Geometric Parallelism

D.B. Carpenter

Department of Electronics and Computer Science,
Southampton University,
Southampton, S09 5NH

March 29, 1993

Contents

This document describes a notational scheme for expressing a restricted form of "geometric parallelism". Currently the prime justification for presenting this scheme is as a framework in which to define *occam*-callable communication libraries acting on distributed data.

1 Overview

The notations described in this document are designed to express parallel algorithms dealing with data defined on rectangular grids of one or more dimension. Although many problems require less regular parallel data structures, it is certainly true that an important class of problems *do* have a rectangular grid structure.

1.1 Defining Grids

The underlying model we have is of data structures whose components can be localised at the sites of an abstract rectangular grid. The intention is that the points of this grid will be distributed in some way over a collection of processors, but the exact nature of this distribution is unspecified at this stage.

To define a grid we need a set of "axes". In our notation an axis is formalised by an object called a *distributed index*, or often simply an *index*. In general an index will be implemented by a distributed data structure which might include details of the mapping from the abstract grid to an underlying processor network as well as communication channels between connected processors.

The only aspect of this structure visible in our abstract notation is the total "size" of the index, that is the number of points the grid has in the direction referred to. The declaration of an index takes the form

```
[n] INDEX x :
```

where the count n implies that x refers to a dimension characterised by coordinate values $0 \ldots n-1$. It *doesn't* imply that we are working with an n-dimensional grid in any sense—any similarity to ordinary occam array syntax is largely coincidental. In practice definition of indicies may well be a "configuration" issue, but more dynamic implementations are possible.

Suppose a process has d distributed indicies in scope. The understanding is that we then have a d-dimensional grid available over which to distribute data structures. Some items of data will take different values at every site of the grid, some may be "scalar", that is to say they have a global value. Others may take various values over a subset of dimensions, but have values global with respect to other dimensions.

1.2 Rank and Distributed Data

To capture these possibilities we introduce the idea of *rank*. Rank is somewhat analogous to type, but describes the *distribution* of an object, rather than, for example, its arithmetic properties. Here a rank is just a set of indicies—those over which a data object of such rank is distributed. It's convenient to be able to name a rank in its own right, much as it's convenient to be able to name types. For example

```
RANK XY IS {x, y} :
```

where x and y are existing indicies. A couple of simple operators are useful for combining ranks. Here we will only mention the union operator `\/`. This is used for combining the indicies of two ranks into a single rank, as in

```
{x, y} \/ {z} = {x, y, z}
```

This operation is not quite analogous to set union. In particular it is forbidden to combine two ranks with common indicies. Operators like `\/` are only really provided for convenience in defining the procedure interface—see appendix ?? for more details.

The next concept is the *distribution*, which is our formalisation of the parallel array. For example a distribution of integers over x and y can be declared by

```
{x, y} INT A :
```

or, given the earlier definition of XY, just as

```
XY INT A :
```

In these examples A is similar to a two-dimensional array. It *isn't* a two-dimensional array, however. For example, it cannot be "indexed into" as an array can be. The only way of getting at its elements is to execute a `WHERE` construct.

An object defined with no specified rank, or with the trivial rank, `{}`, is scalar with respect to the currently scoped indicies.

1.3 The WHERE Construct

In general *no* operations can be performed directly on distributions, that is, objects with non-trivial rank. But a construct is provided which "de-scopes" distributed indicies, turning a distribution over that index into a scalar, which *can* be operated on. This construct, `WHERE`, can be regarded as a parallel loop.

For example, the following program is illegal

```
[n] INDEX x :
{x} INT A :
SEQ
  A := A * A
```

because we attempted an operation on a non-scalar object. The correct way to achieve what was probably intended is

```
[n] INDEX x :
{x} INT A :

WHERE {x}
  A := A * A
```

This is legal, because inside the body of the `WHERE` construct A becomes just a scalar (its only index, x, is no longer in force). The effect of the `WHERE` is to perform the process in its body, notionally in parallel, for all values of the "x-coordinate". Because x is "out of scope" inside the body of the construct, the following code-fragments are illegal

```
WHERE {x}                 WHERE {x]
  WHERE {x}                 {x} INT B :
    ... body                ... body
```

As a convenient refinement, we will modify the demand the x is "out of scope"—instead we will just say that x ceases to be a distributed index in the body of `WHERE {x}`. Instead this name now refers to a simple integer value. For each parallel process this value is simply the "x-coordinate" of that process. The above fragments remain illegal because, inside the body of the `WHERE`, x doesn't have the correct type (an index) for the context. But we can now write, for example

```
WHERE {x}
  IF
    (base <= x) AND (x < (base + count))
      ... body
    TRUE
      SKIP
```

in which case the body is only executed for a sub-range of the possible values of the x-coordinate. A more fully developed notation would probably provide a special syntax for this situation. For the present we stick with the simple syntax.

The scheme generalises straightforwardly to the case of multi-dimensional grids. For example

```
[n] INDEX x :
[m] INDEX y :
{x, y} INT A :
{x}    INT b :

WHERE {x}
  SEQ
    b := x
    WHERE {y}
      A := b * y
```

Inside the x loop b is treated as a scalar and x as an integer value, so the assignment to b is legitimate. Notice that the outermost loop has no effect on y, so A looses its x index but remains distributed over y—its type at this stage is `{y} INT`. On entering the y loop A and y also become scalar, and the A assignment is legal.

Because nesting of `WHERE` constructs is so common we introduce the special syntax

```
WHERE {x, y}
```

to represent immediately nested `WHERE {x}` and `WHERE {y}` constructors. More generally the `WHERE` constructor takes an arbitrary rank-valued expression as its argument.

Appendix ?? discusses some aspects of `WHERE` usage not entered into here.

1.4 The Procedure Interface

The final essential ingredient of our scheme is an abstraction mechanism for operations on distributed data—in particular for "communication", which typically takes the form of permutation or other operations on elements of distributions. This is the most complicated part of the scheme to specify in detail, because to be truly useful the procedure interface really needs to be "polymorphic". Thus a given routine can apply to distributions of different types. For example it would be unsatisfactory if a routine for transposing a matrix (a two-dimensional distribution) had to be re-written for every possible arithmetic data type of its elements—`INT`, `REAL32`, `BYTE`, arrays thereof, etc. The full specification of this interface involves rank- and type- variables, and a simple form of pattern-matching. The object is to provide polymorphism but retain the possibility of static type-checking.

Here we will just give a couple of examples of interfaces. The full definition of the mechanism is described in appendix ??.

An example of a possible interface for a procedure designed to perform a "cyclic shift" on a distribution over a specified index is

```
PROC rotate ([] INDEX i, {i} \/ RANK r TYPE t A,
                         VAL {i} \/ r t B, VAL INT shift)
```

The simplest way to explain what this interface means is to give an example of the calling sequence.

```
[n] INDEX x, y :
{x, y} REAL32 data, result :
SEQ
  ...
  rotate (x, result, data, -1)
  ...
```

On calling *rotate*, i will be identified with x, A will be identified with *result* and B will be identified with *data*. To match the actual rank and type of *result* the rank-variable r will be identified with the rank value `{y}` (note that the rank expression `{i} \/ r` then evaluates correctly to `{x, y}`) and the type-variable t will be identified with the type value REAL32. The rank and type of the third formal parameter (B) then match the actual parameter (*data*), so overall rank and type consistency will be accepted. In general a call to *rotate* is valid so long as its first argument is an index, and its second arguments are of the same rank and type, and their rank includes the first argument.

As a second example, the procedure *broadcast* takes data from a particular element of a distribution over some index, and returns it in a datum global with respect to that index.

```
PROC broadcast ([] INDEX i, RANK r TYPE t A,
                          VAL {i} \/ r t B, VAL INT src)
```

can be legally called in the following program

```
[n] INDEX y :
INT result
{y} REAL32 data :
SEQ
  ...
  broadcast (y, result, data, 0)
  ...
```

Identifying the binding of i, r, t, A and B is left as an exercise.

Incidentally, a code fragment like

```
WHERE {y}
  broadcast (y, result, data, 0)
```

is illegal, because inside a y-loop the symbol y refers to a simple integer value, whereas *broadcast* expects an `INDEX` as its first parameter. This is an essential feature of the scheme... without it simple deadlock-free implementations would not be possible.

2 Static Implementation

In this section we outline briefly an implementation scheme for the language described in the previous section. The implementation is static in the sense that we assume definition of the abstract grid (i.e. declaration of indicies) is performed once at some configuration level of the program. More dynamic implementations would clearly be desirable—section ?? contains some comments.

In grid-oriented geometric decompositions, the grid upon which a problem is defined (the *problem grid*) will usually be mapped onto some *process grid*. The process grid is a rectangular grid, typically of the same dimensionality as the problem grid. The process grid is in turn mapped onto some processor array. The two mappings

problem grid $\rightarrow$ *process grid*

process grid $\rightarrow$ *processor array*

will both be many to one in general. The first mapping reflects some "domain decomposition". In general it will not be practical to to provide an individual process for each element of the problem grid for reasons of memory efficiency (every process needs its own workspace and copy of global variables). The second mapping reflects process placement, and again for a variety of efficiency reasons it may be desirable to have several processes on each processor. Note that in general the connectivity of the process grid may be much more general than nearest-neighbour—higher connectivity, if available, facilitates operations such as broadcasts along directions of the process grid.

The abstract grid defined by the distributed indicies of a program can be utilised in more than one way. Most obviously it can be identified with the problem grid, leading to an easy style of programming in which the domain decomposition can be hidden from the programmer. Alternatively it can be identified with the process grid, leaving the programmer to manage the domain decomposition. This leads to a less natural programming style, but to easier implementation of the constructs of the language, and provides the dedicated programmer with more freedom to optimise codes. The latter interpretation will be discussed first. Section ?? provides examples of programming in both styles. Both schemes assume the more restrictive choice of usage rule for `WHERE` loops described in appendix ??.

2.1 Abstract Grid as Process Grid

Implementation of our notations, which we will express in terms of translation to standard *occam* is now almost trivial. Any distribution has only one element in each process (though of course these elements may be sequential arrays—domains indeed).

An index x, say, has associated with it a "distributed data structure". On each process this structure incorporates channels for communication with other processes which can be reached by moving in the x-direction from the current process, as well as the value of the "x-coordinate" of the current process. The exact connectivity depends on what the hardware can support—though for a machine with hardware through-routing capabilities their may as well be channels for direct communication with all processes on the same line of the grid. The x structure will not incorporate channels for communicating with processes on other "lines", i.e., processes that have different coordinates associated with any other axes of the grid.

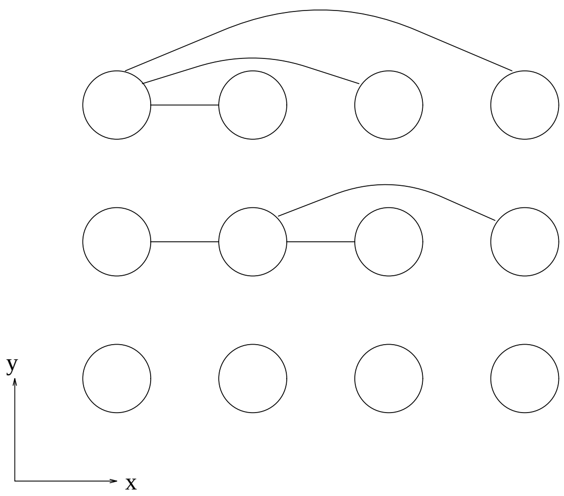

Figure 1: Example channel connections associated with x index

Figure ?? illustrates some of the channels associated with the x-index in a two-dimensional example. Communications procedures such as *broadcast* or *rotate* are passed the index data structure, including such channels, as parameters.

In this case the implementation of `WHERE {x}` simply consists of "hiding" the x data structure by abbreviating the x-coordinate (an element of the structure) as x. As a mundanely concrete example, the program

```
PROC example ([n] INDEX x)

  {x} INT A :
  INT b :
  SEQ
    ...
    broadcast (x, b, A, 0)
    ...
    WHERE {x}
      ...
:
```

could translate to the *occam*

```
PROC example (VAL [] INT x)

  INT A :
  INT b :
  SEQ
    ...
    [] BYTE b RETYPES b :
    [] [SIZE b] BYTE A RETYPES A :
    broadcast (x, b, A, 0)
    ...
    SEQ
      VAL x IS position (x) :
      ...
:
```

The index is implemented here by an array of integers, which will presumably include pointers to channels. This structure is set up at the "configuration level" and accessed ultimately only by predefined procedures such as *broadcast*. Its details needn't be specified here. In the abstract, *broadcast* is type-polymorphic. In a concrete translation such routines would probably be written as procedures operating on `BYTE` arrays, hence the retyping. In the case where each element of the abstract grid has a whole process, no details of the rank need to be passed to procedures such as *broadcast*. Finally, the only effect of `WHERE` is to extract the coordinate value from the index data structure (done here with an access function *position*).

The configuration level of the program declares the channels, spawns the grid of processes, sets up the index structures, and calls the above "user" procedure in each process.

Providing the above translation scheme is performed from a "correct" program it is reasonably straightforward to convince oneself

- Copies of variables global with respect to particular indicies are held redundantly across processes. Consistency of copies is automatically maintained.
- At any stage in the program, *either* an index is in scope and can be passed to a communication routine (such as *broadcast*) *or* processes on a line along that index can be performing independent actions, through a `WHERE`. *Not* both. So mismatched calls to communication procedures cannot cause deadlocks.
- No synchronisation is required for operation of `WHERE` loops (mainly because no memory is shared).

2.2 Abstract Grid as Problem Grid

In this implementation "domaining" is managed below the level of the abstract program specification. This approach is less amenable to manual translation to a language like *occam*. The process grid and associated communication channels are set up in a similar way, but now the ranges of indicies are "sectioned" according to some prescription, and this sectioning induces a natural distribution of arrays over the process grid, generally with many elements of each array held by each process. The index data structure will hold parameters of the sectioning.

`WHERE` loops are translated to sequential loops over the locally held section. The local domain of a distribution is held in an array. But at various stages the "active" data of a distribution may not be a contiguous slice of this array. This is because a `WHERE` construct may localise attention to, say, a particular y slice of an `{x, y}` distribution, leaving a `{x}` distribution. If data is stored with y "most rapidly varying" this `{x}` distribution is not a contiguous region of memory. Thus in general the local representation of a distribution includes information about "strides" in the data array. This information has to be passed to procedures along with the data.

2.3 Less Static Implementation

Why demand that indicies only be declared at configuration level? Imagine code like

```
[n] INDEX x :
SEQ
  ... preamble
  [m] INDEX y :
  SEQ
    ... body
```

Declaration of an index actually causes spawning of processes—the obvious implementation would be something like

```
... declare ‘x’ channels
PAR i = 0 FOR x.num.procs
  INDEX x :
  SEQ
    ... initialise ‘x’ structure
    ... preamble
    ... declare ‘y’ channels
    PAR j = 0 FOR y.num.procs
      INDEX y :
      SEQ
        ... initialise ‘y’ structure
        ... body
```

where `INDEX` is a data-type we don't specify in full detail. Unfortunately this doesn't work, because every process in the second replicated `PAR` needs to have its own copy of *anything* declared before the `[m] INDEX y` declaration. In particular all the x channels need replicating so that each of the now-several x-lines of processes have their own independent x channel sets. Thus simple declaration of an `INDEX` is actually quite complicated to translate unless all indicies are declared together at some outer level, before data declarations start. Implementation is not overwhelmingly difficult, just relatively so.

3 Examples

All but one of the examples given here assume the "hidden domain" implementation outlined in section ??, where the index grid is identified with the problem grid. This allows algorithms to be specified more straightforwardly, but doesn't always permit the most efficient implementation.

```
[n + 1] INDEX x, y :
RANK XY IS {x, y} :
XY REAL32 U :
BOOL converged :
SEQ
  WHERE XY
    IF
      (x = 0) OR (x = n) OR (y = 0) OR (y = n)
        U := boundary (x, y)
      TRUE
        U := 0.0
  parity    := 0
  converged := FALSE
  WHILE NOT converged
    XY BOOL ok :
    XY REAL32 neighb.xn, neighb.xp, neighb.yn, neighb.xp :
    SEQ
      rotate (x, neighb.xn, U,  1)
      rotate (x, neighb.xp, U, -1)
      rotate (y, neighb.yn, U,  1)
      rotate (y, neighb.yp, U, -1)
      WHERE XY
        IF
          (((x + y) REM 2) = parity) AND
              (0 < x) AND (x < n) AND (0 < y) AND (y < n)
            REAL32 new :
            SEQ
              new := 0.25 * ((neighb.xn + neighb.xp)
                             + (neighb.yn + neighb.yp))
              ok  := ABS (U - new) < epsilon
              U   := new
          TRUE
            ok := TRUE
      parity    := (parity + 1) REM 2
      converged := and.2 (ok)
```

Figure 2: Red-black Relaxation for Laplace Equation

3.1 Simple Grid Problems

As a simple prototype of a grid-oriented problem take solution of the Laplace equation by red-black relaxation. A straightforward and fairly detailed implementation is given in figure ??. The ordinary function *boundary* defines the fixed edge values of U. The "predefine" *and*.2 performs logical "and" across the elements of a two-dimensional boolean distribution (a possible definition is given in appendix ??).

One practical issue that arises is the requirement for whole distributions of neighbours. This is unlikely to be a problem for Laplace's equation, where memory requirements are modest. In more memory-intensive applications it could be awkward. There are

```
...
SEQ
  ...
  WHILE NOT converged
    ...
    {x,y} REAL32 neighb.xn, neighb.xp :
    SEQ
      rotate (x, neighb.xn, U,  1)
      rotate (x, neighb.xp, U, -1)
      WHERE {x}
        {y} REAL32 neighb.yn, neighb.xp :
        SEQ
          rotate (y, neighb.yn, U,  1)
          rotate (y, neighb.yp, U, -1)
          WHERE {y}
            ... body of 'WHERE XY' loop
      ...
```

Figure 3: Red-black Relaxation with Reduced Memory

techniques for significantly reducing the amount of memory used by temporary arrays. For example in figure ?? the y neighbours are declared as just `{y}` distributions—normally such arrays require less memory than `{x, y}` distributions.

There is another style of geometric programming which can also be expressed in the current notation. In this case the grid defined by the indicies is regarded as the process grid, and domain decomposition is managed explicitly by the programmer. For a slightly simplified version of the Laplace equation an example is given in figure ??. We ignore here the issue alternating parity and treatment of boundaries. This implementation is clearly more complicated than the first one (almost as complicated as an ordinary *occam* implementation). We have explicitly kept an s by s sequential segment of the array on each of n/s by n/s "processes". Note that in this way the space occupied by the neighbours is much smaller than the data array (if s is moderately large) —we only need four words per processor instead of some arrays the same size as U.

Incidentally the `MOD` operator is a piece of shorthand—it means roughly the same as the *occam* `REM` operator, redefined to return the "correct" value for negative arguments.

3.2 Simple Matrix Operations

Figure ?? implements matrix transpose for a two-dimensional distribution (of `INT`s) in terms of the simpler operation *rotate*. In practise both *in.situ.rotate* and (a polymorphic form of) *transpose.matrix* would probably be provided in predefined libraries, implemented more efficiently.

In a similar vein figure ?? embodies a standard distributed algorithm for multiplying two square matrices. Notice the call to *broadcast*. Remember that at this point in the program x is an index, y is simply an integer value (the "loop counter" for the y loop).

```
[n / s] INDEX x, y :
RANK XY IS {x, y} :
XY [s] [s] REAL32 U :
SEQ
  ... initialisation of 'U'
  WHILE NOT converged
    XY BOOL ok :
    SEQ
      WHERE XY
        ok := TRUE
      SEQ i = 0 FOR s
        SEQ j = 0 FOR s
          XY REAL32 neighb.xn, neighb.xp, neighb.yn, neighb.xp :
          SEQ
            WHERE XY
              SEQ
                neighb.xn := U [(i - 1) MOD s] [j]
                neighb.xp := U [(i + 1) MOD s] [j]
                neighb.yn := U [i] [(j - 1) MOD s]
                neighb.yp := U [i] [(j + 1) MOD s]
            IF
              i = 0
                rotate (x, neighb.xn, U,  1)
              TRUE
                SKIP
            IF
              i = (s - 1)
                rotate (x, neighb.xp, U, -1)
              TRUE
                SKIP
            IF
              j = 0
                rotate (y, neighb.yn, U,  1)
              TRUE
                SKIP
            IF
              j = (s - 1)
                rotate (y, neighb.yp, U, -1)
              TRUE
                SKIP
            WHERE XY
              REAL32 new :
              SEQ
                new := 0.25 * ((neighb.xn + neighb.xp)
                             + (neighb.yn + neighb.yp))
                ok  := ok AND (ABS (U [i] [j] - new) < epsilon)
                U [i] [j] := new
      converged := and.2 (ok)
```

Figure 4: Relaxation with Explicit Domain Decomposition

```
PROC in.situ.rotate ([] INDEX x, {x} \/ RANK r TYPE t A,
                     VAL INT shift)

  -- Rotate input data in 'A', returning result also in 'A'.

  WHERE r
    {x} t temp :
    SEQ
      rotate (x, temp, A, shift)
      WHERE {x}
        temp := shift
:

PROC transpose.matrix ([] INDEX x, y, {x, y} INT result, A)

  -- Set 'result' to transpose of 'A'.
  -- Must have SIZE x = SIZE y.

  SEQ i = 0 FOR SIZE x
    SEQ
      WHERE {x, y}
        IF
          x = y
            result := A
          TRUE
            SKIP
      in.situ.rotate (x, result, 1)
      in.situ.rotate (y, A     , 1)
:
```

Figure 5: Matrix Transpose on a Grid

```
[n] INDEX x, y :
{x, y} INT A, B, result :
SEQ
  ... initialisation of 'A', 'B'
  WHERE {x, y}
    result := 0
  SEQ i = 0 FOR n
    SEQ
      WHERE {y}
        INT diag :
        SEQ
          diag := broadcast (x, A, y)
          WHERE {x}
            result := result + (diag * B)
      in.situ.rotate (x, A, -1)
      in.situ.rotate (y, B, -1)
```

Figure 6: Matrix Multiplication on a Grid

3.3 LU Decomposition

The example given here for LU or Cholesky decomposition is based on the recurrence

$$L = \begin{pmatrix} l_{11} & 0 \\ l_C & L' \end{pmatrix},\ U = \begin{pmatrix} 1 & u_R \\ 0 & U' \end{pmatrix},\ A = \begin{pmatrix} a_{11} & a_R \\ a_C & A' \end{pmatrix}$$

with $LU = A$, i.e.

$$l_{11} = a_{11},\ l_C = a_C,\ u_R = l_{11}^{-1} a_R$$

and

$$L'U' = A' - l_C u_R$$

If A, L and U are $n \times n$ matrices, A', L' and U' are $(n-1) \times (n-1)$ matrices, so n applications of this operation reduce A to LU form. This isn't quite the way LU decomposition is presented in most textbooks, but it suffices for present purposes.

Figure **??** gives a "column-oriented" distribution. The function *locate.largest* is an ordinary sequential routine for locating the largest element of an array. Note that the broadcast of n numbers in *reduction* takes time dominated by a term proportional to n. The arithmetic in one reduction stage also takes parallel time proportional to n—in fact considerably longer if there are several columns per process. Thus the communications overhead is at most a constant factor.

The algorithm is not particularly well load-balanced if the distribution of data is the most direct one, because at any point one is only operating on a slice of the whole array. Again this leads to a constant inefficiency, one that is often alleviated by "wrap-around" mapping of data to processors. The specification here is at too high a level to express this mapping, although it is possible that the implementation might adopt it.

4 Discussion

The notation described so far was intentionally minimal. Some minor enhancements suggest themselves. For example it would be convenient to introduce "slices" of indicies so that clumsy constructions like

```
WHERE {x}
  IF
    (base <= x) AND (x < (base + count))
      ... body
    TRUE
      SKIP
```

could be replace with something along the lines

```
WHERE {[x FROM base FOR count]}
  ... body
```

Doing this in a fully consistent way would probably require that ranks be able to include index slices as well as "full indicies". This might complicate the procedure interface, if distributions over such ranks had to passed as parameters.

```
INT FUNCTION find.pivot ({[n] INDEX x} [n] REAL32 mat, VAL INT col)

  -- Calc pivot for column 'col', and broadcast it.

  INT pivot :
  VALOF
    {x} INT temp :
    SEQ
      WHERE {x}
        IF
          x = col
            temp := col + locate.largest ([mat FROM col FOR n - col])
          TRUE
            SKIP
      broadcast (x, pivot, temp, col)
    RESULT pivot
:

PROC reduction ({[n] INDEX x} [n] REAL32 mat, VAL INT stage)

  INT pivot :
  [n] REAL32 first.col :
  SEQ
    pivot := find.pivot (mat, stage)
    WHERE {x}
      mat [stage], mat [pivot] := mat [pivot], mat [stage]
    broadcast (x, first.col, mat, stage)
    WHERE {x}
      IF
        x > stage
          SEQ
            mat [stage] := mat [stage] / first.col [stage]
            SEQ i = stage + 1 FOR n - (stage + 1)
              mat [i] := mat [i] - (first.col [i] * mat [stage])
        TRUE
          SKIP
:

[n] INDEX x :
{x} [n] REAL32 A :

SEQ r = 0 FOR n
  reduction (A, r)
```

Figure 7: Column-oriented LU Decomposition

A more interesting question is whether any of ideas discussed here could be incorporated into a more dynamic scheme, which would allow more general (non-rectangular) "grids" to be defined at run-time. In our execution model processes are spawned whenever an index is declared. In an analogous dynamic framework it might be possible to bring an index into existence by "allocating" it rather than declaring it. This would dynamically spawn processes (which might later be killed by de-allocating the index). Moreover, through the `WHERE` construct, these dynamic indicies could be allocated at a localised point in an existing grid, permitting some form of dynamic grid refinement—or, less ambitiously, simply the creation of non-rectangular grids. Certainly such a scheme requires modifications to the simple ideas of rank and distribution described here—a much more dynamic rank-like object would be needed, which would itself have aspects of a distributed array, in the sense that it would contain different (local) indicies at different sites.

A Where Usage

The `WHERE` construct can be regarded as a form of parellel loop. As in *occam*, parallel constructs need some usage rules associated with them. We have not discussed the possibility of channel communication inside `WHERE` constructs, so for the purposes of this document we don't need a usage rule for channels. This leaves treatment of variables. Fairly clearly, the variable usage rules only need restrict use of "global" variables—that is to say, inside a `WHERE {x}` construct, we only need to put restrictions on variables *not* distributed over x. There is no danger of parallel access to variables which *are* x distributions.

There are two plausible choices of usage restriction on global variables. The most "natural" is to treat `WHERE` as if it were a `PAR`, the restriction being that only one component of the loop may write to a particular global variable, and if any component does so, no component may read that variable. An advantage of this scheme is that it provides an intrinsic mechanism for copying an element of a distribution to a global variable. For example

```
{x} INT A :
INT b :

WHERE {x}
  IF
    x = 23
      b := A
    TRUE
      SKIP
```

copies the 23rd element of A to b.

One disadvantage of this regime is that in the static implementation the assignment `b = A` has to be implemented implicitly as a broadcast. This implementation is difficult, because the sites (if any) from which broadcasts are to proceed are not known globally.

Such considerations suggest a second regime in which assignments to global variables are simply forbidden inside `WHERE` constructs. Any broadcasts require explicit calls to a communication routine. (Incidentally, a minor advantage of this more restrictive scheme is that usage-checking is now "automatic"—essentially no more difficult than type-checking.)

At a purist level the choice between the two regimes would presumably favour the former—the advantage is in having the possibility of broadcasts instrinsic to the language, especially since any communication pattern then *can* be built (probably very inefficiently) from broadcasts, without recourse to externally defined communication procedures. From an implementation point of view the second regime is more attractive.

B The Procedure Interface: Pattern-matching.

Following the *occam* model, procedure call semantics are defined in terms of abbreviation and textual substitution. Informally speaking (with some provisos about name-clashes), the code-fragment

```
PROC eg ([] INDEX x, {x} INT a)

  ...  body of 'eg'
:

[n] INDEX y :
{y} INT b :
SEQ
  ...  preamble
  eg (y, b)
```

is "semantically equivalent" to the fragment

```
[n] INDEX y :
{y} INT b :
SEQ
  ...  preamble

  [] INDEX x IS y :
  {x} INT  a IS b :
  ...  body of 'eg'
```

With this understanding it suffices to give a meaning to type and rank matching in abbreviations, in order to define their behaviour in the procedure interface.

Three generic forms of abbreviation are

```
abbreviation  ::=  name IS expression :

abbreviation  ::=  type_pattern name IS expression :

abbreviation  ::=  rank_pattern type_pattern name IS expression :
```

(`VAL` forms would also be available). In the first form the type (and rank, if there is a non-trivial one) of *name* are inherited from *expression* (this is a normal *occam* convention). In the second form rank is inherited from *expression*—*type_pattern* must match the type of *expression*. Section ?? describes treatment of this form. Section ?? describes treatment of the third form. First we introduce a couple of operators involving types and ranks.

B.1 Some Type and Rank Operators

The operator **TYPEOF** applied to an expression returns to type of an operator. Similarly the operator **RANKOF** returns an expression's rank.

Two primitive "arithmetic" operators on ranks are defined. The "union" operator `\/` can be used to combine several ranks, returning the rank containing all the indicies from the component ranks.

```
{x, y} \/ {z} = {x, y, z}
```

Note however that it is erroneous to try and combine ranks with common indicies using `\/`. The union operator is commutative.

The "subtraction" operator `-` can be used to combine two ranks, returning the rank containing those indicies in its left-hand operand which are not in its right-hand operand.

```
{x, y, z} - {z} = {x, y}
```

It is erroneous to try to combine ranks for which the right-hand operand is not a subset of the left-hand operand using `-`. Below we take the subtraction operator to have lower precedence than the union operator.

B.2 Type Matching

The syntax for a type pattern is simply

```
type_pattern   ::=  type_constant | type_variable

type_variable  ::=  TYPE name
```

and *type_constant* is either the name of a previously defined type, or a literal type (such as `[3] INT`).

Type patterns are reduced rather straightforwardly. In the case where the pattern is a type-constant, no reduction is necessary. In the case where it is a type-variable, the reduction is

```
TYPE name_1 name_2 IS expression :
```

is equivalent to

```
TYPE name_1 IS TYPEOF expression :
name_2      IS expression :
```

We can now give a full (if fairly trivial) example of a type-polymorphic procedure:

```
PROC copy (TYPE t out, VAL t in)

  out := in
:

REAL32 a, b :
SEQ
  ...
  copy (a, b)
```

is first reduced to

```
REAL32 a, b :
SEQ
  ...
  TYPE t out IS a :
  VAL  t  in IS b :
  out := in
```

by the procedure expansion rules. The second abbreviation needs no reduction. The first is reduced according to the above rules to

```
REAL32 a, b :
SEQ
  ...
  TYPE t   IS TYPEOF a :
  out      IS a :
  VAL t in IS b :
  out := in
```

In other words, the type variable t is bound to the value `REAL32`, and the types of the a and b abbreviations match correctly.

B.3 Rank Matching

The syntax of a rank pattern is more elaborate. Introduce

```
index_constant ::=  name

index_variable ::=  [] INDEX name | [expression] INDEX name

rank_element   ::=  rank_constant | rank_variable

rank_constant  ::=  name | { index_constant_list }

rank_variable  ::=  RANK name | { index_variable_list }
```

where *index_constant_list* and *index_variable_list* are comma-separated lists. The syntax of a rank pattern is then

```
rank_pattern   ::=  rank_element | rank_element \/ rank_pattern
```

In other words, a rank pattern is a union of rank constants and rank variables. To disambiguate pattern-matching, we assume that there is at most one *rank variable* in a particular rank pattern. For example

```
r \/ s \/ {[] INDEX x, [] INDEX y}

{x} \/ RANK r
```

are legal rank patterns, with the last term of the union being the rank variable in both cases. But

```
RANK s \/ {[] INDEX x}
```

is illegal, because it includes two rank variables.

In the case where the rank pattern has no variable part, the abbreviation reduces immediately to the case of the previous section

```
rank_pattern type_pattern name IS expression :
```

is equivalent to

```
type_pattern name IS expression :
```

provided the constant *rank_pattern* matches the rank of *expression*. Otherwise a rank-checking error has occured. In the case where the rank pattern includes a variable part and a constant part, denote the variable part by *var* and the constant part by *const*, and the reduction is to

```
var               IS RANKOF expression - const :
type_pattern name IS expression :
```

Since the latter manipulation is slightly intricate we give two examples. In

```
r \/ s \/ {[] INDEX x, [] INDEX y} INT a IS expr :
```

var is identified as `{[] INDEX x, [] INDEX y}` and *const* as `r \/ s`, with the resulting reduction

```
{[] INDEX x, [] INDEX y} IS RANKOF expr - r \/ s :
INT a IS expr :
```

and in

```
{x} \/ RANK r TYPE t a IS expr :
```

var is identified as `RANK r` and *const* as `{x}`, leading to

```
RANK r   IS RANKOF expr - {x} :
TYPE t a IS expr :
```

Finally we have to give a meaning to abbreviations of the form

```
{ index_variable_list } IS rank_expression :
```

such as that appearing in the first example above. If the number of index variables in the list doesn't correspond to the size of the rank expression on the left hand side, a rank-checking error occurs. Otherwise, if there is only one index variable in the list the meaning is fairly clear. If there are several, an ambiguity exists. To give a meaning to this case we assume that all indicies in existence have a precedence defined amongst them. For example the index defined first—with the largest scope—could be the highest precedence, and so on. This gives a natural ordering to the indicies in the *rank_expression* set. After evaluating this expression and ordering it thus, the substitution takes the form

```
{[e_1] INDEX v_1, ..., [e_n] INDEX v_n} IS {c_1, ..., c_n} :
```

is equivalent to

```
[e_1] INDEX v_1 IS c_1 :
...
[e_n] INDEX v_n IS c_n :
```

C Some "Communication" Primitives

Here we give the definitions of some of the routines used as "primitives" in the examples. The specification given for *broadcast*, in terms of which the other routines are defined, is unlikely to be acceptable under realistic usage regimes—instead it could be taken as a true primitive. In all cases (except, possibly *and*.2) the implementations given here would be extremely inefficient. Most of them are essentially sequential. The idea is just to specify the behaviour of the routines.

A useful communication library would need many more such routines.

```
PROC broadcast ([] INDEX x, RANK R TYPE T A, VAL {x} \/ R T B,
               VAL INT src)

  -- Copy elements of ‘B’ from x = src to variable ‘A’, which
  -- is global with respect to ‘x’.

  -- Notice that this program is illegal under the (most practical)
  -- strict form of WHERE usage rules.

  IF
    (0 <= src) AND (src < (SIZE x))
      WHERE {x}
        IF
          x = src
            WHERE R
              A := B    -- assignment to global variable!
          TRUE
            SKIP
    TRUE
      STOP
:
```

```
PROC rotate ([] INDEX x, {x} \/ RANK R TYPE T A, VAL {x} \/ R T B,
            VAL INT shift)

  -- Shift values in ‘B’ in ‘x’ direction by ‘shift’ sites,
  -- returning result in ‘A’

  INT n :
  SEQ
    n := SIZE x
    SEQ i = 0 FOR n
      R T element :
      SEQ
        broadcast (x, element, B, i)
        WHERE {x}
          IF
            x = ((i + shift) MOD n)
              WHERE R
                A := element
            TRUE
              SKIP
:

BOOL FUNCTION and ({[] INDEX x} BOOL A)

  -- Logical "and" of 1-d Boolean distribution.

  BOOL result :
  VALOF
    SEQ
      result := TRUE
      SEQ i = 0 FOR SIZE x
        BOOL element :
        SEQ
          broadcast (x, element, A, i)
          result := result AND element
    RESULT result
:
```

```
BOOL FUNCTION and.2 ({[] INDEX x, [] INDEX y} BOOL A)

  -- Logical "and" of 2-d Boolean distribution.

  BOOL result :
  VALOF
    {x} BOOL temp :
    SEQ
      WHERE {x}
        temp := and (A)
      result := and (temp)
    RESULT result
:
```