1 Translation of "Level-I" DAP occam to occam

By "Level I DAP occam" I mean the implementation where the abstract grid defined by the indicies of the problem is also the "process grid", ie, there is one process for every site of the grid. Also I am assuming the index declaration is done at the configuration level.

As mentioned in the document "geometric parallelism", apart from the configuration level definition of indicies and instantiation of the parallel processes, the only non-trivial part of translation of a correct program is in translation of the procedure interface.

1.1 Translating Procedure Definitions and Calls

A valid procedure header contains a list of type, rank and index variables and constants as well as data "variables" (in this context any name preceded by a specifier is regarded as a variable, including for example VAL data parameters). In general these variables and constants will be combined by some "rank expression" operators. These operators, along with type, rank and index *constants* are only relevant for generalised type-checking purposes. The only terms which need to appear in the parameter lists in the occam translation are type, rank, index and data *variables*. Therefore, the first stage in translation is to remove from the formal parameter list all constants and operators of the aforementioned type to leave just a linear list of variable parameters. Examples:

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

becomes

```
PROC broadcast (INDEX x, RANK R, TYPE T, VAR A, VAL B, VAL INT src)
```

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

becomes

```
PROC rotate (INDEX x, RANK R, TYPE T, VAR A, VAL B, VAL INT shift)
```

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

becomes

```
BOOL FUNCTION and.2 (INDEX x, INDEX y, VAL BOOL A)
```

For the time being, in the case of data variables with non-static type, we go back to the old *occam I* notation of simple untyped `VAL` or `VAR` parameters. Data variables that *are* literally typed, such as the `BOOL` in the last example, are already in their final translated form. So we need translations of index, rank, type and untyped `VAL/VAR` data parameters.

First we deal with untyped data. Such parameters will be passed as unsized `BYTE` arrays in *occam* translation. In general this means that a redundant SIZE value will be implicitly passed, but if translation is to *occam* there doesn't seem to be much alternative.

Secondly consider type variables. These may be passed in a vestigial form as `VAL INT` parameters defining the size in bytes of the type. For the kind of operations that any realistic implementation would permit on variable-typed data no further information is likely to be needed. Notice that this information is fairly redundant in an occam translation, because lengths of unsized arrays are passed automatically, as mentioned in the previous paragraph—an alternative would be to simply omit type variable parameters in the translation.

Thirdly consider **INDEX** variables. An index will be represented in the *occam* translation as some data structure, probably incorporating pointers to communication channels—notice that *occam* doesn't actually support such structures, and that the lowest level access routines to indices will have to be written in, for example, assembly language (or even C). For the moment we will just refer to this structure as an imaginary *occam* data structure called an **INDEX**, which might, for example, be represented as an integer array in the concrete translation.

Finally we consider rank variables. It may be sufficient to simply omit these parameters, depending on how general the implementation of the language is to be. It is fairly clear that some programs that would be legal in the abstract definition of the procedure interface will not actually be implementable efficiently, because their polymorphism is "too great" to admit static type- and rank- checking. The final rule in appendix B of [?] which gives a meaning to abreviations of the form

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

is a case in point. Although this rule is useful in defining the procedure interface it would probably be fairly difficult to statically rank-check a program which contained such abbreviations. This abbreviation is the only mechanism for explicitly unbundling indicies from rank variables. If we simply forbid such abbreviations in real programs there ceases to be any requirement for passing **RANK** variables (as opposed to individual **INDEX** variables) in the parameter list of a procedure in the current (level I) implementation. A *level II* implementation requires passing at least a vestigial form of such rank variables, to permit traversal of data distributed over the rank. But in the level I implementation traversal of ranks is trivial (since each process holds at most one element of a distribution).

2 Translation of "Level-II" DAP occam to occam

After thinking about it for a day I've ended up with the following scenario...

An index structure contains two fields—the process field which is identical to the index structure of the level I implementation; and the access field which includes a small integer which uniquely identifies the index (a sort of descriptor), along with information about sectioning of the index across processes.

A distribution has associated with it an access structure which includes a short array (indexed by an index descriptor) containing information about array strides. In a typical implementation the sizing of this array places a limit on the number of indicies that can be "active" at any point in the program (the definition of "active" here needs sorting out). Naturally the access structures have to be passed to procedures along with the distribution.

Finally, a rank is passed to a procedure simply as a table of pointers to index structures.

2.1 Ranges

How is this scenario modified by the "new" ideas about ranges? A range can be treated, for example, as pair of integers and a pointer to an index. To what extent, though, does a range need to be implemented as a different type of structure from a full index? For example ranges and full indicies can be freely mixed in ranks, so it would make life awkward if the two had different representations. Is it viable to assume that an index "is a" range? In other words is

```
RANGE OF x r IS x :
```

a legal abbreviation? I can't really see why it would cause any problem. It "explains" why a rank can contain indicies, assuming that the definition of a rank is as a set of ranges.

Perhaps this "is a" relation only holds for *sized* indicies. I also have to bear in mind the option for declaring unsized indicies. It's not terribly clear how a procedure which is passed an index can distinguish between a sized and an unsized index. Certainly the current syntax doesn't provide a means of conveying this information in a procedure header. That's a bit annoying really.

Suppose I start using unsized indicies. Then I have to change all my procedure specifications to things like

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

I suppose that's reasonable. To use these things I need a couple of operators for extracting the base and count from a range, say `BASE` and `SIZE`.

The main problem is whether I have to "recompile" a procedure like that when I go from a sized to an unsized index x. Clearly a procedure which was written to exploit the size of an index won't work if passed an unsized index, but would a procedure like the above one, which didn't exploit an index's size, have to be written differently in the sized and unsized cases? I think the answer must be an emphatic *no*. In other words all procedures should be able to cope with the sized index data structure, even if some of the fields in it are undefined in the unsized case. The alternative is to abandon the concept of a sized index, which was one of my earliest thoughts when I was thinking about ranges.

How about this...

An index is actually handled as two entities. One is the process part, similar or identical to the index structure used in implementing level I. The second is a range structure. The latter part is always carried about with an index and passed to procedures, but in some cases it may contain a nonsense range which implies that in fact the index was declared unsized.

Range structures contain information about what part of a range is held on the current processor, and an index descriptor.

Ranks are now tables of (pointers to?) *range structures*, *not* index structures (in fact there is no longer any atomic concept of an index structure—an index is represented in two parts).

2.2 Array access

Now I have a tentative idea about what data structures I'm going to need, I should think more concretely about what actually has to go *in* each of these structures to implement the basic elements of the language—the communications library and the `WHERE` construct.

Although the language itself assumes that all distributions over a particular index will be "co-distributed", in the sense that an element corresponding to the same value of an index lives in the same process, it may be possible to implement the range structures such that the communications libraries don't make this assumption—at least this is a target to aim at. Of course `WHERE` doesn't make much sense without this assumption, but then it would be nice to keep the communications library implementation as independent of the detailed language context as practical.

2.2.1 Requirements on Implementation of Ranges

By implementation of "ranges" I mean more generally how a range of index values is mapped over the process structure. Two extreme examples would be the "direct" and "wraparound" strategies, where either each process holds a single *contiguous* section of the whole range with successive sections on successive processes, or successive elements of the range are held on succesive processes, cycling through processes if there are more points in the range than processes. Important properties for any practical scheme are that

- There should be a simple globally applicable algorithm for determining on which process a given point in a range is located (for implementing point-to-point and more general communication operations).
- There should be a simple algorithm for traversing all points in a range held on the local process (for implementing `WHERE` constructs).
- There should be a compact (distributed) structure for specifying a range.
- Given such a structure, there should be a simple algorithm for constructing another range structure corresponding to a co-distributed sub-range.

Additionally any strategy must admit reasonable "load-balancing"—for example the general strategy which assigns all elements of all ranges to process 0 (satisfying all the above requirements admirably) fails notably on this last account. This doesn't mean that *all* possible ranges have to be load-balanced in this sense—some co-distributed sub-ranges of well balanced ranges will obviously not themselves be well balanced. But the general strategy must allow easy ways of selecting load-balanced distributions for "parent" ranges.

2.3 Implementation of Ranges

There are two obvious mappings which satisfy the above requirements—the "direct" and "wraparound" mappings just mentioned. It would also be possible to have a hybrid of the two. I'm not sure whether it is better to work things through with the hybrid form first, and specialise if necessary to those two special cases at the end, or the work with the two special cases, then general to the hybrid case at the end if it seems practical. I think in fact I'll go through the two special cases first, *then* go through the hybrid case. There's something to be said for having a number of distinct implementations—you start to see the more profound abstractions as the ones that are common to implementations.

2.3.1 Direct Mappings

Assume we have a process array of size p and a (parent) range of size n. The distribution is has $w = [n - 1/p] + 1$ sites on the first $p - 1$ processes, and the remainder on the last process (note: either $n = 1$ is treated as a special case, or the operation $[x/y]$ is carefully defined). The formula for finding the process i holding site x is just $i = x/w$, and for such a range enumerating the sites held on a given process is straightforward (we will say more later). The requirement of being able to construct sub-ranges leads to a generalisation of this kind of distribution.

Before addressing that I need to think more about how a subrange is indexed. In my program model the index associated with a sub-range is numerically equal to the index of the same site in the parent range. Is this the best way to implement things, or would it be better to have a site in a sub-range addressed by its position relative to the start of the sub-range? Assuming that indeed all parent distributions *do* have the form noted above, formulae will be simplified if I stick to the original idea. The base and size of a (sub-)range are supposed to be globally available anyway, so it's an easy enough business to go over to the relative addressing mode if necessary. The only annoying thing is that it would somehow be more elegant if a sub-range could be handled independently of its parent range, rather than have this relic of its ancestory carried around with it. Also it would be a nice (if probably pointless) generalisation not to enforce parent distributions to have the special form implied. The fact is though that it's fairly fundamental to my model that the "value" of an index is a unique specifier for "where" it is. Two subranges derived from the same parent still need to retain some knowledge about which of their common elements are in the same place, and this is embodied in the value of the index. So I think that really there isn't much justification to go over to the "nicer" form.

Having said that...I suspect that really I should work things through with both assumptions, and see how they look in the end.

Just to clarify my thoughts, I'm going to have a short aside on the relationship between ranges and "local access structures". A local access structure is a distributed data structure uniquely tied to an individual distribution (parallel array). It contains a pointer to the first (currently active—given the `WHERE` context) element of the distribution in local memory, and a set of strides used to find any individual element of the multi-dimensional array. Notice that two distributions of the same rank, say, are allowed to have completely different local access structures, but they will share the range structures (referred to in their common rank structure). Achieving this separation of concerns cleanly places some restrictions on the possible model we can adopt for distributing the elements of a range. If there is to be a constant stride associated with traversing in some direction all local elements of a distribution, taking a subrange must not break contiguity of locally held sites

I really need to know precisely what operations have to be performed on range structures to implement high-level features. For implementing communications the only special thing that's likely to be needed is: given an index value determine on which process (plane) and at what local index value the corresponding data resides. I think all other operations can in essence be expressed in terms of the set of operations needed to construct general `WHERE` loops. So maybe I should just take a typical reasonably hard `WHERE` loop and translate it.

```
PROC copy.into (RANK R TYPE T A, VAL RANK S T B)

  WHERE R
    A := B
:
```

This is certainly a fairly hard thing to translate. However it isn't statically rank checkable. It's valid under various assumptions about R and S but, as I say, it wouldn't get through simple rank-checking, so it probably couldn't be compiled. Nevertheless I should be able to manually translate it I suppose—say with dynamic rank-checking.

```
PROC copy.into (VAL RANK R, VAL ACCESS A.access,     [] [] BYTE A.data,
                    VAL RANK S, VAL ACCESS B.access, VAL [] [] BYTE B.data)

      WHERE R
        A := B
    :
```

The "inputs" to the most general version of a `WHERE` loop are

- A set of access structures for all distributions referred to in the body of the loop.
- A set of rank structures covering all distributions referred to in the body of the loop.
- The rank structure over which the loop runs.

The structures used in the body of the loop are

- A set of access structures with modified base pointers associated with elements of all distributions referred to in the body of the loop—in particular where these distributions have been reduced to scalar these base pointers are just the address of the single element that can be accessed.
- A set of modified rank structures covering all distributions referred to in the body of the loop.
- A set of coordinate values for all indicies contained in the loop-defining rank and referred to in the body of the loop.

I think maybe I should just do a pseudo-implementation of a big procedure which just goes in a `WHILE` loop (a sort of "next" operator), and generates all those outputs given the inputs, and says when the loop should terminate...something like

```
... declare loop body structures
LOOP.STATE loop state :
BOOL another :
SEQ
  init.loop (another, loop body structures, loop state,
             input access and rank structures)
  WHILE another
    SEQ
      ... body of loop (utilise loop body structures)
      next.loop (another, loop body structures, loop state,
                 input access and rank structures)
```

More concretely, the interface of *next.loop* (for example) is

```
  init.loop (another, loop body structures, loop state,
             input access and rank structures)
  PROC next.loop (BOOL another,
                  RANK out.loop.rank, VAL [] RANK *out.ranks,
        [] ACCESS.BASE out.bases, [] INT out.coords,
            LOOP.STATE loop.state,
              VAL RANK in.loop.rank,  VAL [] RANK *in.ranks,
    VAL [] ACCESS.BASE in.bases, VAL [] ACCESS.STRIDES strides)
```

I've used a mish-mash of *occam* and *C* conventions here—the latter to indicate passing of pointers.

Immediate observation: the computation of "out.rank" parameters is loop invariant—ranks effective inside the body of the loop are the same for every instance of the loop body. Their computation can be moved into *init.loop*. At this stage I haven't specified what is contained in the `LOOP.STATE` structure. It will certainly involve one or more loop counters, and probably assorted other state required to optimise computation of bases and coordinates. In fact it may contain much of the information in the last four parameters, unbundled by *init.loop* for easier computation of the outputs from *next.loop*.

In fact, let me work on the last assumption, defining *next.loop* entirely in terms of a set of data structures embodied in `LOOP.STATE` specifically chosen to simplify and/or optimise the performance of this procedure, without reference to the last four parameters in the interface above. One of the jobs of *init.loop* is then to set up these structures, extracting the appropriate information from the general rank and access structures. The concrete implementation of the latter structures can then be chosen to simplify the task of *init.loop*.

I'll start from a simple fragment for traversing an arbitrary dimensional index space.

```
{(SIZE sizes >= 1) /\ (ANY i: sizes [i] > 0)}

VAL dimension IS (SIZE sizes) - 1 :

INT counters.ptr :
[max.indicies + 1] INT counters :
SEQ
  SEQ i = 0 FOR dimension + 1
    counters [i] := 0
  counters.ptr := 0
  WHILE counters.ptr < dimension
    SEQ
      access (counters)
      SEQ
        counters.ptr := 0
        counters [0] := counters [0] + 1
        WHILE counters [counters.ptr] = sizes [counters.ptr]
          SEQ
            counters [counters.ptr] := 0
            counters.ptr            := counters.ptr + 1
            counters [counters.ptr] := counters [counters.ptr] + 1
```

(the size of *sizes* is $dimension + 1$ to allow a sentinel element which eliminates a test in the inner loop condition). All work is done in the routine *access*, the *next* operation corresponds to the remainder of the outer loop body. For our purposes *next* should be more complicated—it has to include the address computation for all arrays used in the equivalent to *access*, rather than just passing it an index set. For this purpose it will need access to all the stride arrays, and to an "indirection" table which maps the dimensions traversed by the above loop to *index descriptors* for accessing the stride arrays.

```
[dimension + 1] INT sizes :
[dimension]     INT index.descs :

[num.arrays] INT access.base :
[num.arrays] [max.indicies] INT access.stride :

INT counters.ptr :
[max.indicies + 1] INT counters :
SEQ
  SEQ i = 0 FOR dimension + 1
    counters [i] := 0
  counters.ptr := 0
  WHILE counters.ptr < dimension
    SEQ
      access (access.base)
      SEQ
        counters.ptr := 0
        counters [0] := counters [0] + 1
        SEQ i = 0 FOR num.arrays
          access.base [i] := access.base [i] +
            access.stride [i] [index.desc [0]]
        WHILE counters [counters.ptr] = sizes [counters.ptr]
          SEQ
            counters [counters.ptr] := 0
            SEQ i = 0 FOR num.arrays
              access.base [i] := access.base [i] - (sizes [counters.ptr] *
                access.stride [i] [index.desc [counters.pointer]])
            counters.ptr            := counters.ptr + 1
            counters [counters.ptr] := counters [counters.ptr] + 1
            SEQ i = 0 FOR num.arrays
              access.base [i] := access.base [i] +
                access.stride [i] [index.desc [counters.pointer]]
```

The loops of the form `SEQ i = 0 FOR num.arrays` only arise because *num.array* is variable—this is only needed because we want to embody the *next* operation in a general procedure. If the code was included inline, as it probably would be normally for implementing a loop, these loops and the associated indexing by i would not be overheads. The other major overhead is in the assignments to *access.base*. It would be *much* more efficient to precompute an array of "steps" for each array:

```
SEQ i = 0 FOR num.arrays
  SEQ counters.pointer = 0 FOR dimension + 1
    SEQ
      IF
        counters.pointer < dimension
          steps [i] [counters.pointer] :=
            access.stride [i] [index.desc [counters.pointer]]
        TRUE
          SKIP
      IF
        counters.pointer > 0
          steps [i] [counters.pointer] := steps [i] [counters.pointer] -
               (sizes [counters.ptr - 1] *
                access.stride [i] [index.desc [counters.pointer - 1]])
        TRUE
          SKIP
```

The code then simplifies to

```
[dimension + 1] INT sizes :

[num.arrays] INT access.base :
[num.arrays] [max.indicies] INT steps :

INT counters.ptr :
[max.indicies + 1] INT counters :
SEQ
  SEQ i = 0 FOR dimension + 1
    counters [i] := 0
  counters.ptr := 0
  WHILE counters.ptr < dimension
    SEQ
      access (access.base)
      SEQ
        counters.ptr := 0
        counters [0] := counters [0] + 1
        SEQ i = 0 FOR num.arrays
          access.base [i] := access.base [i] + steps [i] [0]
        WHILE counters [counters.ptr] = sizes [counters.ptr]
          SEQ
            counters [counters.ptr] := 0
            counters.ptr            := counters.ptr + 1
            counters [counters.ptr] := counters [counters.ptr] + 1
            SEQ i = 0 FOR num.arrays
              access.base [i] := access.base [i] + steps [i] [counters.ptr]
```

The next issue that needs addressing is computation of index "coordinates". Although I haven't yet spelled out my assumptions about mapping of ranges over process arrays (by the way, they are actually somewhat more general than supposed at the beginning of this section—that was written some days ago) it's likely that the formulae for them in terms of the local indicies (embodied in *counters* above) will be like

$$x = c + i_x + J[(i_x + b)/W]. \tag{1}$$

This is function of i_x which increases with unit slope but jumps by an extra J at intervals of W. (This scheme will encompass a convenient generalisation of both direct and "wraparound" mappings: for direct mapping W is very large, so the "jump" never happens; for wraparound mapping it is 1, so there is a jump every time i_x is incremented).

Either this function (which is fairly complicated) can be calculated directly in each iteration, or it can be calculated incrementally (something like...)

```
INT x := c + (J * (b / W)) :
INT r := (W + (b - 1)) REM W :

SEQ i = 0 FOR size
  SEQ
    ... body
    x := x + 1
    r := r - 1
    IF
      r = 0
        SEQ
          x := x + J
          r := W
      TRUE
        SKIP
```

It's far from obvious *a priori* which method will be fastest. A completely different approach would be to precompute a *distribution* of the coordinate values, which was embodied in the index structure, then use the mechanism for accessing distribution elements to access the index values as well! This is radically inefficient in space terms, but it's horribly tempting from the point of view of unifying two different aspects of the implementation. Actually, when I say it's radically inefficient, I mean that it's a "small" constant inefficiency. The worst case would be where only one index was defined, and only one distribution over this index introduced, when it could be a factor two. I don't know. As I say, it's tempting. It might be well worth going for it in a first-cut implementation. I need to think about it.

After thinking on it a bit, I think it's worth bearing this implementation in mind, and using it as an excuse for postponing working out the best alternative implementation. But ultimately there will have to be an alternative implementation, for example, to cope with "unsized" indicies for which I can't precompute distributions of coordinate values.

Anyway, putting that on one side, the next thing I need to think about is how *init.loop* is implemented. I now know (ignoring coordinate values) the interfaces more concretely should be along the lines

```
PROC where.init (BOOL out.another,
              [] INT out.bases,
                 INT out.counters.ptr, [] INT out.counters,
           [] [] INT out.steps,
                 INT out.dimension,    [] INT out.sizes,
        VAL [] RANK * out.ranks,
             VAL RANK in.loop.rank,
        VAL [] RANK * in.ranks,
           VAL [] INT in.bases, VAL [] [max.indicies] INT in.strides)

PROC where.next (BOOL another,
              [] INT bases,
              [] INT counters, INT counters.pts,
        VAL [] [] INT in.steps
              VAL INT in.dimension, VAL [] INT in.sizes)
```

The translation of

```
WHERE R
  P
```

goes as follows:

1. Inspect the body for all references of distributions declared outside the loop and referred to in P. Call the set of these distributions $\mathcal{D}(P)$.

2. Inspect the set $\mathcal{D}(P)$ for subsets of distributions known to have been declared at the same "level" (in the same "`WHERE` context") and with the same rank. Call the set of these subsets of "common access" distributions $\mathcal{A}(P)$. The cardinality of $\mathcal{A}(P)$ is *num.arrays*.

3. Inspect the set $\mathcal{A}(P)$ for all distinct ranks belonging to the distributions in the set. Call the set of these ranks $\mathcal{R}(P)$. The cardinality of $\mathcal{R}(P)$ is *num.ranks*.

4. The size of (number of ranges in) the rank R is *dimension*.

5. Assemble *num.arrays* access structures into tables passed to *init.loop* as *in.bases* and *in.strides*. Assemble *num.ranks* rank structures into tables passed to *init.loop* as *in.ranks*. Declare space for *num.arrays* working base addresses, *num.arrays* $\times$ *dimension steps*, *num.ranks* rank structures applicable inside P, *dimension* counters, *another* and *counters.ptr*.

The translation is

```
... Declartions mentioned above
SEQ
  where.init (...parameter list detailed above)
  WHILE another
    SEQ
      P (bases)
      where.next (...parameter list)
```

I think I'm going to have to put this to sleep for a while and come back to it in a few weeks (after I've actually done the first-cut implementation of the level I comms library, probably). Closing thoughts on the current status...

- The detailed specification of the various data structures—access, range, rank, index—is still to be done yet. A lot of it will hinge on permitting an efficient implementation of *where.init*.
- The above translation applies to the most dynamic form of `WHERE` loop. It provides a starting point for numerous optimisations applicable in more static instances. In practise many of the structures passed around here would only need to exist in the compiler. Calls to the two procedures above would be replaced by inline code, obviating the need to pack pointers into arrays for parameter passing. Dimensions of most arrays introduced here would be known at compile time, and the arrays would be implemented by sets of scalars, eliminating loop and indexing overheads. In nearly all cases *dimension* would be known at compile time and the `WHILE` loop would be replaced by nested `FOR` loops. It would be hoped that with rather straightforward compiler optimisations of this nature most `WHERE` loops could be implemented about as as efficiently as a set of nested replicated `SEQ`s—in fact possibly even more efficiently because indexing of multi-dimensional arrays would be handled by simple incremental calculations.
- Current plans for the "hashing" from grid site to process will incorporate the most direct decomposition and straightforward wraparound mapping as special cases—this hashing could even be changed at configuration level by changing parameters with no alteration to "user" code.

DISCLAIMER: Where I refer to the "compiler" above, it should be understood that the compiler I have in mind is a human being.