a single lane street as in Figure 15. A number of cars start at equidistant positions, while the street is closed to a circle. Each car determines its acceleration/deceleration according to the space in front of it plus a certain random factor (positive or negative). If the car concentration exceeds a certain threshold, sudden and unmotivated traffic jams occur.

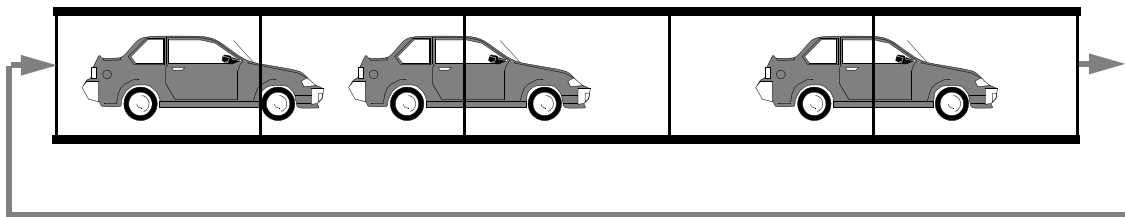


Figure 15: Traffic model

For this simulation in Parallaxis, two disjoint configurations have been used, one configuration for cars and one for street segments. Cars may not take over each other, so they keep their linear order. The street is modeled as a closed ring and is only needed for display purposes.

```
CONFIGURATION cars[0..max_cars-1];
CONNECTION
  next: cars[i] <-> cars[(i+1) MOD max_cars] :back;

CONFIGURATION street[0..width-1];

VAR pos, dist,
    speed, accel: cars OF REAL;
    collision    : cars OF BOOLEAN;
    my_car      : street OF BOOLEAN;
    time, z     : INTEGER;
```

At initialization all cars are started at equal distance across the street, which is expressed by:

```
pos := FLOAT(DIM(cars,1)) / FLOAT(max_cars);
```

The simulation itself is one big FOR-loop, which generates one graphics line for each iteration. If there is sufficient space in front of a car, it accelerates up to a maximum speed by a constant value plus a small random term. The randomness prevents all cars from maintaining identical distances from each other. Collisions are detected in parallel by measuring the distance of all pairs of subsequent cars. They cause a sudden stop, from which the cars can again accelerate in the subsequent simulation step. The integration required for determining velocity and position from acceleration has been simplified to summation.

```
FOR time := 1 TO steps DO
  ... (* show "collision" at current line *)
  my_car := DIM(street,1) = TRUNC(pos<:0:> * FLOAT(width));
  ... (* show "my_car" at current line *)
```

```

dist := MOVE.back(pos) - pos;
IF dist < 0.0 THEN dist := dist + 1.0 END; (* close street to loop *)

collision := dist < 0.0;

IF collision THEN speed := 0.0;
ELSE (* no collision, accelerate *)
  accel := max_accel + rand_fac * (RandomReal(cars)-0.5);

  (* brake, if necessary *)
  IF dist < min_dist THEN accel := - max_accel END;

  (* update speed, apply speed limit *)
  speed := min(speed + accel, max_speed);

  (* do not back up on autobahn ! *)
  IF speed < 0.0 THEN speed := 0.0 END;

  (* update position *)
  pos := pos + speed;

  (* leaving right, coming back in left *)
  IF pos >= 1.0 THEN pos := pos - 1.0 END;
END;
END;

```

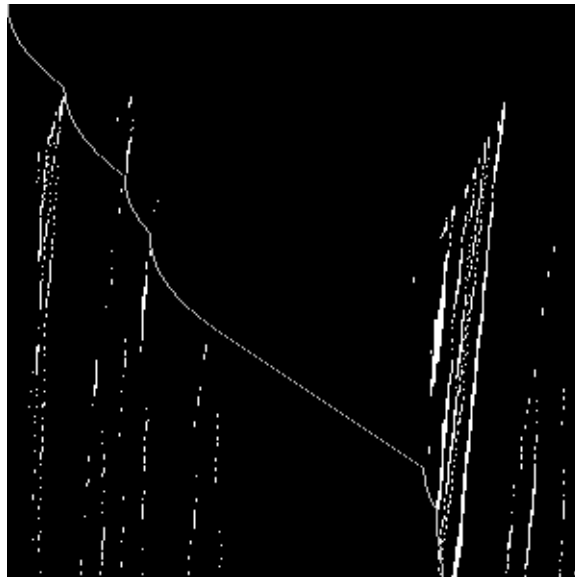


Figure 16: Simulation of traffic congestion

A sample simulation run of the traffic program is shown in Figure 16. The street is displayed as a horizontal line, while time flows from top to bottom in the figure. Standing cars are marked as bright spots. Also, the route of one individual car is shown, starting in the upper left corner. It is easy to recognize the acceleration phase of the individual car (parabolic curve), leading to a phase of continuous speed (straight line). Sudden breaks occur due to heavy traffic, simply caused by too many cars on the street. Several spontaneous traffic jams occur in this simulation, all slowly propagating in the direction opposite to the driving direction. Some congestions are increasing, while others are decreasing.

6. Comparison with other Data Parallel Languages

All current procedural data parallel languages are extensions of existing sequential languages, i.e. Pascal, Modula-2, C, or Fortran. The following list discusses some of them with their differences and similarities to Parallaxis. All languages provide transparent mapping between logical PEs (domains, vectors, or whatever they are called) and physical processors. While most languages support multi-dimensional parallelism, none of them addresses the definition of the data exchange (or network) structure.

C* was originally developed by J. Rose and G. Steele [Rose, Steele 87] as a data parallel extension of C++ for programming the Connection Machine [Hillis 85]. However, in 1990 the language was completely redefined - still keeping the name C* (suffix "Version 6"), but containing totally new language constructs and being based on ANSI-C [Thinking Machines 90]. Several concepts are similar to Parallaxis. C*'s data declaration distinguishes between host and vector PEs; vector-to-scalar reduction is performed by overloading standard C operators. The "shape" definition (until version 5 called "domain") allows declaration of vector structures similar to Parallaxis' configurations. A "with" statement has been introduced to select a "shape" for data parallel operations. As has been mentioned before, this is not necessary in Parallaxis, since it can use the corresponding data configuration defined for each variable. C* has no equivalent for Parallaxis' connections. Data exchange is handled via indices in C*: each PE has to compute its neighbor's position and may then perform a send or receive operation by using the syntax of an assignment operation. The network structure is transparent to the C* user; data exchange may be performed between arbitrary PEs while the Connection Machine's operating system has to route unstructured data exchange operations through its physical hypercube structure.

Fortran 90 [Metcalfe, Reid 90] was the long awaited successor of Fortran-77, which among numerous other changes and additions also contained language constructs for data parallel processing. Fortran 90 contains array constructs which simplify vector and matrix operations by using arrays. Data parallel operators are overloaded to standard arithmetic operators, so two array variables can be added or multiplied by using the same syntax as for scalar variables. Since the language constructs are machine-independent, sequential and parallel compilers exist. Unfortunately, data parallel features are still very limited in Fortran 90, so compilers could not efficiently utilize the parallel hardware. This led to the development of High Performance Fortran HPF [Koelbel et al. 1993].

PASM Parallel-C [Kuehn, Siegel 85] is similar to C*. The PASM multicomputer is partitionable into several independent SIMD and/or MIMD machines. Consequently, Parallel-C contains constructs accounting for both, SIMD and MIMD operation modes. Parallel data-declaration is similar to ordinary C's array declaration with multi-dimensional structures being allowed. SIMD parallel execution is performed either by selecting PEs using "*address masks*" or by using an if-statement with parallel test condition.

Refined C [Dietz, Klappholz 85] and **Refined Fortran** [Dietz, Klappholz 86] have been called "*sequential languages for parallel programming*" by their authors. The languages are not, however, oriented towards the explicit expression of data level parallelism, but to build an optimizing compiler to *extract* control level parallelism by applying data flow analysis.

Vector C [Li 86] is another C superset for high level SIMD programming. Parallel execution may be expressed by using *vector* variables (which are modified C arrays) in expressions with special *vector operators*. The language concepts have similarities to the language APL with its explicit vector operators.

VCode [Blleloch, Chatterjee 90] is an approach for a common low level data parallel intermediate language. It tries to take advantage of the large similarities among all data parallel languages by providing a common low level basis for compilation. This would simplify implementation and porting for a whole family of data parallel languages. The languages C*, Paralation-Lisp and Fortran 90 are targeted, implementations exist for Connection Machine and Cray Y-MP. Since all data parallel languages are similar, a compiler based on VCode could also be constructed for Parallaxis.

Braid [West, Grimshaw 94] is an extension of the object oriented language **Mentat**, trying to integrate both, task level concurrency and data parallel extensions. The language allows the declaration of complex data parallel operations, which are handled by the compiler via iterations within the data set. So far the language has only been implemented on MIMD machines.

DAPPLE [Kotz 94] stands for "Data-parallel programming library for education". Unlike most other approaches, it is not a language by itself, but rather a library which is linked to a C++ program. Libraries are quite common in task level concurrency, but are not in wide use for data parallel applications. The library provides vectors and matrices as classes, which can be operated on using the standard C++ operators via overloading. The advantage of this approach is its simplicity in implementation (no new compiler is necessary) and application (no new language has to be learned). However, due to the limitations of the base language, it lacks some expressive power, which makes it difficult to formulate basic data parallel constructs, like parallel `IF`, parallel `WHILE`, etc.

Summary

We have presented Parallaxis-III, a machine-independent data parallel programming language. Programs in Parallaxis can be compiled for parallel and sequential architectures, which makes it an ideal tool for education and algorithm development. The source level debugger for Parallaxis allows the examination and visualization of vector data sets through the use of graphics and color. Also provided is efficiency data in form of PE load diagrams from current and past PE activity data. A typical data parallel application in discrete simulation has been shown.

Parallaxis has been proven to be a useful tool in education environments. Numerous universities have selected Parallaxis in courses on concurrency for teaching basic data parallel concepts in a structured environment. The Parallaxis simulation system and debugger have successfully been used in a large number of lab courses and for a wide range of applications.

Acknowledgments

The author would like to acknowledge the implementation work of all students involved in the Parallaxis-III project at Univ. Stuttgart, Germany, especially Jörg Stippa (debugger), Eduard Kappel, Harald Lampke, and Hartmut Keller (compiler).

References

[Blleloch, Chatterjee 90] G. Blleloch, S. Chatterjee, *A Data-Parallel Intermediate Language*, Proceedings Frontiers of Massively Parallel Computation, Oct. 1990, pp. 471-480 (10)

- [Bräunl 89] T. Bräunl, *Structured SIMD Programming in Parallaxis*, Structured Programming, vol. 10, no. 3, July 1989, pp. 121-132 (12)
- [Bräunl 91] T. Bräunl, *Designing Massively Parallel Algorithms with Parallaxis*, Proceedings of the 15th Annual International Computer Software & Applications Conference, compsoc91, Tokyo, Sep. 1991, pp. 612-617 (6)
- [Bräunl 93] T. Bräunl, *Parallel Programming – An Introduction*, Prentice Hall, Englewood Cliffs NJ, 1993
- [Bräunl 96] T. Bräunl, *Parallaxis-III User Manual*, Computer Science Report, no. 1996/08, Univ. Stuttgart, June 1996, pp. (V+53)
- [Dietz, Klappholz 85] H. Dietz, D. Klappholz, *Refined C: a sequential language for parallel programming*
Proceedings 1985 International Conference on Parallel Processing, IEEE Computer Society, Aug. 1985, pp. 442-449 (8)
- [Dietz, Klappholz 86] H. Dietz, D. Klappholz, *Refined Fortran: another sequential language for parallel programming*
Proceedings 1986 International Conference on Parallel Processing, IEEE Computer Society, Aug. 1986, pp. 184-191 (8)
- [Geist et al. 94] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, V. Sunderam, *PVM: Parallel Virtual Machine*, MIT-Press, Cambridge, 1994
- [Grosch 95] J. Grosch, *Toolbox for Compiler Construction*, internal report, GMD - Univ. Karlsruhe, www.gmd.de/SCAI/lab/adaptor/cocktail.html, Sep. 1995
- [Hillis 85] W. D. Hillis, *The Connection Machine*, Ph.D. Thesis, MIT Press, Cambridge, 1985
- [Koelbel et al. 93] C. Koelbel, D. Loveman, R. Schreiber, G. Steele Jr., M. Zosel, *The High Performance Fortran Handbook*, MIT-Press, Cambridge, 1993
- [Kotz 94] D. Kotz, *A data-parallel library for education*, Technical Report PCS-TR94-235, Dept. of Computer Science, Dartmouth College, Nov. 1994
- [Kuehn, Siegel 85] J. T. Kuehn, H. J. Siegel, *Extensions to the C programming language for SIMD/MIMD parallelism*, Proceedings 1985 International Conference on Parallel Processing, IEEE Computer Society, Aug. 1985, pp. 232-235 (4)
- [Li 86] K.-C. Li, *A note on the Vector C Language*, ACM SIGPLAN Notices 21, no. 1, Jan. 1986, pp. 49-57 (9)
- [MasPar 91] MasPar Computer Corporation, *MasPar Programming Language (ANSI C compatible MPL) User Guide*, Software Version 2.2, MasPar System Documentation, DPN 9302-0101, Dec. 1991

- [Metcalf, Reid 90] M. Metcalf, J. Reid, *Fortran 90 Explained*, Oxford University Press, Oxford, 1990
- [Rose, Steele 87] J. R. Rose, G. Steele, *C*: An Extended C Language for Data Parallel Programming*, Thinking Machines Co., Second International Conference on Supercomputing, May 1987, pp. (36)
- [Thinking Machines 89] Thinking Machines Corporation, *Connection Machine Model CM-2 Technical Summary*, version 5.1, Technical Report, May 1989
- [Thinking Machines 90] Thinking Machines Corporation, *C* Programmig Guide Version 6.0*, Technical Report, Nov. 1990
- [West, Grimshaw 94] E. West. Andrew Grimshaw, *Braid: Integrating Task and Data Parallelism*, Computer Science Report, No. CS-94-45, University of Virginia, Charlottesville, Nov. 1994
- [Wirth 83] N. Wirth, *Programming in Modula-2*, Springer-Verlag, Berlin Heidelberg New York, 1983

Author Biography



Thomas Bräunl is Associate Professor at The University of Western Australia, Perth, where he directs the Mobile Robot Lab. He received MS degrees in Computer Science from Kaiserslautern University and the University of Southern California, Los Angeles, in 1986 and 1987, respectively, and a PhD and Habilitation in Computer Science from Stuttgart University in 1989 and 1994, respectively. Professor Bräunl works in the areas of mobile robots, vision, graphics, and concurrency. He has published several textbooks and is creator of the EyeBot mobile robot family.