University
of Southampton

Techniques for the Automatic Parallelisation of 'Distributed Fortran 90'

John Merlin

SNARC 92-02

Technical Report Series
Southampton Novel Architecture Research Centre

University of Southampton
Novel Architecture Research Centre

Within the Department of Electronics and Computer Science, the Novel Architecture Research Centre is the main focus for research on parallel systems and applications, and includes many of the activities of the Concurrent Computation Group (CCG). Close liaison is maintained with the Computer Science Group who have a stong interest in languages for parallel systems. Many other groups within the department also have projects either utilising or directly concerned with parallel computation.

Support for applications development both within the Department and in the University as a whole has always been one of the groups major activities. Recognition of, and support for, this role has recently provided with the establishment of the Southampton Novel Architecture Research Centre (SNARC), funded by SERC. SNARC itself covers a full range of activities from basic hardware and software development, through applications and tools for parallel systems.

Apart from individual projects SNARC is involved in much wider scale projects funded by the ESPRIT program, which involve collaboration with academic and industrial partners across Europe. Southampton has participated in such projects for a number of years, starting with the successful Supernode project in 1985. Involvement continued with the PUMA and GENESIS projects, and several of the ideas developed within these will bear fruit in the form of hardware and software to be developed in the GP MIMD project and other ESPRIT-3 projects.

SNARC also maintains close contacts with other Parallel Computing groups within the University of Southampton. The technology transfer role of the CCG was originally provided by Transputer Technology Solutions (TTS), which was one of the original Transputer Support Centres funded by the SERC/DTI Transputer Initiative. This role has been substantially extended by the establishment of the Southampton University Parallel Applications Centre (PAC), funded by the SERC/DTI Parallel Applications Programme. The University Computing Service now provides a parallel computing user service, with support through its Parallel Computing Support Group (PCSG).

Techniques for the Automatic Parallelisation of 'Distributed Fortran 90'

John Merlin

Department of Electronics and
Computer Science
University of Southampton
Southampton SO9 5NH
United Kingdom

SNARC 92-02
ESPRIT P2701 (PUMA) Report D4.3.2

Department of Electronics and Computer Science

University of Southampton

Southampton S09 5NH

Contents

1 Introduction . . . 3
2 Data distribution . . . 5
2.1 'Distributed Fortran 90' . . . 5
2.2 Restrictions in 'Distributed Fortran 90' . . . 9
2.2.1 Restrictions on the use of distributed data . . . 9
2.2.2 Other restrictions . . . 11
2.3 Array handling at procedure boundaries . . . 13
2.4 More general data distribution schemes . . . 14
3 Basic transformations . . . 15
3.1 Array declarations and subscripts . . . 15
3.2 Attribute vectors . . . 17
3.3 Regular sections and section parameters . . . 19
3.4 Temporary and automatic arrays . . . 20
4 Transformation of array expressions and assignments . . . 23
4.1 Overview of array expressions . . . 23
4.1.1 Expressions in Fortran 90 . . . 23
4.1.2 Expression trees . . . 25
4.2 Target distributions . . . 25
4.2.1 Assignment target distributions . . . 26
4.2.2 Actual argument target distributions . . . 32
4.2.3 Scalar and subscript target distributions . . . 36
4.2.4 Arbitrary target distributions . . . 37
4.2.5 Determining unknown target distributions . . . 38
4.2.6 Operand target distribution and co-distribution criteria . . . 42
4.3 Processing an expression . . . 44
4.4 Processing a function reference . . . 48
4.5 Processing an array section, and data redistribution . . . 52
4.5.1 Storage allocation . . . 56
4.5.2 Dimensional redistribution . . . 56
4.5.3 Sectional redistribution, and co-distribution criteria . . . 60
4.6 Minimising storage allocation for target distributions 62
4.7 Processing an assignment . . . 63
4.8 WHERE statement and construct . . . 68

5 Suggested language extensions 70
5.1 FORALL loops 70
5.2 User-defined ELEMENTAL procedures 72
6 The ADAPT pre-processor 73
7 Conclusions 75
Bibliography 77
A ADLIB procedure interfaces 79
A.1 Overview 79
A.2 Data redistribution 80
A.3 Fortran 90 array intrinsic functions 83

1 Introduction

Distributed memory message-passing systems have many advantages over rival multi-processing architectures such as shared-memory and vector systems, among which are cost effectiveness, scaleability and, for many applications, high efficiency. Currently, however, their acceptance is hindered by the difficulties of programming them. To program data parallel applications, for instance, it is necessary to explicitly decompose the data into a collection of pieces, each 'owned' by a single process, and specify all the data communications between the processes. This is a complicated and error-prone task, and the resulting programs are usually not portable and are hard to adapt. It would appear that these deficiencies in *programmability* and *portability* must be overcome for distributed memory architectures to achieve widespread use.

In order to address these issues, at least for data parallel applications, we have been developing a system to automatically transform data parallel programs written in standard Fortran 90 [1] so that they can be executed on T9000+C104 networks. Parallelisation is by array partitioning, that is, by distributing data arrays over a (logical) multi-dimensional process array, guided by 'distribution' declarations that the programmer inserts into the program. The system is based on implementing data parallel execution of the array syntax of Fortran 90, automatically inserting communications where necessary. We use the name 'Distributed Fortran 90' for the input language for this system, namely Fortran 90 plus distribution declarations.

This research is based on exploiting new developments in both hardware and programming languages, namely, the efficient universal message-passing provided by the T9000 + C104 architecture, and the array syntax and other new features introduced to Fortran by the Fortran 90 standard.

Universal message-passing is required for an effective auto-parallelisation system for two main reasons. Firstly, it permits data partitioning of arbitrary dimensionality, so that the potential data parallelism of an application can be fully realised (*cf* T800 architectures restrict data partitioning to 2 dimensions unless special purpose communications harnesses are used). Secondly, it allows the efficient implementation of general array operations, many of which require long-range communications (*e.g.* gather-scatter, broadcasts, transpose, reduction operations, *etc*).

We have concentrated on *new* language features of Fortran 90 because they simultaneously assist both the auto-parallelisation and programming tasks. There are well known difficulties with trying to transform programs written in conventional sequential languages (*e.g.* Fortran 77) into a form suitable for distributed memory execution. One difficulty is the unnecessary sequentialisation (`DO`-loops) introduced by such languages. Another is that 'dusty deck' codes, in Fortran at least, tend to contain many features that hinder the analysis necessary for distributed memory implementation, such as aliasing, equivalencing, and automatically reshaping arrays across procedure boundaries.

Fortran 90's array syntax, on the other hand, provides an attractive source language for several reasons: it is implicitly data parallel; it can be readily implemented on a wide variety of parallel architectures, including SIMD, vector and shared memory systems, and so is portable across computer architectures; and, equally important, it provides a clear, concise and natural programming style. Fortran 90 also introduces other features that are potentially helpful for auto-parallelisation, and which this research exploits. For example, the provision of explicit procedure interfaces removes the need for interprocedural analysis and permits checking, and in the present context, allows the partitioning of array parameters to be assumed or automatically modified across procedure boundaries.

The system that we have been developing has two components:

ADAPT ('Array Distribution Automatic Parallelisation Tool'), a *pre-processor* to transform a Distributed Fortran 90 program into a standard Fortran 77 program that constitutes a component process of the process array

ADLIB ('Array Distribution LIBrary'), a *library* of communications procedures and Fortran 90 array intrinsic functions (generalised to handle both distributed and non-distributed data), that are called as necessary by the transformed Fortran program.

(In fact, the nomenclature has not been completely resolved, as we also use the name ADAPT for the complete system!) The reason for transforming to Fortran 77 is to allow the output program to be compiled by commonly available Fortran 77 compilers.

Although this system was developed with the features of the T9000 + C104 architecture in mind, the techniques it employs are applicable to all distributed memory systems that support universal message-passing. Furthermore, it can be ported to another platform by adapting just the ADLIB library, as its routines perform all of the message-passing and other machine-specific functions. The pre-processor need not be modified, as it generates standard Fortran 77 without machine-dependent extensions.

It has not been possible to complete the implementation of the whole ADAPT system within the timescale of the PUMA project, but the development of the pre-processor is close to completion.

This report gives a detailed description of the techniques employed by ADAPT for the auto-parallelisation of Distributed Fortran 90. It assumes that the reader is familiar with Fortran 90, especially with its array features and other new facilities, such as procedure interfaces, on which the auto-parallelisation is based. If not, an overview of these features can be found in [2, 3]. These references also provide an overview of the execution model for the transformed output program, while this report describes the execution model in more detail.

The layout of the report is as follows. Section 2 describes the data distribution model and declarations of Distributed Fortran 90, and the restrictions it imposes on the use of distributed data and of some old Fortran features. Sections 3 and 4 describe in detail the auto-parallelisation techniques employed by ADAPT, and comprise the main body of the report. Section 3 introduces the basic principles and techniques, such as array partitioning, subscript transformation, attribute vectors, regular section parameterisation, and dynamic storage allocation. Section 4 describes the transformation of array expressions and assignments, which is the core of the auto-parallelisation process. Section 5 suggests two further language extensions to Distributed Fortran 90 (or perhaps Fortran generally) that would assist distributed memory auto-parallelisation while not impairing language portability. Section 6 contains a brief overview of the ADAPT pre-processor, and the last section presents our conclusions.

An appendix describes the interfaces of the ADLIB library procedures.

2 Data distribution

2.1 'Distributed Fortran 90'

The only addition required to standard Fortran 90 programs to enable them to be parallelised by ADAPT is to declare how data arrays are to be partitioned over the process array[1]. We use the term 'Distributed Fortran 90' for the extended version of Fortran 90 that includes this partitioning information.

It provided by declaring a `DISTRIBUTION` attribute for partitioned arrays, whose usage is similar that of the standard `DIMENSION` attribute. (The best place for the `DISTRIBUTION` attribute is alongside the `DIMENSION` attribute in the type declaration of the array. However, Fortran 90 also allows dimension information to be given in separate `DIMENSION` statements, so for the sake of consistency the partitioning information may also be given in separate `DISTRIBUTION` statements.) The `DISTRIBUTION` keyword is followed by a parenthesised list of non-negative integer expressions, one for each dimension of the data array. A value $p > 0$ indicates that the dimension is distributed over dimension p of the process array. The value 0 indicates that the dimension is *not distributed*, i.e. that the whole dimension is held internally on each process. All non-zero values must be distinct, as only one data dimension can be distributed over any given process array dimension.

For example, the declaration:

```
REAL, DIMENSION (8,8,8), DISTRIBUTION (2,0,1)  :: a
```

[1] We talk of *processes* rather than *processors*, as the transputer architecture allows many processes to run concurrently on each processor.

results in the first data dimension being distributed over dimension 2 of the process array, the second data dimension being held internally in each process, *etc*. If this program is targetted to a 2×2 grid of processes, each process is assigned a segment of the data array as follows:

p_1	p_2 0	1
0	(1:4, :, 1:4)	(5:8, :, 1:4)
1	(1:4, :, 5:8)	(5:8, :, 5:8)

Notice that the size of the process grid is *not* declared in the program. In fact, this information is provided to ADAPT in an separate file (whose filename has a default value or is declared in the command line) in the following form:

```
proc_array = (3,2,4)
```

To run the program on a different-sized process array it is only necessary to change the parameters in this file and run ADAPT again; the input program need not be modified.

The extent of a data array dimension need not be exactly divisible by the number of processes across which it is distributed. In general, if N elements are distributed across P processes, they are allocated to the processes as follows:

> $n = \lceil N/P \rceil$ elements are allocated to each of the first $\lfloor N/n \rfloor$ processes;
>
> the remaining $N \backslash n$ elements (if any) are stored on the next process;
>
> no elements are allocated to the remaining processes (if there are any).

($\lceil x \rceil$ is the smallest integer $\geq x$, $\lfloor x \rfloor$ is the largest integer $\leq x$, and $m \backslash n$ is the remainder of $m \div n$.) The following table gives some examples of this distribution scheme for 4 processes ($P = 4$):

number of elements (N)	process number (p) 0	1	2	3
	elements allocated to each process			
4	1	2	3	4
5	1,2	3,4	5	-
6	1,2	3,4	5,6	-
7	1,2	3,4	5,6	7
8	1,2	3,4	5,6	7,8
9	1,2,3	4,5,6	7,8,9	-

This mapping is chosen in preference to a more 'uniform' distribution for several reasons: (i) it simplifies the formulae for element location; (ii) it often simplifies the communications required to make regular sections co-distributed; and (iii) it preserves optimal load balancing, that is, it minimises the maximum number of elements per process.

Arrays need not be distributed, in which case no `DISTRIBUTION` declaration is necessary, or they may be distributed over a subset of the process grid dimensions. For instance, if a program is targetted to a 2-dimensional process grid, any vectors that it contains can only be distributed over one of the grid dimensions. Scalars and non-distributed arrays are *replicated* – a copy of the complete data object is kept on every process. Similarly, arrays that are distributed only over a subset of the process grid dimensions are replicated over the others. The parallelisation scheme guarantees that replicated data values are consistent, that is, all copies have the same value at any step of the program. This is because an assignment to a replicated object would be performed to all copies simultaneously, as we shall show later.

Arrays may also be declared to be distributed over *non-existant* process grid dimensions. For example, the declaration:

```
REAL, DIMENSION (8,8,8), DISTRIBUTION (1,2,3) :: a
```

may appear in a program that is compiled for a 2- or 1-dimensional process array, or even a single process. Data dimensions that are mapped onto non-existent process dimensions are not distributed – the whole dimension is held internally on every process. (An equivalent viewpoint is that the process grid has an extent of 1 in these non-existent dimensions, so the data are distributed over 1 process.) This allows programs that are targetted to large process arrays to be re-compiled for small process arrays or even a single process if necessary.

It is worth emphasising the distinction between the words *internal*, *replicated* and *global*. 'Internal' applies to data dimensions that are not distributed. 'Replicated' applies to data objects that have more than one copy – distinct copies being kept in distinct processes, and 'global' applies to data objects that have a complete copy in *every* process. Basically, a data object is replicated *unless* it is distributed over every dimension of the process array. Thus, an array may have some internal dimensions and yet not be replicated, because it has a dimension distributed over each process array dimension. On the other hand, it may have no internal dimensions and yet be replicated, because its rank is less than that of the process array, so it cannot be distributed over the whole process array.

Assumed and variable distribution An array that is a dummy argument can take its distribution from the corresponding actual argument, rather than having it declared explicitly. This is analogous to the *assumed-shape* feature provided by Fortran 90 for dummy arrays, and is accordingly called *assumed-distribution*. It is denoted by omitting the expressions from the

`DISTRIBUTION` attribute (although the appropriate number of commas must be supplied).

The distribution of a dummy or local array can also depend on the procedure's dummy arguments, rather than being constant. In the case of a local array, this means that the shape of its local segment is only determined when the procedure is called. Such an array is called *automatic*, meaning that it comes into existence just for the duration of the procedure call. This is a natural extension of the Fortran 90 provision for automatic arrays, which are local arrays whose *shapes* are functions of the procedure's dummy arguments.

There is also an extra intrinsic function:

```
DISTR (array [,dim])
```

that returns the distribution of an array, analogous to the standard Fortran 90 intrinsic functions `SIZE`, `LBOUND` and `UBOUND` for dimension information.

The next example shows an array function that uses some of these distribution facilities

```
FUNCTION matvec (mat, vec)
   REAL, DIMENSION (:,:), DISTRIBUTION (,) :: mat
   REAL, DIMENSION (SIZE(mat,2)),                    &
            DISTRIBUTION (DISTR(mat,2)) :: vec
   REAL, DIMENSION (SIZE(mat,1)),                    &
            DISTRIBUTION (DISTR(mat,1)) :: matvec

   DO i = 1,SIZE (matvec)
      matvec (i) = SUM (mat (i,:) * vec (:))
         ! SUM () is a Fortran 90 intrinsic function
   ENDDO
END FUNCTION matvec
```

The programmer is free to distribute arrays in any manner chosen. As we shall see below, ADAPT is designed to handle any data distribution in any context, and to insert data communications where necessary. However, it is obvious that an unsuitable data distribution may severely degrade the performance of the resulting data-parallel program by increasing communications and/or decreasing the potential for data concurrency. We envisage that an optimal data partitioning would typically be achieved as a result of a limited amount of experimentation.

2.2 Restrictions in 'Distributed Fortran 90'

2.2.1 Restrictions on the use of distributed data

A design philosophy of the ADAPT system is that:

> there are *no* restrictions on the use of distributed data in *executable statements*, provided that any external procedure that is referenced has an explicit interface.[2]

The transformation scheme employed by ADAPT is designed to handle any distribution of data elements appearing in executable statements, and to insert data communications where necessary. Some examples of the flexibility that this allows the programmer are:

- The primaries in an expression can have any data distribution, regardless of the distribution of other primaries.
- Distributed data can appear in any type of expression in executable statements. Thus, it is permitted in subscript expressions, `DO`-loop and even implied `DO`-loop control expressions, logical expressions in `IF` statements and constructs, output expressions, actual arguments, etc., as well as in assignment statements.
- An expression in an assignment statement can have any data distribution, regardless of the distribution of the assignment variable.
- A procedure reference may have actual arguments with any data distribution, *provided the procedure interface is explicit*. The transformation scheme will re-distribute an actual argument if necessary to match the required distribution of the dummy argument. (Section 2.3 describes how data is handled at procedure boundaries, including the case where *no* procedure interface is provided).
- I/O can be performed to or from data with any distribution, including arbitrary regular or irregular sections of arrays.

The only restrictions on the use of distributed data are in declarations. They are:

1. `PARAMETER` arrays cannot be distributed.
2. Specification expressions (i.e. expressions which declare array bounds and distributions, and character lengths) cannot involve distributed data.

[2] This means that the calling procedure knows the specifications of its dummy arguments and, in the case of a function, its result. This information is made available by declaring an '`INTERFACE` block' within the calling procedure. This feature is new to Fortran 90.

3. Distributed arrays cannot be statically initialised (i.e. they cannot be initialised in a `DATA` or type declaration statement).

4. If a distributed array appears in `EQUIVALENCE` statement, the *whole* of the array must be specified. Furthermore, no correspondence can be assumed between the elements of equivalenced distributed arrays unless they have *identical* distribution (i.e. the same shape, distribution and wordlength), or between the elements of a distributed and a non-distributed array.

Restriction 1 is obvious. Basically, `PARAMETER`s are constants whose values are known to the compiler. If they were distributed, then in the output program only some of them would be available – those held in the local segment of the 'constant' array – the others would have to be input from other processes if their values were required, and so by definition would not be constant. There is another difficulty, which is that the output program would have to contain a process-dependent initialisation of the local segment of the 'constant' array, but initialisation is done at compile-time and so cannot be process-dependent.

Restriction 2 is also obvious. A specification expression can be constant or variable. If it is constant then it cannot be distributed, as explained above. If it is variable, then all its data elements must be defined on entry to a procedure, and in particular before any executable statements; therefore, the output program cannot perform communications in order to obtain data for a distributed specification expression!

In principle, ADAPT could generate output code which achieved an equivalent effect to the static initialisation of distributed arrays (restriction 3). Obviously distributed data cannot *actually* be statically initialised in the output program, since static initialisation is done at compile-time and so cannot be process-dependent. It would have to be implemented by equivalent assignments to the local segment of the data, at the start of the procedure, which would only be performed the first time the procedure is executed. The variables concerned would also have to be given the `SAVE` attribute (if it was not explicitly declared in the input program). However, this transformation is very complicated, particularly because of the variety of forms allowed in `DATA` statements. It is easier to force the user to initialise such data in ordinary assignments, which are then handled by ADAPT's normal transformation scheme!

Restriction 4 should be self-explanatory. Distributed data is partitioned into segments allocated to different processes, so no relation can be assumed between the elements of equivalenced distributed arrays unless they have identical distribution.

The next item is not a restriction, but a *warning*:

5. If an explicit interface is *not* specified for a procedure, then ADAPT has know way of knowing the required target distribution of its arguments, so array arguments are passed-in with their pre-existing distributions (except that array *sections* are copied into temporary arrays with contiguous storage and uniform distribution, as we explain in section 2.3). Therefore, if a dummy array has its distribution explicitly specified (rather than assumed) the programmer must ensure that the corresponding actual argument matches this distribution. Note that this is only possible if the actual argument is an array variable, as the result of an array expression has unspecified distribution.

For this reason, the programmer is *strongly recommended to provide explicit interfaces for all procedure calls*, thereby avoiding potential problems with data distribution across procedure boundaries. Explicit interfaces also provide much greater security by allowing the correspondence between actual and dummy arguments to be checked, and they are necessary in order to use some of the valuable new features of Fortran 90, such as array-valued functions, optional arguments, keyword-specified arguments, and the specification of argument *intent* (which specifies whether an argument provides an input value, an output value, or both, and thus enables more thorough checking).

2.2.2 Other restrictions

A programming practice that is permitted in some circumstances in standard Fortran 90 is completely forbidden in Distributed Fortran 90, namely:

> In a procedure reference, shape disagreement between an actual argument and the corresponding dummy argument is *not* allowed.

In fact, standard Fortran 90 only allows shape disagreement in limited circumstances [4], and anyway it is a deprecated feature in the new standard (that is, it is an old Fortran feature that is no longer recommended and for which better alternatives exist in Fortran 90). We shall now briefly explain why it is incompatible with distributed data.

There are various manifestations of shape disagreement in 'old' Fortran (e.g. Fortran 77). Firstly, an *array element* can be passed as the actual argument corresponding to a *dummy array*. Since all arguments are passed by address in Fortran 77, the *address* of the array element is passed to the procedure, where it is taken to be the starting address of the dummy array, whose extent is determined by its local declaration. Thus the dummy array is associated with the storage of the actual argument array, starting at the address of the element that was passed. This only works in Fortran 77 because all arrays occupy contiguous storage. It cannot work if the actual argument array is *distributed*, since then a particular array element may only be stored in one process, and its address is irrelevant and indeed unknown in other processes. Therefore, in Distributed Fortran 90, a different interpretation is applied to an actual argument that is an element of a distributed array – it is treated as a *scalar* argument whose value must be passed to the procedure in all processes, so it is broadcast. Thus the address that is passed to the procedure is that of a temporary variable containing the broadcast value, and nothing can be assumed about the contents of adjacent memory locations.

Another manifestation of shape disagreement in Fortran 77 is automatic 'reshaping' of an array across procedure boundaries. Thus, an array declared as `A(100)` may be passed to a procedure as the actual argument `A` and received as a dummy array declared as `D(10,10)`. Once again the association is by storage sequence, in array element order, and again this relies on arrays occupying contiguous storage. It does not work with distributed data, since now a different shape implies a different data distribution.

This conclusion applies even if the rank is not changed. For instance, assume that a procedure declares an array

```
INTEGER, DIMENSION (52):: cards
```

and contains the following procedure call:

```
CALL shuffle (cards, 13)
```

and that subroutine `shuffle()` contains following declarations:

```
SUBROUTINE shuffle (hand, n)
   INTEGER n
   INTEGER, DIMENSION (n) :: hand
```

(we omit `DISTRIBUTION` declarations for simplicity). The intention is obviously for `shuffle()` to act on just the first 13 elements of `cards`. This does not work in Distributed Fortran 90. As we shall see, the current implementation assumes that an array whose shape is explicitly declared (for example, `hand(n)`) is uniformly distributed over the relevant process grid dimension. If the actual argument were `cards(1:13)`, the relevant section *would* be re-distributed so that it is correctly associated with `hand`, but as it stands `hand` will be incorrectly associated with a uniformly distributed array of 52 elements.

All of these problems concern distributed data. However, ADAPT's transformation scheme generates incorrect results in the presence of shape disagreement even if both actual and dummy arguments are *not* distributed. This is because it introduces an 'attribute vector' for every array argument (as we shall see in section 3.2), which contains a complete description of the array's shape and distribution. Shape disagreement causes this information to be wrong, leading to incorrect behaviour of the output program.

All of the facilities provided by shape disagreement in Fortran 77 can be achieved by other means in Fortran 90 which *are* compatible with distributed data, namely, by using array sections and the intrinsic function `RESHAPE` to obtain the required actual arguments.

2.3 Array handling at procedure boundaries

At this point it is useful to briefly consider how arrays are handled at procedure boundaries. We start by explaining the array-passing mechanisms suggested by the Fortran 90 standard, which are implemented in ADAPT:

- An actual argument that is a *regular section* of an array may be handled in two ways. Firstly, notice that the regular section may select elements that are not stored contiguously (e.g. `A (1:9:2)`).

  If the corresponding dummy argument has *explicit shape* (i.e. its shape is explicitly specified rather than assumed), then the selected elements of the regular section are copied into a temporary array with contiguous storage, which is passed into the procedure, and the temporary array is copied back on return. This is done because all arrays whose shapes are explicitly specified (whether by constant or variable expressions) are taken to occupy contiguous storage. This is the default array-passing mechanism, which is used if the interface of the procedure is not specified.

  However, if the dummy array is *assumed shape* a different mechanism is used, namely, the original array is passed into the procedure accompanied by 'section parameters' that specify the regular section (which will be described in section 3.3). This mechanism avoids the expense of copying arrays at the cost of a more complicated indexing scheme. In the case of ADAPT, copying an array section into a temporary array may be even more expensive than normal as it may involve data re-distribution, so the assumed shape mechanism is especially useful.

- If an actual argument is an *irregular* array section (i.e. a section with one or more vector subscripts), then it is treated as an expression. The irregular section is copied into a temporary regular array which is passed into the procedure, but no copy-back is performed on return, so the procedure should not assign to the corresponding dummy argument. (If the `INTENT` of the dummy argument is specified in an interface block, it must have `INTENT (IN)`).

Incidentally, the Fortran 90 standard requires that a procedure interface must be explicit if any of its dummy arguments are assumed shape, because the calling procedure must pass additional information in this case. In fact, ADAPT correctly handles assumed-shape dummy arguments even if the interface was not specified at the point of call, because the required information is always passed into procedures anyway – as we explain in section 3.2. However, this feature is non-portable, and it may not be supported in future versions of the system.

The treatment of array *distribution* across procedure boundaries by ADAPT mirrors the treatment of array shape, namely:

- If the distribution of a dummy argument array is explicitly specified (either constant or variable), then the actual argument is re-distributed if necessary to match it, assuming that the procedure interface is explicit.
- If a dummy argument array has *assumed* distribution, then it can correspond to an actual argument of any distribution, so that in general it is not necessary to re-distribute the actual argument. Instead, the original array is passed to the procedure accompanied by a vector containing details of its distribution. Only the *rank* of the dummy and actual arguments must match. (However, re-distribution may still be necessary if an *array section* is passed to an explicit shape dummy array, since then the array section must be copied to a contiguous array with uniform distribution, as we explained above).
- If an explicit interface is *not* specified for a procedure, then ADAPT has know way of knowing the required target distribution of its arguments, so array arguments are passed-in with their pre-existing distributions, except that *sections* are copied into temporary arrays with contiguous storage and uniform distribution, on the assumption that the actual arguments are explicit shape rather than assumed shape. We have already pointed-out the risks associated with this mechanism!

If an actual argument is an array *variable*, as opposed to an expression, and it was copied into a temporary array before being passed to the procedure, then it is copied back to the actual argument on return from the procedure, unless the actual argument is an irregular section or the corresponding dummy argument has `INTENT(IN)`. Note that this may again involve re-distribution.

2.4 More general data distribution schemes

The data partitioning strategy implemented in this version of the system is somewhat limited. For some problems, an optimal data distribution (in the sense of minimising communications) would require the specification of more general distributions, for example:

- Allowing an array to be distributed so that its elements are aligned with a *regular section* of a larger, regularly distributed array, which may be regarded as a *reference distribution*. This allows a pair of arrays with different sizes to be distributed so that certain elements are aligned, for example, so that an array `F(1:N,1:N)` can be co-distributed with the interior section of an array `G(0:N+1,0:N+1)`, which may be useful for such applications as solving Poisson's equation.

- Allowing elements, or blocks of elements, to be distributed cyclically around a ring of processes.
- Mapping a single data dimension onto several process grid dimensions, or several data dimensions onto a single process dimension.

Of course, these mappings could also be combined, and it is not difficult to think of other general types of data partitioning. The first two mappings above can be specified using the Fortran-D extensions for specifying data distribution [5].

The present scheme was chosen as a simple starting point, and we hope to extend it to more general data distributions, such as those in Fortran-D, in future research. In fact, the current system already contains most of the mechanisms required to support the first enhancement listed above, since these distributions can be considered as 'regular sections' of uniform distributions, which ADAPT handles routinely in the context of assumed-shape arrays and array expressions, as we shall see later.

3 Basic transformations

This section presents some fundamental features of the parallelisation of Distributed Fortran 90 by ADAPT, namely: array partitioning and subscript transformation, attribute vectors, sections parameters and dynamic memory allocation.

3.1 Array declarations and subscripts

Every array declaration in the input program is transformed into a declaration of the *local segment* of the array. If an array dimension with extent n is distributed over *np* processes, the corresponding dimension of the local segment is declared to have extent *locn*, given by[3]:

$$locn = (n + np - 1)/np$$

Note that this is actually the *largest* extent of a local segment in that dimension, bearing in mind that the total number of elements in a dimension may not be exactly divisible by the number of processors over which it is

[3] This is the Fortran integer expression corresponding to $\lceil n/np \rceil$. In this section we shall generally use Fortran-style integer expressions rather than mathematical formulae.

distributed (*np*). Because of the Fortran 77 requirement for static memory allocation, it is necessary to declare the size of the largest segment on all processors.

In addition to this transformation, all lower bounds are transformed to 0 (*normalised*). This is done for several reasons. Firstly, when a distributed dimension is partitioned over the process array, each local segment has a lower bound which, in terms of the original subscripts, depends on the coordinate of the process. This is only determined at runtime and so cannot appear in the Fortran 77 array declaration, which requires constant array bounds. Therefore, the lower bound of the local segment is given the same *constant* value in all processes, and subscripts are transformed into *local* subscripts with respect to this new lower bound (so that now the local subscript of a particular element is process-dependent, rather than the lower bound). Even for internal dimensions it is advantageous to change all lower bounds to a fixed value, for reasons that we shall explain in the next subsection. 0 is chosen in preference to 1, as it simplifies subscript calculations.

It is evident that there are various possible subscript sets for an array dimension. They are given the following names:

original subscripts: The subscripts as specified in the input program, namely, with respect to the full array (not just the local segment) and the declared lower bounds.

normalised global subscripts: Subscripts with respect to the full array but with the lower bounds normalised to 0.

real global subscripts: This description refers to the fact that a named array, A say, may in fact be a *regular section* of another named array, *Aorig*, so that A's elements may have an offset and stride with respect to the contiguous storage allocated to the parent array *Aorig*. (This may be the case if A is an assumed-shape dummy array, for instance, as we explained in section 2.3). In that case, the above two subscript sets are with respect to the *subset of elements belonging to A*, whereas the corresponding *real global* subscripts are with respect to the underlying parent array *Aorig*, that is, with respect to the layout of the array in memory. Put another way, the above two subscript sets are *logical* subscripts while these are *physical* subscripts.

As for the previous subscript set, this subscript set is normalised to a lower bound of 0. Obviously, if the subscripted array is not a regular section, then the real and normalised subscripts are the same.

local subscripts: Subscripts with respect to the local segment, starting at a lower bound of 0 in every dimension of the local segment. These are also *real* subscripts, in the sense given above.

Notice that even if the original subscripts appearing in the input program are constant, the corresponding *local* subscripts will be non-constant in distributed dimensions, and for arrays that are regular sections, both the *real* and *local* subscripts will be non-constant. Section 3.3 contains formulae for transforming between these subscript sets.

3.2 Attribute vectors

Every array appearing in the output program has an 'attribute vector' declared and initialised for it. This contains complete information about the type, dimensions, and distribution of the array. It is needed for two principal reasons: to pass this information to the ADLIB procedures that are referenced in the output program (which perform data communications, I/O, Fortran 90 array intrinsic functions, memory allocation, etc.), and to pass it to user-defined procedures that take assumed shape or distribution dummy arrays.

The attribute vector, *attrvec*, contains the following information:

```
INTEGER :: type, len, rank
INTEGER, DIMENSION (rank) :: n, d,                   &
                             locn, locl, locu,       &
                             sectn, sectl, sects
INTEGER, DIMENSION (3+8*rank) :: attrvec
attrvec = (/ type, len, rank,                        &
             ( n(i), d(i),                           &
               locn(i), locl(i), locu(i),            &
               sectn(i), sectl(i), sects(i),         &
                                i = 1,rank ) /)
```

type, *len* and *rank* are respectively (an integer representation of) the data-type of the array, the total number of elements of the *local* segment, and the rank.

$n(i)$ is the extent of dimension i of the *whole array*, as declared in the input program. If dimension i has declared lower and upper bounds of l_i and u_i respectively, then $n(i)$ is given by:

$$n(i) = \mathrm{MAX}(u_i - l_i + 1, 0)$$

$d(i)$ is the process array dimension over which data dimension i is distributed, as specified in the `DISTRIBUTION` attribute. We call it the *process mapping*. Unfortunately we cannot use the more obvious name of 'distribution', as we shall find it convenient to reserve that word to mean a *complete description* of the mapping of data elements onto the process array, which comprises all of the information in the attribute vector, not just its d parameters.

locn(i) is the extent of dimension i of the *local segment*, as declared in the output program. It is given by:

$$\mathit{locn}(i) = \begin{cases} n(i) & d(i) = 0 : i \text{ internal;} \\ (n(i) + \mathit{np}(d(i)) - 1)/\mathit{np}(d(i)) & d(i) > 0 : i \text{ distributed} \\ & \text{over } \mathit{np}(d(i)) \text{ processes.} \end{cases}$$

locl(i) is the lower bound of dimension i of the local segment, in terms of *real global* subscripts. It is given by:

$$\mathit{locl}(i) = \begin{cases} 0 & d(i) = 0 : \text{ internal dimension } i; \\ p_i \star \mathit{locn}(i) & d(i) > 0 : \ p_i \text{ is the process coordinate in} \\ & \qquad \text{dimension } d(i) \text{ of the process array.} \end{cases}$$

To explain the meaning of *locu*(i), we must recall that the declared local segment may not be completely used in each process. This is because the total number of elements in an array dimension may not be exactly divisible by the number of processors over which they are distributed, leading to a non-uniform allocation of data elements to processes. This effect is parameterised by the vector *locu*. Its $i^{\underline{\text{th}}}$ component gives the maximum *local subscript* in dimension i of a *used* element of the local segment. With the chosen distribution strategy for non-uniform distributions (section 2.1), the formulae are:

$$\mathit{locu}(i) = \begin{cases} \mathit{locn}(i) - 1 & \text{internal dimension or} \\ & p_i < n(i)/\mathit{locn}(i); \\ & \\ \mathrm{MOD}(n(i), \mathit{locn}(i)) - 1 & p_i = n(i)/\mathit{locn}(i); \\ -1 & p_i > n(i)/\mathit{locn}(i); \end{cases}$$

As before, p_i is the process coordinate in dimension $d(i)$ of the process array.

Incidentally, all of the *loc_*(i) parameters are redundant, as they can be computed from $n(i)$ and $d(i)$. They are included in the attribute vector merely to avoid frequent re-computation and to simplify subscript expressions in the output program.

We have already mentioned that, in the transformation of array declarations, *all* lower bounds are normalised to 0. We justified this for *distributed* dimensions (because the declared lower bounds of the local segment cannot be process dependent in the output program), but did not explain why it is also done for *internal* dimensions.

The reason is that, in Fortran 90, the declared lower bounds of an array can be different in each procedure in which it is used (whether it is a global or a dummy array). To support this, the output program would have to re-initialise the *locl* and *locu* parameters in the attribute vector of every non-local array at the start of a procedure, and change them back at the end, which would be very complicated. Notice also that if the original lower bounds were kept in the output program, then the *locl* parameter in *attrvec* would *not* be redundant, because obviously the ADLIB routines must know an array's lower bounds in order to interpret its subscript values. These complications are avoided by making all lower bounds 0.

The final attributes in *attrvec* are the section parameters, *sectn*, *sectl* and *sects*, which are explained in the next section.

3.3 Regular sections and section parameters

ADAPT introduces a parameterisation for regular sections, `A(1:10:2)`, so that they can be passed to procedures and used in expressions without the need to copy the selected elements into a temporary array (perhaps involving re-distribution). Instead, the original array is used and the regular section is specified by *section parameters*.

Section parameters are used by ADAPT for the following purposes:

1. They are used internally within ADAPT, to describe the distribution of array sections in array expressions, temporaries and variables.
2. They are installed in the 'attribute vector' of an array-valued actual argument corresponding to an *assumed-shape* dummy argument, thus permitting a regular section to be passed without the need for copying (see section 2.3). All of the array arguments of the Fortran 90 array intrinsic functions fall into this class, as indeed do the array arguments of all ADLIB procedures.

The section parameters for dimension i of an array are the parameters (*sectn*(i), *sectl*(i), *sects*(i)) in its attribute vector. If a *contiguous* array A has declared lower bound L_i in dimension i, and a regular section in that dimension is specified by the subscript triplet $l_i{:}u_i{:}s_i$, then the corresponding section parameters are:

sectn(i)	–	the number of elements, MAX$((u_i - l_i + s_i)/s_i, 0)$;
sectl(i)	–	the lower bound in *real global* subscripts, $(l_i - L_i)$;
sects(i)	–	the stride, s_i.

As special cases, a single element j has section parameters $(1, j - L_i, \text{-})$, and the whole dimension has section parameters $(n(i), 0, 1)$.

In some circumstances a regular section may be selected from another regular section (i.e. a 'section of a section'). In particular, a regular section may be taken of an assumed shape dummy argument array, which may itself correspond to a regular section of the associated actual argument array. If a regular section $l_i{:}u_i{:}s_i$ is specified in a dimension that is itself a regular section with section parameters $(\textit{sectn}(i), \textit{sectl}(i), \textit{sects}(i))$, the resultant section parameters are given by:

$$\textit{sectn}'(i) \quad = \quad \text{MAX}((u_i - l_i + s_i)/s_i, 0) \text{ must have } \textit{sectn}'(i) \leq \textit{sectn}(i)$$

$$\begin{aligned} sectl'(i) &= sectl(i) + (l_i - L_i) \star sects(i) \\ sects'(i) &= s_i \star sects(i) \end{aligned}$$

Note that the section parameters $sectl'$ and $sects'$ are always in terms of the *underlying, contiguous array* (cf. the lower bound l_i and stride s_i of the regular section are in terms of the subset of elements belonging to the previous regular section, which themselves have an offset of $sectl(i)$ and a spacing of $sects(i)$). These formulae are always used when taking regular sections of assumed shape dummy argument arrays. So that they give the correct results, the section parameters for a newly declared or allocated array are always intialised to $(n(i), 0, 1)$ – for a whole dimension.

At this point it is useful to give the formulae for transforming an original subscript J (appearing in the input program) into the various subscript types introduced in section 3.1. In these formulae, i is the array dimension to which the subscript belongs, and L_i is the declared lower bound in that dimension.

subscript type	formula
original	J
normalised global	$J' = J - L_i$
real global	$J'' = sectl(i) + J' \star sects(i)$
local	$j = J'' - locl(i)$

3.4 Temporary and automatic arrays

In Fortran 77 all non-dummy arrays must be static, that is, declared with constant bounds. However, Fortran 90 allows *automatic arrays*, which are local arrays that have non-constant bounds and exist only for the duration of a procedure call. In addition to this, in the course of transforming Distributed Fortran 90 into Fortran 77 it is often necessary to introduce *temporary arrays* with non-constant bounds, as will become evident in this report. Both types of array require a stack memory management scheme to be implemented in the output Fortran 77 program.

This is done by introducing a common block that contains a large vector of length `i$freelen` (a parameter in an `INCLUDE` file), ideally containing all of the free memory available. This vector, which we call the 'stack vector', is equivalenced to every datatype required by the temporary and automatic arrays. The common block also stores the subscript value, `i$ptr`, of the current 'top-of-stack' position in this vector, as well as a few vectors that are used to store the attributes of the next array to be allocated:

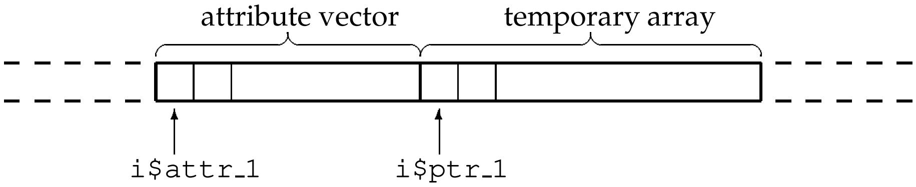

Fig. 1: Organisation of a temporary array and its attribute vector in the 'stack vector'

```
COMPLEX  c$temp (i$freelen/2)
DOUBLE PRECISION  d$temp (i$freelen/2)
INTEGER  i$temp (i$freelen)
LOGICAL  l$temp (i$freelen/2)
REAL     r$temp (i$freelen)
EQUIVALENCE (c$temp, d$temp, i$temp, l$temp, r$temp)
COMMON /c$stack/ i$ptr, i$temp, i$d(7), i$n(7)
```

The common block is declared in every procedure of the output Fortran 77 program.

Temporary and automatic arrays are taken taken from this stack vector. Each one has a variable, *ptr*, associated with it which contains the stack vector subscript corresponding to its first element. Element i of the temporary array is then referenced as element $(ptr + i)$ of the stack vector. A small drawback of this scheme is that all automatic and temporary must be referenced as one-dimensional vectors, even if they are logically multi-dimensional arrays.

Recall that every array in the output program is accompanied by an 'attribute vector' (section 3.2). This of course includes temporary and automatic arrays. Therefore, when space is allocated for an array it is also allocated for its associated attribute vector, which immediately precedes the array in the stack vector, as shown in Figure 1.

Allocation and de-allocation is performed by inserting calls in the output program to ADLIB procedures such as:

```
CALL s$alloc (i$attr_1, i$ptr_1, type, rank, n, d)
...
CALL s$dealloc (i$attr_1)
```

The first call allocates a temporary array with given `type`, `rank`, shape `n` and process mapping `d`, and also allocates its attribute vector (which is initialised). On return, `i$ptr_1` points to the first element of the temporary array and `i$attr_1` to the first element of its attribute vector. The second call frees the allocated storage. Because this scheme is stack-based, it is essential that arrays are deallocated in the reverse order to their allocation. ADAPT ensures this by placing the identifier of each allocated array in an internal stack and deferring its deallocation until its identifier is at the top of the stack again.

Note that the total length of the allocated space, *alloclen*, can be derived from information stored within it, namely from the *type*, *len* and *rank* parameters in *attrvec*:

$$\begin{aligned} alloclen &= (3 + 8 \star rank) \star 4 + len \star wordlen(type) \\ &\quad \text{rounded-up to a multiple of the maximum } wordlen. \end{aligned}$$

This is used by `s$dealloc()` to calculate the amount of storage to be freed.

To demonstrate the use of temporary arrays, we show how the declaration and use of an automatic array in Fortran 90 is converted into Fortran 77. The Fortran 90 version is:

```
        SUBROUTINE s (a, m, n)
           INTEGER  m,n
           REAL, DIMENSION(m,n) :: a
           REAL, DIMENSION(m,n) :: work
                                 ! `work' is an automatic array
           ...
           work = a              ! assignment to automatic array
           ...
        END SUBROUTINE s
```

In Fortran 77 this becomes:[4]

```
        SUBROUTINE s (a, i$attr_a, m, n)
           INTEGER  m,n, i$attr_a(0:18)
           REAL a(0:m-1,0:n-1)
           ... declaration of the 'c$stack' common block
           INTEGER  i$attr_work, i$ptr_work

  C-----allocate 'work' array at start of subroutine:
           i$n (1) = m
           i$n (2) = n
           i$d (1) = 0
           i$d (2) = 0
           CALL s$alloc (i$attr_work, i$ptr_work, i$real,
          +              2, i$n, i$d)
           ...
  C-----transform the assignment, work = a:
           DO i2 = 0, (n-1)
              DO i1 = 0, (m-1)
                 r$temp (i$ptr_work + i1 + m*i2) = a (i1, i2)
              ENDDO
           ENDDO
           ...
  C-----de-allocate 'work' array just before return:
           CALL s$dealloc (i$attr_work)
        END
```

[4]For clarity we use Fortran 90's DO–ENDDO format rather than Fortran 77's labelled DO-loops!

Temporary arrays (as opposed to user-defined automatic arrays) are normally introduced by ADAPT in the course of transforming array expressions and assignments, where they are required for data redistribution and for storing the results of array functions and expressions. They are only allocated for the duration of their useful lifetime, which usually corresponds to a single assignment statement in the original Fortran 90 program. This helps to reduce memory requirements. Despite this fact, the output programs generated from Distributed Fortran 90 are still likely to be expensive in terms of storage requirements, so attention has been paid to the problem of allocating temporary arrays in an optimal order, so that they can be freed as soon as possible.

This will be shown in the next section, which gives a detailed description of the transformation of array expressions and assignments.

4 Transformation of array expressions and assignments

In order to evaluate an expression or perform an assignment that involves distributed data, the data must be 'co-distributed', meaning that the corresponding elements of the variable and each term in the expression must be stored in the same process. If this condition is not satisfied *ab initio*, the data must be redistributed.

This section describes in detail the analysis and transformation of expressions and assignments performed by ADAPT, and the implementation of data redistribution. In its full generality this is very complex task, because there are absolutely no restrictions on the use of distributed data within expressions and assignments in executable statements (viz. an operand can have any data distribution, regardless of the data distribution in other operands or the assignment variable; a procedure actual argument can have a different data distribution from that required by the corresponding dummy argument; and even *subscript* expressions can use distributed data). However, this transformation is the core of the auto-parallelisation process, since all parallelism is obtained by evaluating the components of distributed array expressions concurrently over the process array. Also, all inter-process communications that are introduced into the output program arise as a result of this transformation.

Throughout this section, an *expression* is taken to mean a general array expression, which includes a scalar expression as a limiting case.

4.1 Overview of array expressions

4.1.1 Expressions in Fortran 90

To make the discussion more concrete and define some terms, we start with a brief review of expressions in Fortran 90 (neglecting features such as defined operators and derived types that are not supported by ADAPT).

An expression is an *operand* or a series of operands separated by *operators* (+, −, >, `.AND`, `//`, etc.). An *operand* can be a constant, a variable or a function reference, and can be scalar or array-valued. A variable or constant that is an array section or array element is specified by a list of *subscripts*, which are integer expressions that may be scalar (for single subscripts specifying a single element, and subscript triplets specifying a regular section) or vector (for vector subscripts that specify an irregular section). A function reference usually involves a list of *actual arguments*, which are themselves expressions whose type and shape match those of the corresponding dummy arguments in the function definition. Thus, expressions are defined recursively and can be arbitrarily complicated.

We shall often use the words *primary* or *term* for 'operand', and we shall also use *array section* as a generic term for a whole array, a subarray or a single array element. We should point out that the words 'primary' and 'operand' only refer to the 'top-level' data objects and function references of an expression, that is, they exclude terms that appear in subscripts or actual arguments of the given expression.

At this point it is useful to give a precise definition of the terms *rank* and *shape* in Fortran 90. The *rank* of an array is the number of declared dimensions that it has, including dimensions with extent 0 or 1. Thus an array declared as

```
REAL, DIMENSION (10) :: A
```

has rank 1 and shape (10),while an array declared as

```
REAL, DIMENSION (10, 1, 1) :: B
```

has rank 3 and shape (10,1,1). Notice that these arrays do not conform, and so cannot appear together in an expression such as `A + B` even though they have the same number of elements and a seemingly obvious correspondence between their elements. An array dimension may be distributed even if it contains only one element, in which case that element will reside in the first process in the given process grid dimension, and the other processes in that dimension are not allocated any elements.

The rank of an *array section* is the number of subscript triplets and vector subscripts that it contains, even if they only select 0 or 1 element. Thus, the expression:

```
A (5)
```

is a scalar array element, while:

```
A (5:5)
```

is a section with rank 1 and shape (1). By the rules of Fortran 90, a scalar such as `A(5)` can be used anywhere in an array expression and conforms with any shape, since it is equivalent to broadcasting the scalar value to an array of the appropriate shape. This is not true of the section `A(5:5)`, however.

4.1.2 Expression trees

We shall often refer to the internal representation of an expression within ADAPT, which is in the form of an *expression tree*. This is a binary tree, whose internal nodes can be: an operator, a function or array name, a subscript list, subscript triplet, subscript doublet or subscript expression, or an argument list or expression. Its leaves are simple variables or constants.

Some of the procedures presented in this section perform a *preorder traversal* of the expression tree. This is defined recursively as follows: (i) if a tree consists of one leaf, visit the leaf; (ii) if the tree has root n, with children $n_1, n_2, \ldots n_k$ in order from left to right, visit n, then visit the subtree rooted at n_1 in preorder, then the subtree rooted at n_2 in preorder, and so on [6]. In our case, k is limited to 2 for a binary tree.

Sometimes we perform a *partial* traversal of the tree in preorder, which does not visit the children (if any) of primaries (i.e. it omits subscripts and actual arguments of the given expression). This is accommodated by making the following changes to the above definition: (ii) if the tree has root n *that is an operator*, with children ...; (iii) if the tree has root n *that is a primary*, visit n.

4.2 Target distributions

Before describing how ADAPT transforms expressions and assignments, we must consider the *target distribution* required of expressions and their operands. We define the term 'target distribution' as follows:

- the *target distribution of an operand* is the distribution required of its elements in order that expression evaluation can proceed;
- the *target distribution of an expression* is the distribution required of the elements of the expression result in order that the expression can be used in the required context, e.g. assigned to a variable, or passed as an actual argument to a procedure.

It might seem that an operand should have the same target distribution as the expression in which it appears (and indeed, the first version of ADAPT adopts this scheme for simplicity), but in general this may not be the most efficient way to organise data distribution within expressions, as we discuss later. Hence, we choose to separate these concepts in this discussion.

A distribution is specified by a data structure called a *distribution vector*, which contains most of the information stored in an attribute vector, namely the *rank* and the dimensional parameters $(n, d, \mathit{locn}, \mathit{locl}, \mathit{locu}, \mathit{sectn}, \mathit{sectl}, \mathit{sects})$, as well as a few flags that we shall mention later. It is evident from earlier discussion that this is sufficient to specify the distribution of an arbitrary regular section of an array with any shape and process mapping, which is the most general type of data distribution that we shall need to describe. ADAPT attaches a distribution vector to every node and leaf of an expression tree that has a 'distribution' associated with it, namely, those representing arrays, array sections, array elements, array functions, and operators within an expression with non-trivial distribution.

This subsection describes the target distributions of expressions and operands in various contexts, and gives criteria for determining whether an operand is co-distributed with a target distribution.

4.2.1 Assignment target distributions

We shall start by examining the distribution of data in assignment statements. This investigation will also extend our concepts of data distribution, and provide us with criteria for co-distribution.

First, consider a scalar assignment involving corresponding elements of arrays A and B with identical shape and process mapping (which therefore reside in the same process):

```
A (2) = B (2)
```

An array element is a scalar object, and according to our rules on the distribution of declared data objects, scalars are globally replicated. Naive application of this rule would suggest that the element `B(2)` should be broadcast to a temporary scalar in every process and then written back to `A(2)`, which is obviously unnecessary, as the assignment could take place directly in the process storing the required elements without any need for communication.

This situation demonstrates the need to be able to describe, in the context of array expressions and assignments, the distribution of an arbitrary element or regular section of a distributed array. This can be done using *section parameters*, which were introduced in section 3.3. For our present purposes, it is necessary to extend the Fortran 90 definition of the 'shape' of a regular section (section 4.1.1) as follows:

> The 'shape' of a regular section can include dimensions that are indexed by a single subscript, which are omitted according to the Fortran 90 definition of the term. We use the term '*shape extension*' to describe this modification to the normal Fortran 90 interpretation of 'shape'.

With this extension, the array elements in the above example would be regarded as rank 1 sections with shape (1) rather than, as dictated by the Fortran 90 rules, as scalars. Simple analysis shows these sections have identical distribution, so the assignment can be performed directly without communication.

For the purpose of checking the Fortran 90 rules on conformance, it is useful to distinguish dimensions indexed by a single subscript from *real* sections that select a single element, such as `A(2:2)`. This is done by entering -1 rather than 1 for the shape in the *sectn* vector.

Obviously, sufficient conditions for the co-distribution of a pair of array sections (which may have shape-extension) are:

(i) they have the same mapping of their dimensions onto the process array dimensions, and

(ii) corresponding dimensions have the same mapping of their selected elements onto the process coordinates.

However, these conditions are not necessary. To see this, consider assigning a section of a *globally replicated* array `G` to a section of a *distributed* array `D`, e.g.:

```
D (1:8) = G (3:10)
```

where array `D` has process mapping $(\mu \neq 0)$ and `G` has process mapping (0). Once again the assignment can take place directly in the processes storing the selected elements of `D`, without communication, because the *whole* of array `G` is stored in *every* process, so the required elements of `G` are available where they are needed. Therefore `G(3:10)` can be regarded as being co-distributed with the target data object, `D(1:8)`, in the sense that the selected elements of `G` exist in the same processes as their corresponding target elements, even though `G` has a completely different distribution from `D` (viz. a process mapping of (0) rather than (μ), so that the selected elements of `G` exists in *all* processes whereas those of `D` do not). This conclusion is valid regardless of which elements of `D` and `G` are involved in the assignment (provided the sections conform, of course).

This suggests the following alternative condition for co-distribution:

> An operand dimension with process mapping (0) is co-distributed with a corresponding target dimension with *any* process mapping, regardless of the operand and target section parameters. This is because *all* the elements of the operand are stored in *all* processes in the given process array dimension, including the process(es) storing the corresponding target data elements.

(Of course, this condition only ensures co-distribution with respect to a single dimension; a multi-dimension operand and target must satisfy co-distribution criteria in *every* dimension in order to be co-distributed.)

Our examples so far have involved arrays with the same number of dimensions. Obviously, this is not necessarily the case. Fortran 90 specifies that all *array-valued* objects in an expression or assignment must have the same shape (i.e. *conform*), but the definition of conformance only includes dimensions that are indexed by a subscript triplet or a vector subscript – it ignores dimensions indexed by a single subscript, which just select a single element. Therefore, terms can differ by dimensions indexed by a single subscript, which in our notation are distinguished by having *sectn* $= -1$. We can summarise this situation as follows:

> In comparing an operand distribution with a target distribution, dimensions with *sectn* ≥ 0 must correspond (as required by the Fortran 90 rules on conformance), but the operand and target distribution may differ by dimensions with *sectn* $= -1$, which may occur anywhere in their dimension lists.

We introduce the term '*elemental dimension*' for a dimension indexed by a single subscript, which therefore has *sectn* $= -1$.

As this rule suggests, the correspondence between the dimensions of a pair of array sections is established by first ignoring the elemental dimensions and pairing-up the rest, which must have matching values of *sectn* ≥ 0 for conformance. When this is done, if the first dimension of each section has *sectn* $= -1$ then they are paired. Similarly, if corresponding dimensions of both sections are followed by a dimension with *sectn* $= -1$, then the latter dimensions are also paired. This may be repeated if there is a sequence of dimensions with *sectn* $= -1$. If any such dimensions are left over, then they do *not correspond* to a dimension in the other section.

Our discussions so far allow us to draw the following conclusions regarding co-distribution in the presence of elemental dimensions:

1. If a *target* dimension with *sectn* $= -1$ does not correspond to an operand dimension, then the operand is co-distributed with the target in that dimension, regardless of the target distribution.

2. If an *operand* dimension with *sectn* $= -1$ does not correspond to a target dimension, then it must have process mapping (0) in order to be co-distributed with the target; if not, the relevant element must be broadcast.

3. If corresponding operand and target dimensions have *sectn* $= -1$, co-distribution is determined by the normal criteria, namely, if the target process mapping is (μ), the operand process mapping must be (0) or (μ).

Rules 1 and 2 regarding unmatched elemental dimensions are easily verified by introducing an extra dimension with extent (1) and process mapping (0) in the corresponding position in the dimension list of the opposing array section – which makes no difference to the number or distribution of its elements. There is now a correspondence between this new dimension and the previously unmatched elemental dimension. The stated conclusions readily follow from our previous rules.

To illustrate these rules, consider the following assignment:

```
D (1,2) = E (1)
```

where array `D` has shape (m, n) and process mapping (μ, ν), and `E` has shape (m) and process mapping (μ). Here, the first dimension of the operand and target are co-distributed, but the second dimension of the target does not correspond to a dimension of the operand. In this case, rule 1 above indicates that the operand `E(1)` is co-distributed with the target `D(1,2)`, which is indeed the case. (This is because `D(1,2)` is stored in the process with coordinates (μ_1, ν_1), say, while `E(1)` is stored in all processes with coordinates $(\mu_1, \nu), \forall\nu \in [0, P_\nu - 1]$, that is, the same μ coordinate and *all* ν coordinates. By the same argument as before, `E(1)` is replicated across process grid dimension ν, and so exists at the particular ν coordinate that stores `D(1,2)`.)

If we reverse the assignment:

```
E (1) = D (1,2)
```

then rule 2 suggests that the element `D(1,2)` must be broadcast in process grid dimension ν in order to be co-distributed with `E(1)`, and this is readily seen to be the case.

So far most of the examples have involved scalar assignments for simplicity, but they illustrate rules that apply generally to array-valued assignments. However, scalar operands warrant special consideration in their own right. This is because Fortran 90 states that: "a scalar conforms with any shape, and is equivalent to broadcasting the scalar value to an array of the given shape". We must now consider the implications of this for our scheme.

We introduce the term *scalar object* for a scalar that is a complete object in its own right, rather than a scalar component of a larger object (i.e. an *array element*). Thus, a 'scalar object' is a scalar constant, variable or function reference. According to our rules on data distribution, a scalar object is globally replicated. We have shown above that this implies that it is co-distributed with *any* target distribution, and Fortran 90 states that a scalar object is legal in any array-valued expression and conforms with any shape. This leads to the following conclusion:

> A scalar object (viz. a scalar constant, variable or function reference) conforms with and is co-distributed with *any* target distribution.

An array element is also scalar, of course, but unlike scalar objects it has a distribution in its own right (using the concept of shape extension). If it is an element of a *distributed* array, then it is stored in only a subset of the processes, in which case it is unlikely to be co-distributed with a target distribution unless the target itself is *scalar* (i.e. all of its *sectn* parameters are ± 1), which was the case in the examples above. If the target distribution is *array-valued*, the Fortran 90 rules imply that the array element must be broadcast to all of the processes encompassed by the target distribution. The most straightforward implementation of this is that the array element should be broadcast globally. Hence:

> For an array element to be co-distributed with an *array-valued* target distribution, it must be globally replicated or broadcast.

Although we have not yet stated it explicitly, it is obvious that an expression in an assignment statement takes the distribution of the assignment variable as its target distribution. This is called an **assignment target** distribution. Like all distributions it is specified by a distribution vector, which can describe the distribution of any regular section, array element (using the concept of shape extension) or indeed scalar object (which has the trivial distribution vector with $rank = 0$ and no dimensional parameters!).

There is a slight complication, however, if an operand or assignment variable is an *irregular section*, i.e. if it has vector subscripts in some dimensions. For example:

```
A (VEC_A) = B (VEC_B)
```

How is this handled and how is the distribution described?

Firstly, recall that the extent L of a vector subscript also defines the extent of that dimension of the section, and hence gives the *sectn* parameter of its distribution. Other aspects of the distribution depend on how irregular sections are handled by ADAPT.

If an irregular section is specified in a *distributed* dimension, then in general array elements must be communicated to put them in the required places. This is done using the routines `s$gather()` and `s$scatter()` from the ADLIB library (see section A.2).

If the irregular section is an *operand* (such as `B(VEC_B)` above), it is handled by `s$scatter()`. This takes an input array *in* (`B` in the above example), and copies elements whose subscripts are given by the values of the elements in a vector *vec* (`VEC_B` above) to an output array *out*. The output elements correspond to the subscripts in *vec*, so *out* and *vec* have the same extent.

The interesting point for our present discussion is that `s$scatter()` allows *in* and *out* themselves to be arbitrary *regular sections* of arrays. It is useful to permit *in* to be a regular section, as it may be an assumed-shape array, for example, whose elements are specified by section parameters. It is useful to allow *out* to be a regular section so that it can match a target distribution with section parameters. This means that an assignment such as `A(2:10:2) = B (VEC)` can be performed by a single call to `s$scatter(). s$scatter()` only imposes 2 constraints on the distributions of its arguments: *out* and *vec* must select the same number of elements (by definition) and *in*, *out* and *vec* must all be distributed over same process array dimension, to simplify communications. (Incidentally, if *in* and *out* are multi-dimensional, `s$scatter()` only re-arranges the elements in a single dimension – it does not change the data distribution in other dimensions.)

If the irregular section is an *assignment variable* (such as `A(VEC_A)` above), it is handled by `s$gather()`. This takes an input array *in* (the right-hand side of the assignment), and copies each element to a location in an array *out* (`A` in the above example) whose subscript is specified by the value of an element in a vector *vec* (`VEC_A` in the above example). Each element of *in* corresponds to an element of *vec*, which specifies the subscript of its destination, so *in* and *vec* must have the same extent. Thus, the main difference between this operation and the last is that here *vec* conforms with the relevant dimension of *in* and contains subscripts for the relevant dimension of *out*, and vice versa for `s$scatter()`. Again, the arrays *in* and *out* may be arbitrary regular sections, and similar constraints apply as before (reversing the roles of *in* and *out*!)

From this overview, we can draw the following conclusions regarding the distribution of vector-subscripted dimensions:

> The distribution of a vector-subscripted dimension i of an array has the following parameters specified:
>
> $$\begin{aligned} d_i\ \text{(array-section)} &= d_i\ \text{(parent-array)} \\ sectn_i\ \text{(array-section)} &= sectn\ \text{(vector-subscript)} \end{aligned}$$
>
> The other parameters for that dimension of the array section are arbitrary.

This applies to both operands and assignment variables. Since the distribution of an assignment variable provides the target distribution for the assignment expression, this means that an assignment target distribution may not be fully specified. We will describe later how ADAPT chooses values for arbitrary parameters of a target distribution.

Incidentally, these conclusions also apply to an *internal* vector-subscripted dimension, although no data communications are required and `s$gather()` and `s$scatter()` are not used.

The box below summarises the characteristics of assignment targets, For convenient reference, it also gives formulae for the dimensional parameters of array sections (derived from the formulae in section 3.3).

We defer a summary of the rules concerning operands to a later section, and we also describe later how communications are organised to achieve the required target distribution.

Assignment target distributions:

1. An assignment target distribution is given by the distribution of the assignment variable.

2. The target distribution can have section parameters and shape extension. ('Shape extension' means that it can include array dimensions that are indexed by a single subscript, even though they are omitted from the shape vector according to the Fortran 90 definition of the term. They are distinguished from *real* sections that select a single element by having a *sectn* value of -1 rather than 1.)

Dimensional parameters of array sections:

If an array A has dimensional parameters $(n, d, \mathit{sectn}, \mathit{sectl}, \mathit{sects})$, then section $A(s)$ has dimensional parameters $(n', d', \mathit{sectn}', \mathit{sectl}', \mathit{sects}')$ as in the following table. This table applies to both assignment variables and operands.

type of normalised global subscript, s	dimensional parameters of section $A(s)$				
	n'	d'	sectn'	sectl'	sects'
scalar, i	n	d	-1^*	$\mathit{sectl} + i \star s$	1
triplet, l:u:s	n	d	$\mathrm{MAX}((u-l+s)/s, 0)$	$\mathit{sectl} + l \star s$	$s \star \mathit{sects}$
vector, V	?	d	$\mathit{sectn}(V)$	?	?

* indicates a dimension indexed by a single subscript (included by 'shape extension')

? arbitrary parameter, determined by first array-valued section in expression

4.2.2 Actual argument target distributions

If a procedure with an explicit interface is referenced, the target distribution for each actual argument depends on the specification of the corresponding dummy argument, and in particular on whether its shape and process mapping are assumed or explicit. If no interface is specified, rules must be established to determine a default target distribution. Finally, different rules apply to elemental procedures. We therefore consider each case separately.

Assumed shape dummy argument In section 2.3 we explained that an actual argument that is associated with an *assumed-shape* dummy argument may be a regular section specified by section parameters. In the last section we described how regular section parameterisation is also used for operands and in assignment statements. It is therefore tempting to carry over the concept of shape extension, introduced in the latter context, to the context of actual arguments. This might be used, for example, to allow an array element such as `A(1,2)` to be associated with a scalar dummy argument without the need to re-distribute the array element to a global scalar variable.

However, there are major problems associated with this:

- It would greatly complicate the mechanism for handling assumed-shape dummy arrays, as they could in general have a larger number of dimensions than their declared rank.
- There would also be a large overhead in *referencing* such objects, as the 'extra' distributed dimensions with a single element selected would in general have to be broadcast to make them co-distributed with another operand or an assignment variable that has the declared rank – and which therefore must be global over the 'artificial' dimensions. (However, this does not apply to dummy arguments that are only *defined* by the procedure and not referenced, for the reasons discussed above: a distributed object can be assigned from an object with a 'lower dimensionality' distribution without the need for redistribution.)
- Our rule that scalar dummy arguments are not accompanied by an attribute vector would have to be changed if such arguments could be associated with elements of distributed arrays, as proposed in the above example.

For these reasons, shape extension is *not* allowed for actual arguments. The rank of an actual argument must strictly match that of the corresponding dummy argument, and, although the actual argument may have section parameters (if the dummy is assumed-shape) they cannot include dimensions indexed by single subscript (i.e. *sectn* cannot be -1).

This does not present any problems in standard Fortran 90, which does not have distributed data. If an actual argument is a regular section with subscript triplets (l:u:s) in some dimensions and single scalar subscripts in others, the array can be 're-shaped' across a procedure boundary to 'flatten' the dimensions that reference a single element, leaving only dimensions corresponding to those indexed by subscript triplets. The selected elements of the section can still be expressed as a regular section of the newly-shaped array, so it can be passed into the procedure using section parameterisation without the need for copying. For example, consider a section `A(2,2:4)` of an array with declared dimensions of `(4,4)`. The section has rank 1 and shape (3). As can be seen from Figure 2, this array can be re-shaped as a rank 1 array `AA` with dimension `(16)`, and the required elements form a regular section `AA(6:14:4)` of the re-shaped array, and so can be expressed in terms of section parameters.

However, if the array is distributed in either dimension it *cannot* be reshaped across a procedure boundary, as that would imply data re-distribution.

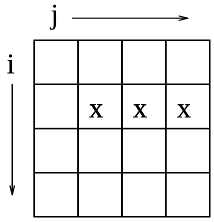

Fig. 2: Regular section `A(2,2:4)` of array declared as `A(4,4)`

Therefore, if a *distributed* dimension of an actual argument array is indexed by a single subscript, it must be broadcast, thereby reducing its rank so that it matches that of the dummy argument. Hence data redistribution may be required even for assumed-shape dummy arguments. The reshaping technique described above can be applied only if the dimension that selects a single element, and all consecutive higher dimensions, up to and including the next dimension that selects a range (if there is one), are undistributed.

Apart from this restriction, an actual argument that corresponds to an *assumed-shape* dummy array may be an arbitrary regular section specified by section parameters. We shall describe later how ADAPT chooses a target distribution for such arrays (see section 4.2.5).

Explicit shape dummy argument If a dummy array has its shape *explicitly declared* rather than assumed, the target distribution for the associated actual argument is a *whole, contiguous* array with the declared shape, rather than a regular section (see section 2.3). The section parameters in this case have the form

$(n{=}\mathit{sectn}{=}N, \mathit{sectl}{=}0, \mathit{sects}{=}1)$.

If the actual argument is a regular section, the transformation scheme copies it into a full array (which may involve communication).

Assumed and explicit distribution dummy argument The *process mapping* of a dummy array may also be either assumed or explicit (or indeed it may not be declared at all, which is equivalent to an explicit declaration that the array is *not* distributed). The type of process mapping, assumed or explicit, is entirely independent of the type of shape specification.

If the process mapping is *assumed*, the associated actual argument can have any process mapping. We shall describe later how ADAPT chooses the target process mapping in this case (in section 4.2.5). If the process mapping is *explicit*, that provides the target process mapping for the actual argument, which is redistributed if necessary to achieve it.

No explicit interface If the calling procedure does not contain an explicit interface for the called procedure, all actual arguments have arbitrary target rank, shape and process mapping, but section parameters and shape extension are not allowed. If the actual argument is array-valued, its target distribution is a contiguous array; if it is scalar, its target is a global scalar. Thus, the same types of target distribution are permitted as for explicit-shape dummy arrays, except that the rank and shape are arbitrary!

This type of target distribution is called an *arbitrary contiguous* target distribution. It is described more fully in section 4.2.4, while section 4.2.5 describes how ADAPT selects a target distribution in this case.

Elemental procedures The discussion so far has implicitly concerned *non-elemental* procedures, which include all user-defined procedures.

However, Fortran 90 also introduces *elemental* procedures, whose application is currently restricted to a subset of the Fortran 90 intrinsic procedures. An elemental procedure is one that is defined for scalar arguments but may also but may also be applied to conforming array arguments. In the case of a function, each element of the result, if any, is as would have been obtained by applying the function to corresponding elements of the arguments. We shall see later that ADAPT transforms an array assignment into an equivalent elemental assignment enclosed in DO-loops over the array subscripts, so that an elemental function reference with array arguments can be transformed into one with array element arguments in the context of the elemental assignment.

For this to work, the array arguments must be co-distributed with the other terms in the expression, that is, their target distribution must be the same as that of an operand in the expression (see section 4.2.6).

Summary We can therefore identify several types of target distribution for an actual argument, depending on the specification of the dummy argument. They are summarised in the box below.

Actual argument target distributions:

Explicit procedure interface:

dummy shape	$(n, \mathit{sectn}, \mathit{sectl}, \mathit{sects})$ parameters of target distribution
assumed (:)	(?,?,?,?) Any shape (of the correct rank) is allowed, as are section parameters, but not shape extension (i.e. $\mathit{sectn} \neq -1$).
explicit (N)	$(N, N, 0, 1)$ Shape must conform. Actual argument must be a full array.
dummy process mapping	(d) parameter of target distribution
assumed (:)	(?). Any target process mapping allowed.
explicit (D)	(D). Target process mapping is specified.

No explicit interface:

Each argument has *arbitrary contiguous* target distribution (i.e. its target is a contiguous array with arbitrary shape and process mapping – see section 4.2.4).

Elemental function:

All arguments have the same target distribution as operands of the expression containing the elemental function reference (see section 4.2.6).

If an actual argument is a variable that is modified by the procedure, and it was copied to a temporary variable before being passed-in (in order to achieve a required target distribution), then it is copied-back on return. In this case, the copy-back is as for an assignment statement, and the original actual argument provides the target distribution.

4.2.3 Scalar and subscript target distributions

General array expressions only appear in a limited number of contexts in Fortran 90, namely: in assignments, output items, actual arguments, mask expressions in `WHERE` statements and constructs, and array and structure constructors.

In all other contexts, except for subscripts, expressions are required to be scalar. Scalar expressions are often used for controlling the program flow: examples are DO-loop and implied DO-loop control expressions, and expressions in `IF`, `DO WHILE`, `CASE`, computed `GOTO`, and arithmetic `IF` statements, etc.; other uses in executable statements include I/O specification. In all these cases the expression result is required globally by every process of the output program. Therefore, the target distribution of a scalar expression is a *global scalar*, that is, its result is broadcast globally.

Subscript expressions can be either scalar or vector. For the reasons given above, the result of a *scalar* subscript expression is also broadcast globally. There are two obvious choices for the target distribution of a *vector* subscript expression: it can be globally broadcast, or distributed over the same process array dimension as the data dimension that it subscripts. Any other distribution would greatly complicate the gather-scatter process used to re-arrange the elements into a regular section (see section 4.2.1). The second alternative – distributed rather than global – is chosen to reduce the use of temporary intermediate storage.

These rules are summarised in the box below.

<u>Scalar and subscript target distributions:</u>

1. The result of a *scalar* expression (viz. an expression that must be scalar by virtue of its context, including a scalar subscript expression) is a *global scalar*.
2. The result of a *vector subscript* expression is a full, rank-1 array (not a regular section), distributed over the same process array dimension as the data dimension that it subscripts.

4.2.4 Arbitrary target distributions

In some contexts there is *no* pre-defined target distribution for an array expression. Even its rank is arbitrary *a priori*. Such expressions are said to have *arbitrary* target distributions. This class falls into two subdivisions:

- Expressions whose results must occupy *contiguous* storage, i.e. they cannot be arbitrary regular sections described by section parameters, and cannot employ shape extension. This is the case for an actual argument expression in the absence of an explicit procedure interface (so that the specification of the corresponding dummy argument is unknown). This type of expression is said to have an **arbitrary contiguous** target distribution.
- Expressions whose results may be described by sectional parameters with shape extension. An example is an output item (i.e. an item in the output list of a `WRITE` statement, etc.). This type of expression has an **arbitrary sectional** target distribution.

The box below summarises the characteristics of arbitrary distributions.

Arbitrary target distributions:

An *arbitrary* target distribution can have any rank. An *arbitrary contiguous* target distribution cannot have sectional parameters or shape extension, while an *arbitrary sectional* one can.

The next section describes how ADAPT chooses a suitable target distribution in these and other cases.

4.2.5 Determining unknown target distributions

In addition to the arbitrary target distributions introduced above, we have seen other contexts where part or all of an expression's target distribution is undetermined *a priori*.

One example is a dimension of an assignment target distribution that corresponds to an irregular section in the assignment variable, which only has its d and *sectn* parameters pre-defined. Another is an actual argument associated with an *assumed* shape and/or distribution dummy argument. Unlike arbitrary distributions, its *rank* is known in advance. As we have already seen, this class comprises three subdivisions:

- *Assumed-shape and distribution*: The expression can have any shape and process mapping (of the specified rank). Sectional parameters are allowed but not shape extension.
- *Assumed-shape, explicit distribution*: The expression can have any shape (of the specified rank), but its process mapping is specified. Again, sectional parameters are allowed but not shape extension.
- *Explicit shape, assumed distribution*: The expression can have any process mapping but its shape is specified. The expression result must occupy contiguous storage – neither sectional parameters nor shape extension are allowed.

These various cases differ in the particular target parameters that are undetermined, and in details such as whether sectional parameters and shape extension are allowed. In this section we describe how ADAPT selects a suitable target distribution in these cases.

Since the distribution of the expression result is (to some extent) arbitrary, it is sensible to choose one that minimises communications and the use of temporary storage. In general, however, finding an optimal target distribution requires a considerable amount of analysis, as the distribution of every term in the expression must be determined and compared. We shall see later that the operands in an expression are processed as they are encountered during a pre-order traversal of the expression tree. For convenience, ADAPT sets any undetermined parameters of the target distribution at the earliest possible stage during this pre-order traversal, which in practice means when the first operand with non-trivial distribution is encountered. Obviously this is optimal if the expression contains only one operand, but it may not be a good choice generally, as it does not take account of the distribution of subsequent terms in the expression tree.

Firstly, if the target *shape* (i.e. its *sectn* parameters) is unknown, it is determined by the first *array-valued* operand encountered in the pre-order traversal of the expression tree. Array elements are ignored for this purpose, as they can appear in any array-valued expression and do not determine its shape. Operand dimensions with $sectn = -1$ are also ignored, as the shape vector is only required for conformance checking at this stage. If the *sectn* parameters are still undetermined when the complete expression tree has been processed, then the expression is known to be scalar.

Let us now assume, for the duration of this paragraph, that the expression is array-valued. Then, if the target *process mapping* (i.e. its d parameters) is unknown, it is set equal to that of the first *distributed, array-valued* operand encountered in the expression tree traversal. Note that for this purpose we ignore array *elements*, as they cannot determine the distribution of an array-valued expression – indeed, they must be globally broadcast. We also ignore operands that are *not* distributed. As we have seen, such operands are co-distributed with *any* target distribution, so it is not necessary to know the final target distribution in order to process them – they need not be redistributed whatever the target distribution. If a subsequent *distributed* operand is encountered, it is better to let *it* determine the target process mapping, for then it may not require redistribution in order to be co-distributed with the target distribution, as it would if the target distribution were global (though redistribution may still be necessary if section parameters are not allowed in the target distribution). Another reason for adopting this approach is that it is always preferable to *distribute* array expressions rather than replicate them, because it increases the potential data parallelism and reduces the amount of storage allocation required for the expression. After all, the whole purpose of this system is to evaluate array expressions in a data-parallel manner!

We must now consider how to handle *distributed array elements* (i.e. elements of distributed arrays). If a distributed array element appears in an *array-valued expression*, it must be globally broadcast, in accordance with the rule in section 4.2.1. It it appears in a *scalar expression* with an arbitrary *contiguous* target distribution, then again it must be globally broadcast because shape extension is not allowed.

The situation is more complex if the target distribution is allowed to have shape extension (viz., if it has an *arbitrary sectional* target distribution). In that case a *scalar expression* may have a non-trivial distribution, because shape extension is allowed. Therefore, if a distributed array element is encountered before any array-valued operand, the appropriate course of action is uncertain: if the whole expression is scalar then the array element itself should determine the expression's target distribution, but if a subsequent operand is array-valued then all array elements must be broadcast (in accordance with the rule given in section 4.2.1), and the array-valued operand should determine the target distribution.

For example, if `A` is a vector with process mapping (μ), then the expression:

```
A(4) + A
```

is legal and its natural target distribution is the pre-exisiting distribution of `A`, but `A(4)` must be broadcast in order to be co-distributed with the expression. We cannot take the target distribution to be that of the element `A(4)` (namely a distributed section of length 1 specified by section parameters) although it is encountered first when the expression is parsed. On the other hand, the latter would be the appropriate target distribution if the expression was simply `A(4)`.

One solution to this problem is to require that all array elements appearing in expressions with arbitrary target distribution are broadcast, but that is inefficient. For example, if the element `A(4)` appears as an output item in a `WRITE` statement, it can be output without changing its distribution – it is unnecessary and inefficient to broadcast it!

A better solution is to determine whether the whole expression is scalar or array-valued *before* 'processing' any of its terms (i.e. before any subscripts or actual arguments are evaluated, or any output code is generated). This can be done using the procedure summarised in Figure 3, and is the solution adopted in ADAPT.

Procedure for determining unknown target distributions We shall now summarise the procedure *determine_target_distr* (*target_distr*, *array_sect*) that is used by ADAPT to determine an unknown or partly known target distribution, *target_distr*. As we said above, this procedure is invoked when each successive array operand, *array_sect*, of an expression tree is processed, and it uses the distribution of *array_sect* to set any undetermined parameters of *target_distr*, provided *array_sect* is suitable for this purpose. The operands are processed as they are encountered during a pre-order traversal of the expression tree.

Before the procedure is invoked, flags are set in *target_distr* to describe certain characteristics of the target distribution, such as whether it is scalar or array-valued, contiguous or sectional, and whether shape extension is allowed. In the following description we use unprimed names for distribution parameters of *array_sect*, and primed names for parameters of *target_distr*.

Scalar expressions For a scalar expression, an arbitrary *target_distr* is determined as follows:

```
function expr_is_scalar (expr_tree) : bool
    (* This function returns the value true if expr_tree is scalar *)

    for each primary p visited in pre-order do
        if p = array name then
            if p has no subscript list then
                return false
            else
                for each subscript s of p do
                    if s = subscript doublet or triplet or not expr_is_scalar (s) then
                        return false
        else if p = function name then
            if p is array-valued then
                return false
            else if p is elemental then
                for each actual argument a of p do
                    if not expr_is_scalar (a) then
                        return false

    return true    (* expr_tree is scalar *)
```

Fig. 3: Function to determine if an expression is scalar

- If *shape extension* is allowed, *target_distr* is taken to be the distribution of the first *array_sect* that is an element of a *distributed* array.
- Otherwise, *target_distr* is a global scalar target distribution.

Array expressions – target shape The target shape (i.e. *sectn′* vector) is taken to be that of the first *array_sect* that is *array-valued*, omitting dimensions with $sectn = -1$ (as the shape is only needed for conformance checking at this stage). If a particular *rank* is expected (e.g. for assumed-shape arrays) an error is reported if it is incorrect.

If *target_distr* must be *contiguous*, then (n'=*sectn′*, *sectl′*=0, *sects′*=1) in every dimension.

If *target_distr* corresponds to an *assumed-shape*, *explicit distribution* dummy array, then the target process mapping (i.e. d' parameters) are already known. In that case, *target_distr* is completely determined once the shape is known, as follows:

For each dimension i:

- If $(d' = d)$ all parameters are copied from *array_sect*. This has the advantage that it avoids the possibility of having to redistribute *array_sect* to match *target_distr*, at least as far as dimension i is concerned.

- Otherwise a 'contiguous' distribution is chosen, i.e.: (n'=*sectn'*=*sectn*, d'=d, *sectl'*=0, *sects'*=1) and the *loc_* parameters are determined as usual (section 3.2). In this case redistribution is needed anyway (unless $d = 0$), so *target_distr* is chosen to distribute the data as uniformly as possible over the process array dimension in order to maximise the potential data parallelism and minimise storage allocation.

Array expressions – target process mapping The target process mapping (i.e. d' vector) is taken to be that of the first *array_sect* that is *array-valued* and *distributed*.

At this point, the target shape (i.e. *sectn* vector) is already known, possibly omitting dimensions with *sectn* $= -1$ (either it is known explicitly – for an explicit-shape, assumed distribution array – or it was determined as above). The remaining target parameters are set as follows:

- *Contiguous* target distributions: (n', *sectn'*, *sectl'*, *sects'*) are already known; it just remains to copy d and calculate the *loc_* parameters.
- *Sectional* target distributions: the whole target distribution is copied from *array_sect*, including dimensions with *sectn* $= -1$ if shape extension is allowed. It is possible that the section parameters of *array_sect* itself are undetermined in some dimensions, namely those with vector subscripts (see section 4.2.1). In that case, *target_distr* is chosen to have contiguous distribution in those dimensions, namely (n'=*sectn'*=*sectn*, d'=d, *sectl'*=0, *sects'*=1).

4.2.6 Operand target distribution and co-distribution criteria

We have not yet considered the target distribution of individual operands in an expression – only the distribution of the expression result.

Naively, it might appear that an operand should have the same target distribution as the expression result. This means that all operands are re-distributed (if necessary) to have the required distribution of the expression result *before* the expression is evaluated, whereupon the result is available where it is needed without further redistribution. This is known as the 'owner computes' rule, and it is usually employed by automatic parallel systems.

However, this is not always the most efficient scheme. For example, consider the following array assignment involving vectors with identical extent and distribution:

```
A (0:N−1) = B (1:N) + C (1:N)
```

Assuming that these sections are distributed over more than one process, the 'owner computes' rule forces both `B` and `C` to be shifted by one element (involving communications) into temporary vectors, whereupon the expression is evaluated and assigned to `A`. A better solution is to evaluate the expression *in situ* and shift the result into `A` – involving just one temporary vector and one communication versus two previously.

For simplicity, however, the first version of ADAPT will employ the 'owner computes' rule. Relaxing this rule will be a priority in future research.

The following box summarises the characteristics of operand distributions, and gives criteria for the co-distribution of an operand with a target (most of which were derived in section 4.2.1).

Operand target distributions:

1. An operand distribution can have section parameters and shape extension.
2. Operands take their target distribution to be that of the expression result, if one is defined; otherwise it is arbitrary.

Co-distribution criteria:

1. The following conditions are sufficient, but not necessary, for an operand and target to be co-distributed:
   - (i) they have the same mapping of their dimensions onto the process array dimensions;
   - (ii) corresponding dimensions have the same mapping of their selected elements onto the process coordinates.
2. An operand dimension whose process mapping is (0) (an internal dimension) is co-distributed with a target dimension with *any* distribution.
3. A scalar object (viz. a scalar constant, variable or function reference) conforms with and is co-distributed with *any* target distribution. For an *array element* to be co-distributed with an *array-valued* target distribution, it must be globally replicated or broadcast.
4. Operand and target distributions may differ by dimensions with $sectn = -1$.

   Co-distribution of an operand with a target distribution is unaffected by the target having extra dimensions with $sectn = -1$. If the *operand* has extra dimensions with $sectn = -1$, then they must have process mapping (0) or their selected elements must be broadcast in order to achieve co-distribution. If corresponding operand and target dimensions have $sectn = -1$, co-distribution is determined by the normal criteria, namely, if the target process mapping is (μ), the operand process mapping must be (0) or (μ).

4.3 Processing an expression

As we mentioned earlier, ADAPT constructs an expression tree as it parses an expression. As the expression tree is built, certain attributes are derived and stored at each leaf and node. Among these are:

- the expression's data type;
- flags indicating whether the expression is:
  - constant (i.e. involving only literal and named constants);
  - 'simple scalar' (i.e. involving only *simple* scalar data objects, *not* array elements or function references);

Obviously, type checking (viz. checking whether the operands of an operator have an appropriate type) is also performed as the expression tree is constructed.

An expression has the '*constant*' flag set if it involves only literal and named constants. ADAPT evaluates *integer* constant expressions as they are parsed, using the declared values of integer named constants (i.e. `PARAMETER`s), and replaces the appropriate node of the expression tree by a leaf containing the integer literal result. This has the disadvantage that integer named constants used in the input program do not appear in the output program, but it has the considerable advantage of allowing static analysis to be performed in the presence of named constants, so that no performance penalty is incurred through the good practice of using them.

All other processing of an expression is done after the complete expression tree (*expr_tree*) has been constructed, using the procedure *process_expr*() which is outlined[5] in Figure 4. By 'processing' an expression we mean generating code in the output program to evaluate any non-elemental function references that it contains and to perform all necessary data redistribution, and, if the expression is in an assignment statement or must be assigned to a temporary variable, to actually perform the assignment.

The procedure *process_expr*() can 'process' an expression either by itself or in an assignment, provided the assignment variable does not contain *distributed*, *irregular sections*. The latter case is handled by introducing a temporary array to which the expression is assigned, using *process_expr*() to generate code for the expression evaluation and assignment, and then generating code to call the ADLIB routine `s$gather()` to re-arrange the temporary array into the irregular section.

Apart from *expr_tree*, the other inputs to this procedure are the target distribution for the expression result, *target_distr*, which is determined as

[5] All of the procedures illustrated in this section are presented in a very informal style, loosely based on a Pascal-like syntax, but with declarations, pointers, data structures, `begin-end` delimiters, etc, omitted, and with code frequently replaced by English language descriptions.

described in the last section, and the assignment variable, *assign_var*, if there is one. The *target_distr* may be partly or totally unknown at the start of this procedure (e.g. because the expression has arbitrary shape and distribution, or corresponds to an assumed shape and/or distribution dummy argument). In that case, it may be determined by the procedure itself and thus be an output parameter. Also, the procedure may introduce a temporary assignment variable for the result of an expression that does *not* appear in the context of an assignment statement, in which case *assign_var* is an output parameter of the procedure.

No processing is necessary for an expression that has the '*simple_scalar*' flag set. This flag indicates that all of the terms in the expression are 'simple' scalar constants and variables (not array elements or function references). As we have seen, such an expression conforms with and is co-distributed with any target distribution. Furthermore, there are no embedded subscript or actual argument expressions to be 'processed'. Therefore the expression can be output straight away; hence the test at the start of *process_expr*() that branches directly to the assignment stage.

For processing more general expressions, the following stages are involved:

1. *Determine whether storage must be allocated for the expression result*.

   An expression *must* be assigned to a variable if it involves operators *and* is array-valued. This is because Fortran 77 does not support array expressions, so an array expression with operators cannot appear in the output program. If the expression does not involve operators, then it is already a simple variable (possibly a temporary one introduced as a consequence of re-distribution, etc.). Scalar expressions need not be assigned to a variable as they can appear anywhere in the output Fortran 77 program where they are legal in the input one. (There is, however, the question of whether a *distributed* scalar expression, which is only defined in a subset of the processes, should be assigned to a variable within an IF construct in order to restrict its evaluation to processes in which the result is defined. Careful consideration shows that this is unnecessary).

   If these criteria show that an expression must be assigned, and it is not already in an assignment statement, then *must_alloc_result* is set to `true` to indicate that temporary storage must be allocated for the expression result, *assign_var*. If so, then *assign_var* is allocated if the target distribution is known.

   If *target_distr* is initially undefined or incompletely defined, then *assign_var* is added to the end of an 'allocation queue'. As soon as *target_distr* is determined, storage is allocated for all the objects on this queue whose required distribution is *target_distr*, in the order that they appear on the queue. This helps to ensure that data objects are allocated in the optimal order.

```
procedure process_expr (expr_tree, target_distr, assign_var)
     var must_alloc_result : bool

     (* root is the root node of expr_tree. *)

     if expr_tree is 'simple-scalar' then goto 4.

1.   must_alloc_result := no assign_var and root is an operator and
                                          target_distr is array-valued
     if must_alloc_result and target_distr is known then
          alloc (assign_var, target_distr)
          must_alloc_result := false
     else
          add assign_var to the allocation queue

2.   for each primary p visited in pre-order do
2a.       if p = function name then
               process_func_ref (p, target_distr)

2b.       else if p = array name then
               process_array_section (p, target_distr)

     (* Now each term is a simple scalar, an elemental function reference *)
     (* or an array section with the required distribution.              *)

3.   if must_alloc_result then
          alloc_global_object (assign_var, target_shape)     (* globally replicated *)
          must_alloc_result := false

4.   if there is an assign_var then
          process_assignment (assign_var, expr_tree)
          dealloc (expr_tree)
          replace_expr_node (expr_tree, assign_var)
```

Fig. 4: Procedure for expression transformation

2. *Perform a pre-order traversal of the expression tree, visiting only* primaries[6] p.

   2a. *Process function primaries using 'process_func_ref()'.*

   This procedure is described in section 4.4. Briefly, for a non-elemental function it generates code to execute the function reference and, if necessary, redistribute its result so that it is co-distributed with *target_distr*, while for an elemental function it generates code, if necessary, to evaluate its actual arguments with *target_distr* as their target distribution. It also updates the expression tree appropriately: a non-elemental function reference has its subtree replaced by a leaf representing the function result, while for an elemental function reference, the nodes corresponding to its actual arguments are updated if they were evaluated and/or re-distributed.

   2b. *Process array section primaries using 'process_array_section()'.*

   This procedure is described in section 4.5. Briefly, it generates code to evaluate the section's subscripts and section parameters, perform range and conformance checks, re-arrange an irregular section into a regular one, and re-distribute the section if necessary to match *target_distr*. If any redistribution is performed, then on return the array section is replaced on the expression tree by the result of the redistribution.

   Either of the above procedures may set *target_distr* if it was initially not fully defined. If necessary, storage is allocated for *assign_var* as soon as *target_distr* has been fully determined.

   At the end of this stage, the expression comprises only elemental function references, simple scalars, and regular array sections[7], all of which are co-distributed with the target distribution.

3. *If the expression result is still not allocated, allocate it with global distribution.*

   Having processed the whole expression tree, it is possible that *assign_var* is still not allocated because *target_distr* is still not completely determined. The *shape* of the expression (i.e. the *sectn* vector of *target_distr*) must have been established by now if it was initially arbitrary. However, its *process mapping* (i.e. the d vector) may still be undefined. This happens if every term in the expression is globally replicated, since the process mapping is derived from the first distributed term.

[6] Recall that the word *primary* only includes the 'top-level' data objects and function references of an expression, that is, it excludes terms that appear in subscripts or actual arguments of the given expression.

[7] For the rest of this section, we use *array section* as a generic term for a whole array, an array section or an array element.

In that case, storage is allocated for *assign_var* according to a default distribution, namely it is globally replicated, has the shape given by *target_distr* and occupies contiguous storage. This is a suitable target distribution for this expression. Notice, incidentally, that no storage has been allocated for data redistribution because all the terms in the expression are globally replicated, so the optimal storage allocation order has not been upset.

It is tempting to set *target_distr* to this default distribution. This is not done, however, to allow for the possibility that *target_distr* might also apply to subsequent expressions which may have non-trivial distributions, in which case *they* can set the target process mapping. (This may be the case, for example, when processing an expression containing elemental function references. Then *process_expr*() processes all the elemental function arguments, each of which has the same *target_distr* as the parent expression.)

4. *If the expression must be assigned, generate code to perform the assignment*.

   An array-valued assignment must be transformed into an equivalent elemental one enclosed in DO-loops over the array subscripts, because Fortran 77 does not support array operations! Note also that the subscript range is restricted to the locally-stored segment of the assignment variable. This is described in section 4.7.

   In making this transformation, elemental function references with array arguments become references with scalar arguments, namely the corresponding array elements.

   This stage also replaces the *expr_tree* by a node representing the expression result, *assign_var*.

Note that the *process_expr*() procedure is invoked recursively to evaluate actual arguments and subscripts of the given expression, via the procedures *process_func_ref*() and *process_array_section*() respectively.

4.4 Processing a function reference

Figure 5 outlines the procedure used by ADAPT for 'processing' function references. It is invoked by *process_expr*() as shown in the last section. Briefly, for a non-elemental function it generates code to execute the function reference and return a result that is co-distributed with *target_distr*, while for an elemental function it merely generates code, if necessary, to evaluate the actual arguments with *target_distr* as their target distribution. It also updates the expression tree appropriately: a non-elemental function reference has its subtree replaced by a leaf representing the function result, while for an elemental function reference, the nodes corresponding to its actual arguments are updated if they were evaluated and/or re-distributed.

```
procedure process_func_ref (func, target_distr)
    (* func is the root node of an expression tree *)

    if interface of func is explicit or func is elemental then
1.      for each actual argument act_i do
            associate act_i with corresponding dummy argument dum_i in the symbol table
        for each dummy argument dum_i do
            check that dum_i is correctly associated with an act_i

2.      if func is array-valued then alloc (func_result, distr (func))

3.      if func is elemental then
            for each actual argument act_i do process_expr (act_i, target_distr)
            return
        else
            for each actual argument act_i do process_expr (act_i, distr = distr (dum_i))

4.      if func is array-valued then
            write ("CALL func_name (func_result, attr_result,")
        else
            write ("r = func_name (")
        for each dummy argument dum_i do
            write (act_i and any associated extra arguments (e.g. attribute vectors))
        write (")")

5.      for each dummy argument dum_i do
            if act_i = temp_i (* a temporary array *) then
                if act_i was a variable and dum_i is not INTENT(IN) then
                    copy_back (act_i ← temp_i)
                de_alloc (temp_i)

    else    (* Implicit interface *)
3'.     for each actual argument act_i do process_expr (act_i, distr = arbitrary contiguous)

4'.     write ("r = func_name (")
        for each actual argument act_i do
            write (act_i and any associated extra arguments (e.g. attribute vectors))
        write (")")

5'.     for each actual argument act_i do
            if act_i = temp_i (* a temporary array *) then
                if act_i was a variable then copy_back (act_i ← temp_i)
                de_alloc (temp_i)

6.  (* Redistribute func_result if necessary to match target_distr *)
    if func_result is array-valued then process_array_section (func_result, target_distr)

7.  replace_expr_node (func, func_result)
```

Fig. 5: Procedure for transforming a function reference

If the function is elemental or its interface is known, *process_func_ref*() performs the following actions:

1. Each actual argument is associated with the corresponding dummy argument in the symbol table. (Dummy arguments and their attributes are installed in the symbol table when the function interface specification is parsed, or are 'built-in' for intrinsic functions). Actual arguments need not appear in the same order as the corresponding dummy arguments, as the latter can be referred-to by name in the actual argument list and therefore specified in any order.

   After the complete actual argument list has been processed, the dummy list is scanned to check (i) type agreement of the corresponding actual arguments (shape agreement is checked in step 3); (ii) that a dummy argument with intent `OUT` or `INOUT` is associated with a definable actual argument, and (iii) that any dummy argument that is *not* associated with an actual argument has the `OPTIONAL` attribute.

2. If the function is array-valued, storage is allocated for the local segment of its result. It is optimal to allocate storage for the function result before allocating any for the evaluation of actual arguments, as the function result outlives its arguments.

   However, the shape and/or process mapping of the function result may depend on the values or attributes of dummy arguments, which may require the corresponding actual arguments to be 'processed' (in the sense given in the last section) before the function result can be allocated. If any such argument is array-valued and its result is assigned to a temporary array, this causes a non-optimal ordering of storage allocation (i.e. the temporary actual argument array cannot be de-allocated on return from the function, as the function result was allocated *after* it, and hence above it on the stack, and must be preserved).

   This situation does not arise if the specification expressions for the shape and distribution of the function result do not involve dummy arguments, or they involve only:

   - dummy arguments that are associated with constant integer expressions or simple scalar expressions (see section 4.3);
   - *attributes* of dummy arguments that can be determined without processing or evaluating the corresponding actual arguments (e.g. the size or distribution of a dummy array that is associated with a *whole* array variable, rather than an array section or expression).

   If these conditions are not satisfied, then some actual argument expressions must be evaluated before storage can be allocated for the function result.

Note that this allocation problem is inherent in Fortran 90 itself. There does not seem to be any restriction in the Fortran standard to prevent the specification of a function result from involving elements of dummy argument arrays that are associated with actual argument expressions. This complication has nothing to do with distributed data – indeed, Distributed Fortran 90 forbids the use of distributed data in specification expressions.

3. Now *process_expr*() is called for each actual argument, to generate code (if necessary) to evaluate it. The target distribution for each argument is determined by the rules given in section 4.2.2. If the function is elemental, the target distribution for each argument is given by *target_distr*. If the latter contains any undetermined parameters, they may be initialised by *process_expr*().

At this point processing is complete for an elemental function reference. As we described in the last section, the procedure *process_expr*() transforms an array assignment into an equivalent elemental assignment enclosed in DO-loops. The expression in this assignment can include elemental function references, with array arguments replaced by the corresponding array *element* arguments.

Other functions do not have this property. A single function reference corresponds to a single function invocation, even if it occurs in an array expression. Thus it is necessary to generate code for a *non*-elemental function reference *before* generating code for the expression evaluation itself. This is done in the next step.

4. Now the function call is written to the output program. Note that an array-valued function reference is converted to a subroutine call whose first argument is the result, as Fortran 77 does not allow array-valued functions. Also, the actual arguments are re-ordered if necessary to correspond to their dummy arguments, and are accompanied where necessary by extra arguments (e.g. an attribute vector for an array argument, a boolean value to indicate the presence or absence of an optional argument, etc.).

5. For each *temporary* array argument, code is output to:

   (i) Copy it back if necessary into the actual argument array. This is only necessary if the actual argument is a variable that was copied into a temporary array for the purpose of re-distribution or to make a regular section contiguous. (Note also that an irregular section is *not* regarded as a variable in this context, according to the Fortran 90 rules).

   (ii) De-allocate the temporary array.

6. If necessary, the function result is re-distributed to match the target distribution of the expression, *target_distr*, using the procedure *process_array_section*() described in the next section. This procedure may initialise any undetermined parameters in *target_distr*.

7. Finally, in ADAPT's internal representation of the expression tree, the subtree corresponding to the function reference (together with its arguments) is replaced by a node representing the result.

If an explicit interface is not given for the function, this transformation procedure is simplified in several respects. For example: the function result must be scalar; optional arguments, argument re-ordering and argument usage checking are not supported; and all dummy argument arrays can have arbitrary rank, shape and process mapping (although array sections must be copied to contiguous arrays).

4.5 Processing an array section, and data redistribution

This section describes the procedure *process_array_section* (*array_sect*, *target_distr*), which is used to process arrays, array sections and array elements (all of which are hereafter included in the generic term 'array section'). It is invoked both by *process_expr*(), to process array sections appearing in an expression, and by *process_func_ref*() to process function arguments and results.

Briefly, 'processing' an array section involves: (i) evaluating its subscripts and section parameters, and performing range and conformance checks; (ii) re-arranging an irregular section into a regular one; and (iii) re-distributing the section if necessary so that it has the required *target_distr*.

This procedure is responsible for inserting all of the necessary data communications in the output program. Also, if any parameters of the expression's target distribution, *target_distr*, are undefined, they may be determined herein.

The steps in this procedure are as follows:

1. The subscripts are transformed from *original* to *normalised global* values by subtracting their declared lower bounds, and evaluated using *process_expr*(). Both scalar and vector subscripts are evaluated. Their target distributions are determined as described in section 4.2.3. If necessary, the function *expr_is_scalar*() (see Figure 3) is used to determine whether a subscript is scalar or vector, so the target distribution can be initialised accordingly.

2. Range and conformance checks are performed, and the section parameters are derived, as follows. (In the following description, primed names refer to parameters of the *section* and unprimed names to parameters of the *parent array*.)

(i) Range checking is performed for scalar subscripts and subscript triplets, by comparing their (normalised global) values with the array's *sectn* vector. (Vector subscripts are checked later).

(ii) The section parameters of the array section, $(sectn', sectl', sects')$, are calculated using the formulae of section 3.3.

Recall that $sectn'$ is the *shape* of the section. The *extended* shape is derived, that is, it includes dimensions that are indexed by a single subscript, which would be omitted according to the Fortran 90 definition of 'shape' (see section 4.2.6). Such dimensions have an extent of -1 recorded in the $sectn'$ vector, to distinguish them from *real* sections with a single element selected (e.g. `A(2:2)`).

The interpretation of $sectl'(i)$ and $sects'(i)$ for dimension i depends on the type of section referenced in that dimension. For a regular section (which may be the whole dimension), they are the lower bound and stride in terms of *real global* subscripts. For a single element, $sectl'(i)$ is its *real global* subscript and $sects'(i)$ is meaningless. For an irregular section (i.e. a dimension with a vector subscript) they are the lower bound and stride of the regular section that is the result of the scatter operation, which are arbitrary.

(iii) A conformance check is performed, by comparing the section's $sectn'$ vector with that in *target_distr*, if it is defined. For this purpose, dimensions i with $sectn'(i) = -1$ are ignored in both the source and target shape vectors. If every dimension has $sectn'(i) = -1$, then this is an array element and so conforms with any target shape.

(iv) If any parameters of *target_distr* are undefined, the procedure described in section 4.2.5 is invoked. It uses the distribution of the array section to initialise the target distribution, provided the array section is suitable for that purpose.

If this procedure *does* determine the target distribution, then code is generated in the output program to allocate storage for any objects in the 'allocation queue' whose distribution is given by *target_distr*. (Objects may be placed in this queue in step 1 of the procedure *process_expr*() – see section 4.3).

If the scalar subscripts, array bounds and target shape are all constant, these actions are performed statically by ADAPT, and the resulting section parameters are stored internally in a *distr* structure that is attached to the relevant node of the expression tree. Error messages are given if any range or conformance errors are detected.

Otherwise, code is output to call a runtime library routine which performs the necessary actions:

```
CALL s$section (target_attr, sect_attr, array_attr, &
                subscr_list, array_name, lineno)
```

The subscript list is passed to `s$section()` as a vector input argument *subscr_list*, which is allocated and initialised before the call. For each array dimension, it contains an integer tag *count*, followed by scalar subscript values as shown in the following table:

count	accompanying subscripts (global normalised)
≤ 0	none (vector subscript of length $(-count)$)
1	single subscript
2	subscript doublet (l:u)
3	subscript triplet (l:u:s)
4	none (denotes a whole dimension)

All subscripts are global normalised values, as derived in stage 2. The *subscr_list* is de-allocated immediately after the call.

The other input parameters to this routine are the attribute vector of the parent array (*array_attr*), and the array name and input program line number for use in error messages (*array_name* and *lineno*).

`s$section()` performs range checking (by comparing `subscr_list` with `array_attr`), calculates the section parameters, and writes them and the other attributes into the array section's attribute vector, *sect_attr*, which is allocated prior to the call. If the parent array is itself a regular section (viz. it is assumed-shape), the resultant section parameters for the section-of-a-section are written to *sect_attr*. Finally, the procedure checks that the section's shape (i.e. its *sectn'* vector) conforms with the target shape contained in *target_attr*, if it is defined.

If any parameters of *target_distr* are *undefined*, `s$section()` initialises them if possible as described in section 4.2.5.

If the procedure finds a range or conformance error, it generates a runtime error message and halts the program.

3. If the 'array section' is actually an array *element*, then the action taken depends on its distribution and the characteristics of *target_distr*.

   A globally replicated element is guaranteed to be co-distributed with any *target_distr*, so processing is complete at this point and this procedure returns.

   If the element is distributed, and is *not* in a scalar expression for which shape extension is allowed, then it must be broadcast. An ADLIB procedure has been defined specially for this purpose, `s$broadcast_elem()` (see section A.2). It broadcasts a single element with arbitrary distribution globally over the process grid, in contrast to `s$broadcast()`, which broadcasts an arbitrary regular section in just one process grid dimension. A call is generated to `s$broadcast_elem()`, after which this procedure returns.

Otherwise, the array element may have to be re-distributed in order to be co-distributed with *target_distr*, which is done in step 5 of this procedure.

4. Now irregular sections (viz. those with vector subscripts) are processed. If a *distributed* dimension has a vector subscript, then elements must generally be communicated to put them in the required order. This is done by generating code in the output program to call the ADLIB library routine `s$scatter()` (see section A.2), which re-arranges an irregular section into a regular one, simultaneously checking that subscripts are in range:

```
CALL s$scatter (sect,  sect_attr,    &
                array, array_attr,   &
                vec,   vec_attr, dim)
```

   This is done for each *distributed*, vector-subscripted dimension.

   Incidentally, we recall that the distribution of an irregular section is arbitrary except for the parameters d' and *sectn*$'$ (see section 4.2.1). However, it must be known before calling `s$scatter()`, so how is it chosen? We know that *target_distr* is completely determined at this point, as a result of step 2(iv) (since a distributed array section, like the present one, determines it), so the distribution of an irregular section in dimension i is chosen as follows:

   - If $(d'(\text{section}, i) = d(\textit{target\_distr}, i))$, copy the distribution from dimension i of *target_distr*.
   - Otherwise a 'contiguous' distribution is chosen, i.e.: ($n'(i)$=*sectn*$'(i)$, *sectl*$'(i)$=0, *sects*$'(i)$=1), and the *loc_* parameters are determined as usual.

   Redistribution is not required if the dimension is *internal*. In that case the vector subscript is copied though to the output program. It is only necessary to check that the values in the vector subscript are in range, which is done by generating a call to another ADLIB routine:

```
CALL s$checkvec (vec, dim, array_attr,  &
                 vec_name, lineno)
```

   This is done for every internal, vector-subscripted dimension.

5. At this point, the primary in question is a *regular* array section. If it is globally replicated then it is guaranteed to be co-distributed with any target distribution, so this procedure returns at this point.

   Otherwise, the array section may have to be re-distributed in order to achieve co-distribution with the *target_distr*. This even applies to an array element – having reached this stage, it must have a non-trivial distribution and be part of a scalar expression whose result can also have a non-trivial distribution, so the potential need for re-distribution applies to the array element as much as to proper array sections.

The section's distribution is compared with *target_distr*, and modified if necessary, in two stages. The first stage is concerned just with the process mapping, that is, the gross mapping of the data dimensions onto the process array dimensions, ignoring the distribution of elements *within* a process dimension. This aspect of the distribution is called the '*dimensional distribution*', and its modification is called '*dimensional redistribution*'.

The second stage is concerned with the distribution of elements *within* each process array dimension, for which we use the terms '*sectional*' distribution and redistribution. These stages are described in the following subsections.

6. Finally, all temporary vectors are de-allocated.

4.5.1 Storage allocation

Steps 4 and 5 may require temporary storage to be allocated to hold the results of data re-distribution. Before any such allocation or redistribution takes place, storage is allocated for the final result of the data re-distribution, namely an array with distribution *target_distr*İt is optimal to allocate storage for the final result before any intermediate temporaries, as then the latter can be de-allocated at the end, leaving just the final result.

Note that redistribution and allocation are *not* performed unless the array section is distributed, in which case *target_distr* is guaranteed to be completely determined (as a result of step 2(iv)), so the result array *can* be allocated.

If steps 4 and 5 involve a sequence of redistribution phases, then a sequence of temporary intermediate arrays is created whose total storage requirement is cumulative. This is an area that would benefit greatly from optimisation, which will hopefully be a topic for future research.

4.5.2 Dimensional redistribution

We now present the algorithm used to perform dimensional redistribution of an array section (which includes an array element as a limiting case). It is assumed that the target distribution is completely defined. If not, the section in question must be non-distributed (since the first distributed array section completely determines the target distribution), so dimensional redistribution is unnecessary.

The following description assumes that the *sectn* and d parameters have constant values for both the source and target distributions, in which case the analysis can be performed statically by ADAPT. Otherwise, ADAPT outputs code to call a general-purpose re-distribution routine, `s$redistr()` (see section A.2), which performs this analysis and re-distribution dynamically.

We have seen that any array section can be co-distributed with any target distribution by globally broadcasting it. However, our guiding principle is to distribute data rather than replicating it whenever feasible. This even applies to intermediate stages in dimensional re-distribution – if there is a choice between a process mapping of (0) and $(\mu \neq 0)$ at any stage of the algorithm, the latter is chosen, even at the expense of extra communications. The main reason is to minimise the use of temporary storage, which may easily be the limiting factor in the usefulness of this system. However, this does not apply to sections with extent $sectn = \pm 1$. As we shall see later, the allocation routines allocate a local segment of extent 1 for such dimensions whether they are distributed or replicated. In that case, minimising communications is our chief concern.

The algorithm for dimensional redistribution is summarised in Figure 6. It involves only two of the distribution attributes, namely $(sectn, d)$. For the purpose of this algorithm, any unmatched dimension with $sectn = -1$, in either the source or target distribution, is paired with an extra dimension of extent (1) and process mapping (0) (viz. n=1, d=0, *sectn*=-1, *sectl*=0) which is introduced in the corresponding position in the dimension list of the opposing array section – this makes no difference to the number or distribution of its elements. With this modification, the source and target distributions have the same number of dimensions, *ndims*. $(sectn(i), d(i))$ refer to the source distribution, and $(sectn'(i), d'(i))$ to the target distribution.

broadcast_distr(), *rotate_distr*() and *transpose_distr*() generate code in the output program to call the library procedures `s$broadcast()`, `s$rotate()` and `s$transpose()` (see section A.2). These respectively broadcast a distributed data dimension i so that it is held internally on each process, rotate the process mapping of data dimension i from process array dimension $d(i)$ to $d'(i)$, and transpose the process mapping of data dimensions (i, j). In every case, code is output to allocate storage for the result before the procedure is called (unless the result is an assignment variable). We should point out that `s$transpose()`, which transposes the *mapping* of data dimensions onto process array dimensions, has a different effect from the (distributed version of) the Fortran 90 intrinsic function `transpose()`, which transposes the data itself. The distinction is in the storage order within each process of the elements of the result: in the former case, they are stored according to the original ordering of the data dimensions; in the latter, the data dimensions are transposed and the storage order is changed accordingly.

For each dimension i, the action taken depends on the source and target process mappings, as follows:

1. *Source and target process mapping are both the same*: no action is taken.

```
    for (*all data dimensions*) i := 0 to ndims − 1 do
1.      if d(i) = d'(i) then
            (*skip*)
2.      else if d(i) = 0 then
            if d'(i) = d(j), i < j ≤ ndims − 1 and d'(j) = 0 then
                transpose_distr (dest←src,i,j)
            (* else skip – it's ok to assign a distributed target dimension
              from a global source dimension *)
3.      else if d'(i) = 0 then
3a.         if d(j) = 0, i < j ≤ ndims − 1 and d'(j) ≠ 0 then
                transpose_distr (dest←src,i,j)
3b.         else
                broadcast_distr (dest←src,i)
4.      else if sectn(i) = ±1 then
            broadcast_distr (dest←src,i)
5.      else if d'(i) = d(j), i < j ≤ ndims − 1 then
            transpose_distr (dest←src,i,j)
6.      else
            rotate_distr (dest←src,i,d'(i))
```

Fig. 6: Procedure for dimensional re-distribution

2. *Source dimension internal ($d(i) = 0$); target dimension distributed ($d'(i) = \mu \neq 0$).*
   As we discussed in section 4.2.6, re-distribution is unnecessary in this case as well – a replicated source dimension is regarded as co-distributed with any target process mapping, as the source can be assigned to the target without any communications.

   However, in the situation when a pair of source dimensions i and j (not necessarily consecutive) have process mapping $(0, \mu)$, and the corresponding target dimensions have process mapping $(\mu, 0)$, we choose to *transpose* the distribution of those source dimensions. (This is a special case of a distribution transpose, where one of the dimensions is internal and the other distributed – we refer to it as an *internal-external transpose*). This achieves the correct process mapping in both dimensions i and j simultaneously. If we did not do this, we would subsequently need to broadcast dimension j anyway, resulting in a final process mapping of (0,0) for these two dimensions, which occupies a larger amount of temporary storage than the target distribution requires. Also, the complexity of a internal-external transpose is the same as that of a broadcast operation.

3. *Source dimension distributed* ($d(i) = \mu \neq 0$); *target dimension internal* ($d'(i) = 0$).
   In this case we *must* broadcast the source dimension. There are two sub-cases:

   3a. There is a subsequent dimension j, such that in dimensions (i, j) the source process mapping is $(\mu, 0)$ and the target process mapping is $(0, \nu \neq 0)$. In this case, we transpose the process mappings of dimensions i and j (another internal-external transpose). Subsequently, we must change the process mapping of dimension j from $\mu \rightarrow \nu$ (unless $\mu = \nu$). The data redistribution is organised in this way to save temporary intermediate storage, as the output of this step is an array with process mapping $(0, \mu)$ rather than the larger array with process mapping (0,0) that would result from a straightforward broadcast. This saving in storage is made at the expense of extra data movements (a transpose + rotate, versus a single broadcast). The reason for specifying that $\nu \neq 0$ is that there is no point in transposing dimensions i and j ($(\mu, 0) \rightarrow (0, \mu)$), and subsequently broadcasting dimension j ($\mu \rightarrow 0$) – this results in an extra operation compared with a direct broadcast of dimension i.

   3b. Otherwise the distributed dimension is broadcast directly ($\mu \rightarrow 0$).

4. *Source process mapping* $\mu \neq 0$; *target process mapping* $\nu \neq \mu$; *sectn* $= \pm 1$.
   A distributed dimension with one element selected is broadcast rather than rotated. The same storage allocation is required whether it is distributed or replicated, and the communication cost of a broadcast ($\mu \rightarrow 0$) is similar to that of a rotation ($\mu \rightarrow \nu$) (as performed in branch 6 below) and less than that of a transpose (as performed in branch 5).

5. *Source process mapping* μ; *target process mapping* $\nu \neq \mu$; *and there is a subsequent source dimension* j *with process mapping* ν.
   In this case, we transpose the process mapping of dimensions i and j. (Note that we cannot change the process mapping of dimension i from $\mu \rightarrow \nu$ directly, as we cannot have *two* data dimensions distributed over the same process array dimension ν.)

6. *Source process mapping* μ; *target process mapping* $\nu \neq \mu$; *and none of the remaining source dimensions have process mapping* ν.
   Then we rotate the process mapping of dimension i from $\mu \rightarrow \nu$. (Note: this is achieved by a variant of a broadcast operation, and so has the same complexity).

4.5.3 Sectional redistribution, and co-distribution criteria

As a result of the last stage, one of the two criteria for the co-distribution of an array section with a target distribution has been met, namely, 'dimension co-distribution'. This means that each data dimension is distributed over the process dimension specified by *target_distr*, or else is undistributed (in which case it is regarded as being co-distributed with *any* target dimension); in other words, that the d parameters of the source match those of the target.

The second criterion is 'sectional co-distribution', which means that each dimension has the correct mapping of its selected elements onto the process coordinates. This aspect of the distribution concerns the remaining distribution parameters, namely (n, *sectn*, *sectl*, *sects*) (although obviously *sectn*, the number of elements in the section, cannot be changed).

Although the main purpose of the routines used in the last section, `s$broadcast()`, `s$rotate()` and `s$transpose()`, is to perform *dimensional* redistribution, they simultaneously perform *sectional* redistribution in the dimensions concerned. For example, `s$rotate()` can take a regular section (l:u:s) of a dimension with extent n and process mapping (μ), and rotate it to a regular section (l':u':s') of a dimension with extent n' and process mapping (ν) (provided only that the sections select the same number of elements). Therefore, sectional redistribution may be required only in dimensions that were already distributed over the correct process array dimensions.

For each dimension, the following tests are used to determine whether the elements of a pair of regular sections are co-distributed. If not, the ADLIB procedure `s$resection()` is used, which can perform arbitrary sectional redistribution within a given process array dimension (see section A.2). As before, we represent *source* attributes by *unprimed* names and *target* parameters by *primed* names.

1. If $d = 0$, co-distribution is guaranteed.

2. Otherwise, if the sections contain a single element, $(\mathit{sectn} = \mathit{sectn}' = \pm 1)$, co-distribution implies that they reside in the same process, that is:

   $$\lfloor \mathit{sectl}/\mathit{locn} \rfloor = \lfloor \mathit{sectl}'/\mathit{locn}' \rfloor$$

3. If $(\mathit{sectn} = \mathit{sectn}' > 1)$, the sections are co-distributed if both of the following conditions are satisfied:

   (a) The ratio of the section stride to the extent of the local segment, *sects*/*locn* (in *real* arithmetic), must be equal for both sections. Expressed in a form suitable for integer arithmetic, this condition is:

   $$\mathit{sects} \star \mathit{locn}' = \mathit{sects}' \star \mathit{locn}$$

This condition means that the selected elements of the two sections have identical 'spacing' with respect to the process array.

(N.B.: with more 'uniform' distribution strategies, which do not result in the same-sized segments being used in all processes, this condition would have to be replaced by the more restrictive conditions that $n = n'$ and *sects* = *sects*$'$).

(b) The ratio of the section lower bound to the extent of the local segment, *sectl*/*locn* (in *real* arithmetic), must also be equal for both sections, or:

$$sectl \star locn' = sectl' \star locn$$

This means that the lower bounds of both sections have the same location with respect to the process array.

To illustrate these conditions, consider two arrays declared as `A(0:13)` and `B(0:5)`, distributed over 4 processes. Then:

$$\begin{aligned} locn(A) &= (14 + 4 - 1)/4 = 4 \\ locn(B) &= (6 + 4 - 1)/4 = 2 \end{aligned}$$

Our criteria indicate that the section `A(2:10:2)` (with *sectl* = *sects* = 2) is co-distributed with the section `B(1:5)` (with *sectl* = *sects* = 1), and the diagram below shows that this is the case:

array	process number (p)															
	0				1				2				3			
	elements allocated to each process*															
A(0:13)	0	1	**2**	3	**4**	5	**6**	7	**8**	9	**10**	11	12	13	-	-
B(0:5)	0		**1**		**2**		**3**		**4**		**5**		-		-	

* elements belonging to the array sections A(2:10:2) and B(1:5) are in **bold**

In fact, condition (b) on the section lower bounds can be relaxed somewhat. For example, the above diagram shows that the section `A(3:11:2)` is also co-distributed with `B(1:5)`, though it does not satisfy condition 2. Here we give an algorithm to test a more general condition:

(b$'$) Find η, the highest common factor of *locn* and *sects*. Likewise, obtain η' for the other section. The sections are co-distributed if:

$$\lfloor sectl/\eta \rfloor = \lfloor sectl'/\eta' \rfloor$$

The parameter η gives, for the source section, the number of elements in a 'stride segment'. This is defined such that both *sects* and *locn* are an integral number of stride segments, which means that every stride segment is located completely within a process, and also that the selected elements of the source array can be numbered in units of stride segments. The same applies to the selected elements of the target. Furthermore, condition (a) ensures that both the source and target have the same number of stride segments per process, and the same number per *sects*. Thus, if the lower bounds of both sections have the same stride segment number, then the sections will be co-distributed.

To simplify the task of factorising *locn* and *sects*, a typical implementation would just test for a small number of common factors, e.g. 2,3,5,7.

These conditions are sufficient but not necessary: it is possible for sections to be co-distributed and not satisfy them (for example, one can relax the condition on the strides if all the selected elements in both sections reside in the same process). How comprehensive these tests are made is a matter of judgement.

4.6 Minimising storage allocation for target distributions

The array allocation routines attempt to minimise the amount of storage they allocate to each temporary array, for obvious reasons. They take as an input parameter the attribute vector, *attrvec*, of the array to be allocated. If the sectional parameters in *attrvec* indicate that just a subset of the specified array will be used, the array size is reduced if possible to obtain an equivalent array whose *selected elements are located in the same process(es) as those of the original*, and the output attribute vector is modified accordingly. Notice that, for distributed dimensions, the requirement that a regular section has a given mapping onto the process array rules out the possibility of naively reducing the dimension size to that of the regular section – because then the regular section would be uniformly distributed over the process array dimension, which may be incorrect.

The algorithm for minimising the storage allocation is as follows:

For each dimension:

1. If the dimension is internal ($d = 0$), reduce the dimension size to that of the regular section:

$$
\begin{aligned}
n' &= \mathit{locn}' = \mathit{sectn}' \\
\mathit{locl}' &= \mathit{sectl}' = 0 \\
\mathit{locu}' &= n' - 1 \\
\mathit{sects}' &= 1
\end{aligned}
$$

2. If the section selects one element ($sectn = 1$), reduce the local segment extent to 1:

$$\begin{aligned} locn' &= sectn' = 1 \\ n' &= np(d(i)) \\ locl' &= p \\ locu' &= 0 \end{aligned}$$

Now, we want this element to be located in the same process p' as before (namely, $p' = \lfloor sectl/locn \rfloor$). Hence,

$$sectl' = \lfloor sectl/locn \rfloor$$

3. For a distributed section with more than 1 element, find η, the highest common factor of *sects* and *locn*.
If $\eta > 1$:

$$\begin{aligned} locn' &= locn/\eta \\ n' &= locn' \star np \\ sectl' &= \lfloor sectl/\eta \rfloor \\ sects' &= sects/\eta \\ locl' &= p \star locn' \\ locu' &= locn' - 1 \end{aligned}$$

4.7 Processing an assignment

The final stage in processing an assignment statement, or an expression that has been transformed into an assignment (section 4.3), is to generate code in the output program to actually perform the assignment.

This is straightforward for an assignment to a simple scalar variable – it is simply a case of writing out the variable and the accompanying expression tree.

If the assignment is array-valued, however, it must be transformed it into an equivalent elemental assignment enclosed in DO-loops over the array subscripts, because Fortran 77 does not support array assignments! Also, the DO-loop must restrict the assignment to the locally-stored segment of the assignment variable, which generally means that the DO-loop bounds are process dependent. For a distributed dimension with just one element selected, this 'process selecting' function is performed by an IF construct rather than a DO-loop.

This section describes the procedure for processing an assignment to a general array section (which we are still using as a generic term for a whole array, array section or array element). It is assumed that all data are co-distributed with the assignment variable as a result of previous processing.

At the start of this procedure, the *real global* subscripts of the various array sections in the assignment are available via their sectional parameters (*sectn*, *sectl*, *sects*), which are stored either in internal distribution vectors or explicitly in attribute vectors in the output program. The stages of the procedure are as follows:

For each dimension i of the array assignment, in reverse order (i.e. highest dimension first):

1. Classify dimension i according to the criteria in the following table. Unless otherwise specified, the parameters are those of the assignment variable.

Properties of dimension i		class
sectn $= \pm 1$, $d = 0$		(a)
sectn $= \pm 1$, $d \neq 0$ or unknown		(b)
sectn > 1 or unknown:		
$d = 0$	all array-valued terms have equal *sects* and *sectl*	(c)
	all array-valued terms have equal *sects*	(d)
	otherwise.	(e)
$d \neq 0$ or unknown	all array-valued terms have equal d, *locn*, *sects* and *sectl*	(f)
	all array-valued terms have equal *sects*	(g)
	otherwise.	(h)

   Assignments may contain terms with vector subscripts. For vector subscripted dimensions, it is the *sectl* and *sects* parameters of the *vector subscript* that are relevant in comparing terms for the purpose of classifying the dimension. (Incidentally, all vector subscripts and the dimensions they index must be internal – $d = 0$ – since *distributed* dimensions with vector subscripts were re-arranged into regular sections during the data redistribution phase. Therefore, such terms can only occur in classes (c)-(e), (g) and (h)).

   If dimension i of the assignment variable specifies a single element or is internal (i.e. it belongs to one of the classes (a)-(e)) then no further analysis is necessary and a jump is made to step 3 of this procedure, which generates an `IF-THEN` or `DO` statement in the output program, if necessary, to select the required elements of the assignment variable.

2. If the assignment is to a *distributed* section (or to a section of unknown process mapping) containing > 1 element (or an unknown number of elements), it is necessary to evaluate the bounds and number of elements of the *local segment* of the section, (*segl*, *segu*, *segn*), in terms of *local* subscripts.

For the *assignment variable* only, the following parameters are evaluated:

$$
\begin{aligned}
l &= sectl - locl \\
u &= l + (sectn - 1) \star sects \\
\eta &= \begin{cases} \mathrm{MAX}((sects - l - 1)/sects, 0); & sects > 0 \\ \mathrm{MAX}((l - locu - sects - 1)/(-sects), 0); & sects < 0 \end{cases} \\
segl &= l + \eta \star sects \\
segu &= \begin{cases} \mathrm{MIN}(u, locu); & sects > 0 \\ \mathrm{MAX}(u, 0); & sects < 0 \end{cases} \\
segn &= \mathrm{MAX}((segu - segl + sects)/sects, 0).
\end{aligned}
$$

In addition, $segl'$ is calculated for dimension i of each array section in the assignment expression. This is the local subscript of the element 'corresponding' to element *segl* of the assignment variable (i.e. occupying the same position in the regular section). It is given by:

$$segl' = sectl' - locl' + \eta \star sects'$$

where η is the value obtained above and the other parameters are specific to the section concerned (obviously). This formula is valid for both distributed and internal dimensions; in the latter case, $locl' = 0$. For *vector* subscripted dimensions, the formula refers to the vector subscript rather than the dimension that it indexes.

It is perhaps worth giving some explanation of these formulae. l and u are simply the *local* subscript values of the lower and upper bounds of the whole regular section (in dimension i of the assignment variable). For the remaining parameters, we discuss the case where the regular section has positive stride, $sects > 0$. In that case, the section is empty unless $l \leq u$. Therefore, *segl* is defined to be the smallest local subscript ≥ 0 that is an integral number of strides above l, or l itself, whichever is larger. That is, $segl = (l + \eta \star sects)$, such that $\eta \geq 0$ and $segl \geq 0$. This defines the integer parameter η, namely, it is the sequence number within the regular section of the element with local subscript *segl*, starting at $\eta = 0$ for the first element of the section. (Note that the only claim we can make for element η is that, if the local segment contains *any* elements of the regular section, then η is the first such element. If not, then either element η is *not in the local segment* because $segl = l > locu$ – in which case $\eta = 0$ – or it's *not in the regular section* because $\eta > sectn - 1$). *segu* is defined to be the last element in the local segment, or u itself, whichever is smaller.

This is not necessarily a *selected* element (i.e. separated from l by an integral number of strides *sects*) – but it does not have to be because it is used as the upper limit of a DO-loop, as we shall see shortly. The locally-stored elements of the regular section are then given by the section (*segl*:*segu*:*sects*), and *segn* is the number of elements in this section. It can be seen that *segn* may be 0.

The formulae for *sects* < 0 have a similar explanation, except that now the section is empty unless $u \leq l$. Therefore, *segl* $= (l + \eta \star sects)$ is now defined to be the *last* element in the local segment that is an integral number of strides below l, or l itself, whichever is *smaller*, and *segu* is the first element of the local segment, or u itself, whichever is *larger*.

For each array section in the assignment expression, η is used to derive the local subscript *segl*$'$ of the element corresponding to element *segl* of the assignment variable. Note that the sequence number provides the only way of identifying the corresponding element if the dimension of the term is *not* distributed (which is allowed as an internal dimension is co-distributed with any target distribution).

These expressions are stored internally by ADAPT. However, if *sects* is not constant (i.e. its value is not known by ADAPT), then η and *segu* must be evaluated dynamically by outputting code for the above equations within an `IF-THEN-ELSE` block.

It is evident that all of the above parameters are given by non-constant expressions, as they depend on the lower bound of the local segment, *locl*, which is process dependent.

3. If necessary, generate a `DO` or `IF-THEN` statement in the output program. The appropriate form of this statement depends on the class of the dimension, as shown in Table 1. The control expressions for classes (b)-(e) all involve parameters that may be constant (viz. *sectl*, *sectn*, *sects* and *locn*), in which case the expressions are evaluated statically before being output. However, this is not the case for the DO-loop control parameters of classes (f)-(h), namely *segl* and *segu*.

When all dimensions of the assignment variable have been processed as described above, it only remains to output the assignment statement itself, transformed into an elemental assignment (i.e. with references to array sections transformed into references to array elements). Table 1 shows the appropriate subscript expression for each dimension of an array element, which depends on the class of the dimension.

Notice that, for classes (f)-(h), the criterion for choosing between the various forms of DO-loop is based on the *sectl* parameters of the terms rather than on their *segl* values, which is the more obvious choice. The reason is that, for distributed dimensions, *segl* cannot be a compile-time constant (as it depends on the process coordinate), so the use of this parameter in our

dim. type	conditions	Fortran 77 output* corresponding to the assignments `G(I) = ...,  D(I1) = G(I2) ...,  G(L:U:S) = ...` and `D(L1:U1:S1) = G(L2:U2:S2) ...` where `G()` is an internal dimension and `D()` is distributed.
(a)	*sectn* $= \pm 1, d = 0$	`G(SECTL_G) = ...`

(b)	*sectn* $= \pm 1$	`IF (P .EQ. (SECTL_D/LOCN_D)) THEN`
		`  D(SECTL_D - P*LOCN_D) = G(SECTL_G) ...`
		`ENDIF`

(c)	$d = 0$;
	all terms have equal *sects*, *sectl*
		`DO I = SECTL, SECTL + (SECTN-1)*SECTS, SECTS`
		`  G(I) = ...`
		`ENDDO`

(d)	$d = 0$;
	all terms have equal *sects*
		`DO I = 0, (SECTN-1)*SECTS, SECTS`
		`  G(SECTL_G + I) = ...`
		`ENDDO`

(e)	$d = 0$	`DO I = 0, (SECTN-1)`
		`  G(SECTL_G + I*SECTS_G) = ...`
		`ENDDO`

(f)	all terms have equal d, *locn*, *sects*, *sectl*	`DO I = SEGL, SEGU, SECTS`
		`  D(I) = ...`
		`ENDDO`

(g)	all terms have equal *sects*	`DO I = 0, SEGU_D-SEGL_D, SECTS`
		`  D(SEGL_D + I) = G(SEGL_G + I) ...`
		`ENDDO`

(h)	otherwise	`DO I = 0, (SEGN-1)`
		`  D(SEGL_D + I*SECTS_D) = G(SEGL_G + I*SECTS_G)`
		`ENDDO`

* For clarity we use Fortran 90's `DO-ENDDO` format rather than Fortran 77's labelled `DO`-loops!

Table 1: Transformation of a general assignment (which may be distributed and/or array-valued) to an equivalent local elemental form.

criteria would force all distributed dimensions to be transformed into the most general, and least efficient, form of DO-loop. On the other hand, *sectl* may be constant for a distributed dimension. Its use is legitimate because of our co-distribution criteria, namely: for distributed sections, identical *sects* $\Rightarrow$ identical *locn*, so identical *sectl* $\Rightarrow$ identical *segl*.

Other points of interest about the transformation of an assignment statement are:

- Since a distributed dimension of the assignment variable can correspond to a non-distributed (i.e. internal) dimension of an operand, the table shows subscript expressions for both cases where appropriate, using the names `D()` and `G()` to denote distributed and non-distributed arrays respectively.

- A dimension of the assignment variable with *sectn* $= -1$ (a subset of classes (a) and (b)) may have no corresponding dimension in an operand.
- An operand may have extra dimensions with (*sectn* $= -1$, $d = 0$). Each is subscripted by its *sectl* value.
- An operand may of course be scalar. If an array element appears in an array-valued assignment, each dimension of the element is subscripted by its *sectl* value.
- An internal dimension may have a *vector subscript*, V say. In that case, the appropriate subscript expression is:

  $$sectl + V(e) \star sects$$

  where *sectl* and *sects* are parameters of the *parent array* (rather than the *array section*, as elsewhere). e is the subscript expression derived from Table 1. This assumes that V contains *normalised global* subscripts for the parent array, which it has as a result of a previous processing step (viz. step 1 of *process_array_section*()).
- An *allocated* array must be subscripted as a 1-dimensional vector. If it is logically an n-dimensional array, an element with subscripts $(e_1, e_2, \ldots e_n)$ is referenced by the subscript expression:

  $$ptr + e_1 + locn_1 \star (e_2 + locn_2 \star (\ldots e_n))$$

  where *ptr* is a pointer to its first element.
- Elemental function arguments have the same output format as normal operands.

4.8 WHERE statement and construct

We conclude this section by describing how ADAPT implements Fortran 90's `WHERE` statement and construct. These provide a means of performing an array assignment only to certain elements of an array.

We start with the `WHERE` statement, whose general form is:

`WHERE` (*logical-array-expr*) *array-variable* = *array-expr*

The *logical-array-expression* is called a *mask*, and it must have the same shape as the *array-variable*. It is evaluated first, and then those elements of *array-expression* that correspond to elements of the *mask* that are true are evaluated and assigned to the corresponding elements of *array-variable*. The other elements of *array-variable* are left unaltered. An example is:

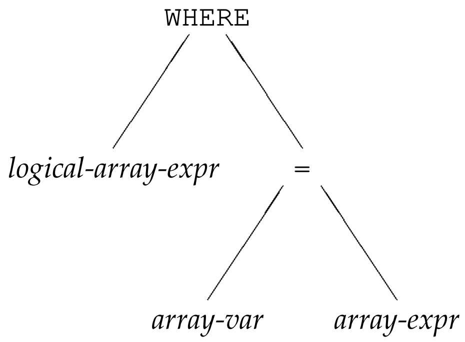

Fig. 7: The expression tree for a `WHERE` statement

```
WHERE (A > 0.0) A = 1.0 / A      ! A is a real array.
```

which only performs the expression evaluation and assignment for positive elements of `A`.

The whole `WHERE` statement is processed as a unit, after the expression trees of the mask expression and assignment statement have been constructed. These are combined into a single expression tree whose root node is the `WHERE` keyword, as shown in Figure 7.

The variable is processed first, and provides the target distribution for both the mask and assignment expressions. The variable must be array-valued. Then both the mask expression and the assignment expression are processed using the procedure *process_expr*(), as a result of which both are co-distributed with the assignment variable.

Notice, incidentally, that the mask expression is *not* in an assignment statement, so if necessary *process_expr*() introduces a temporary variable in which to store its result, as described in section 4.3. The result of the mask expression is also stored in a temporary variable if it involves the assignment variable, or if it masks a `WHERE` construct. This is done to avoid any possible dependence between the mask expression and the assignment(s), as the value of the mask must not be affected by the masked assignment(s).

The final stage of processing the `WHERE` statement is to generate code in the output program to perform the masked assignment. This is done as described in the last section, except that the mask expression is included in the various analysis stages along with the terms in the assignment statement. This means that it is included in the comparison of terms for the purpose of classifying dimensions, and that its local lower bound parameters, *segl*, are derived. When the equivalent elemental assignment is output, it is enclosed in an `IF` construct whose `IF-THEN` statement contains the corresponding element of the mask expression. The required element of the mask expression is referenced in the same way as elements of the assignment (see Table 1).

A `WHERE` *construct* is treated as a sequence of `WHERE` statements, except that the result of the mask expression is preserved throughout the `WHERE` construct. It is still processed by *process_expr*() once for each masked assignment, however, in case it must be re-distributed in order to be co-distributed with the assignment variable. The processing of an `ELSEWHERE` block is identical to that of the `WHERE` block, except that `(.NOT.` *mask_element*`)` rather than `(`*mask_element*`)` is output in the `IF-THEN` statement enclosing the elemental assignment.

5 Suggested language extensions

From the outset it has been our principle to add as few extensions as possible to standard Fortran 90, and to make those that we do add as simple and portable as possible. We have been quite successful in this aim (possibly excluding the portability criterion), as we have only introduced one extra keyword: `DISTRIBUTION`.

However, in this section we argue the case for a couple of other extensions to Fortran 90 that would assist the generation of efficient code for distributed memory machines, and probably for other architectures as well, while not impairing program clarity or portability.

5.1 `FORALL` loops

ADAPT performs no transformation on a DO-loop appearing in the input program (apart from changing its labels) – it is copied through to the output program. This means that every process executes every iteration of the loop. This is usually the desired behaviour, as the array programming style normally uses loops for sequential iteration (typified by the `DO-WHILE` type of loop) rather than for iteration over array elements. If an operation can be performed on many array elements concurrently, then, for the purpose of ADAPT, it should be expressed by an array assignment rather than by a DO-loop over array element assignments.

However, there are cases where an operation *can* in principle be performed on many array elements concurrently, but cannot be expressed as an array assignment because the array syntax of Fortran 90 is too limited. An example is where the value assigned to each element depends on its *subscripts*. For instance, if one wants to define a mask vector `EVEN(0:N-1)` whose elements are `.TRUE.` if their subscripts are even and `.FALSE.` otherwise, it can be done easily using a DO-loop:

```
DO I = 0,N-1
   EVEN (I) = (MOD (I,2) == 0)
ENDDO
```

but cannot be expressed in Fortran 90 array syntax. It is therefore interesting to consider how this DO-loop is transformed by ADAPT. Notice that the assignment involves no communications even if EVEN is distributed (which is why it can potentially run concurrently).

As we have said, ADAPT copies the DO-loop through to the output program. If the vector EVEN is distributed, then the assignment to element EVEN(I) generates the following code in the output program (see Table 1 in section 4.7):

```
IF (I$P .EQ. (I/LOCN)) THEN
   EVEN (I - I$P*LOCN) = (MOD(I,2) .EQ. 0)
ENDIF
```

where I$P is the coordinate of the local process. Therefore, the output for the whole DO-loop is:

```
DO I = 0,N-1
   IF (I$P .EQ. (I/LOCN)) THEN
      EVEN (I - I$P*LOCN) = (MOD(I,2) .EQ. 0)
   ENDIF
ENDDO
```

Every process p executes every iteration of the DO-loop. In iteration i, the IF test is false unless EVEN(I) is stored within process p, in which case the assignment is performed. Therefore, the local segments of the assignment may indeed be executed concurrently across the process array, provided the total time for the loop is not dominated by the time to execute the empty iterations, as it might be for this simple assignment.

On the other hand, if the operation could have been written as an array assignment such as:

```
EVEN (0:N-1) = ...
```

then the corresponding output code would be:

```
DO I = SEGL, SEGU
   EVEN (I) = ...
ENDDO
```

where SEGL and SEGU delimit the local segment of the assignment variable. If EVEN has n elements distributed over p processes, then this form involves n/p DO-loop iterations compared with n DO-loop iterations and n IF tests before.

It would be nice to avoid this overhead by introducing an explicitly data parallel form of DO-loop. This is the purpose of the FORALL loop, which is often incorporated in data parallel dialects of Fortran [5, 7]. It is similar to a DO-loop, but explicitly states that there are no data dependencies within the loop so that it can be treated as an array operation, leading to the more efficient transformation shown above. (In fact, if a DO-loop that *does* have data dependencies, such as

```
DO I = 1,N
   A(I) = ...A(I-1)...
ENDDO
```

is replaced by a `FORALL` loop, its semantics are changed such that all expressions within the loop are evaluated using the data values in existence *before* the execution of the loop, rather than any new values assigned within the loop – i.e. it acquires the same semantics as an array assignment. Thus, a `FORALL` loop does not assert that the enclosed code does not involve communications, only that it is data-parallel).

Incidentally, if the vector `EVEN` is *not* distributed, then the DO-loop and its assignment statement are copied through unchanged to the output program, as the assignment statement is untransformed (it belongs to class (a) of table 1). A `FORALL` loop would make no difference in this case.

Although it is not particularly relevant to the present discussion, it is perhaps interesting to briefly consider the transformation of a DO-loop over elemental assignments which *does* contain data dependencies. An example is:

```
DO I = 1,N-2
   A(I) = 0.5 * (A(I-1) + A(I+1))
ENDDO
```

Each iteration $i > 1$ uses the value of `A(I-1)` computed in the previous iteration, so this loop is inherently sequential. Obviously it cannot be replaced by either an array assignment or a `FORALL` loop without changing its meaning.

As we know from the last section, if `A` is distributed, then the transformation of the assignment statement introduces communications (in fact, array 'shifts' to the right and left which are implemented by calls to `s$resection()`). As a result of these communications, the assignment just involves local data objects, so in the output program it is enclosed in a process-selecting `IF` construct as before. All of this, *including the communications*, is enclosed in the loop `DO I = 1,N-2`. In this case therefore, the communications force the DO-loop iterations to be performed sequentially across the process array, which is the desired result.

5.2 User-defined ELEMENTAL procedures

Fortran 90 introduces the concept of *elemental procedures*, which are defined for scalar arguments but may also be applied to conforming array-valued arguments. For an elemental function, each element of the result, if any, is as would have been obtained by applying the function to corresponding elements of the arguments.

We have seen in section 4.4 that the parallelisation of an elemental function reference with array arguments is more efficient than the transformation of an array function, as it requires no storage allocation for the result, and it can be invoked within the transformed elemental form of the expression, which probably saves DO-loops compared with an equivalent array function.

Unfortunately, Fortran 90 restricts the application of elemental procedures to a subset of the intrinsic procedures – they cannot be written by the programmer. We would therefore propose the extension of allowing the programmer to define elemental procedures, and especially functions. As well as being efficient they would be a very convenient programming tool, as the same elemental function or subroutine can be applied to arguments of any rank. Recent experiences of programming in Distributed Fortran 90 show that there are many occasions when such elemental functions would be useful (e.g. to initialise each element of an array or site of a grid, when the initial value is independent of neighbouring sites [8]).

We suggest that a suitable syntax would be to precede the `SUBROUTINE` or `FUNCTION` keyword by the new keyword `ELEMENTAL`. For a function to be valid for use as an elemental function, in the context of ADAPT, it must obey the following rules:

- It must have scalar arguments with intent `(IN)`, and a scalar result.
- Its local variables must not be distributed.
- It cannot define any global variable, and can only reference a global variable if it is *not distributed*.

(Confusingly, 'local' and 'global' are used here in the normal programming sense, rather than referring to distribution). Since by definition an elemental function can be referenced independently in different processes, it is essential that it does not perform any communications (as the communications procedures must be invoked uniformly along a process axis), and the last two rules above are designed to ensure this. Also, it must not modify global data ('global' in the programming sense), as otherwise it could not be used concurrently.

These rules show that a compiler (for example, ADAPT) cannot deduce from a procedure's interface whether it can validly be used as elemental procedure, as that depends on its local declarations and internal operations. Hence, it is necessary to use a specifier such as `ELEMENTAL` to identify such procedures. The compiler can check that the procedure satisfies all the necessary rules when it compiles the procedure itself.

6 The ADAPT pre-processor

Having described the techniques employed by ADAPT to effect the auto-parallelisation of Distributed Fortran 90, we now give a brief overview of the ADAPT pre-processor itself.

Since the pre-processor transforms Distributed Fortran 90 to Fortran 77, it contains a complete parser for Distributed Fortran 90 (a superset of

Fortran 90). This was developed from scratch as one was not available at the time. It is written in C, using the GNU 'bison' parser generator (which is similar to 'yacc'). For reasons of efficiency, the lexical analyser was developed by hand rather than using an automatic generator such as 'lex'. It can accept both types of source format available in Fortran 90, namely the old fixed source format (as in Fortran 77), and the new free source format.

The parser recognises the whole of Fortran 90. However, as explained in [2], it is impractical to attempt to transform *all* the new features of Fortran 90 into Fortran 77, as the facilities available in Fortran 77 are too limited. Therefore ADAPT does not to support several (quite major) features of Fortran 90 that are not essential to array processing, namely pointers, derived types and operators, allocatable arrays, internal subroutines and modules. It will recognise and check these features in a Fortran 90 program, but will give error messages reporting that they are not supported.

Considerable attention has been paid to details such as error-checking and reporting, as ADAPT is intended to be a high-quality tool for general use rather than merely a research tool. In fact, its error and warning messages should be more helpful than those of the typical Fortran 77 compiler (at least in the author's experience); e.g. it locates errors and warnings to the correct line rather than to the first line of the particular statement, gives a warning if fixed source form statements extend beyond 72 characters, can check for uninitialised variables, etc.

ADAPT does all of its checking and code generation in one pass of the input. All of the declarations of a program unit are scanned, and the symbol table constructed, before any declarations are output. The executable part of a program unit can generally be processed and output on a statement-by-statement basis, with a few exceptions (e.g. for a `WHERE` construct, the first assignment is also processed before any output is generated). The biggest problem for this statement-by-statement transformation scheme is caused by I/O statements that refer to `FORMAT` statements, since in the output program the format specifier must be passed as an argument to the appropriate I/O procedure, and hence is required at the point where the I/O is performed. This is implemented by an inelegant piece of 'trick' coding in the output program, whereby it performs a branch to the `FORMAT` statement, copies the format specifier into a temporary array, and then makes a return jump to the I/O statement, whose label was stored in a 'label' variable using Fortran's `ASSIGN` statement!

ADAPT performs a host of other transformations in addition to those described in this report. Many of them are concerned with the translation into Fortran-77 of the many new features introduced by Fortran-90 [2]. Also, because ADAPT introduces new labelled DO-loops into the output program (e.g. to transform array assignments into elemental ones) it changes the values of all labels appearing in the input program, in order to avoid possible clashes in the use of labels.

We have said that ADAPT generates Fortran 77 output. However, it does require the target compiler to accept the following non-standard extension to the Fortran 77 rules for names:

> Names may contain up to 31 characters, including underscore ('_') and dollar ('$') characters as well as alphanumeric characters. The first character must be alphabetic.

Apart from the inclusion of dollar characters, these are the Fortran 90 rules for identifiers, which must be supported by the target compiler because ADAPT does not modify the names used in the input program. The '$' character is not a legal character in either Fortran 77 or Fortran 90 identifiers, and it will not be allowed in the Fortran 90 input to ADAPT. However, it is included in all new identifiers that are introduced in the output program, to avoid the possibility of name clashes with user-defined identifiers. It is accepted by the *3L* Fortran compiler.

The use of non-Fortran 77 standard names in the output program could be avoided by ADAPT transforming all the input program's identifiers (just as it transforms labels), but this would make it difficult to relate the output program to the input one.

7 Conclusions

The system that we are developing should greatly simplify the task of programming data parallel applications for distributed memory universal message-passing architectures, by allowing the use of a shared memory programming style in a high-level, standard, portable language. Our work differs from other research in Fortran auto-parallelisation (e.g. [9, 5]) in several important respects: we concentrate on exploiting *new* features of Fortran 90 (and not just the array features), and our system generates calls to high-level communications procedures rather than low-level point-to-point communications. This has the advantages of not hard-wiring any particular message-passing model or syntax into the pre-processor, and of allowing communications to be optimised for each target architecture.

The ADAPT pre-processor is currently nearing completion. In a separate task in the PUMA project, an array communications library, ACL, [10], has been developed for the T9000 + C104 universal message-passing architecture, and it is currently being rewritten using ParMacs macros [11] for portability. This library could form the basis for the ADLIB communications procedures.

There are many areas in which this research may be further developed. Of particular interest are the development of techniques to optimise communications and the use of temporary storage, and the enhancement of the system to support more general data distributions.

Bibliography

[1] Fortran 90. ANSI X3J3/S8.115, June 1990.

[2] J. H. Merlin. Parallelisation of Fortran 90 array programs. PUMA Deliverable Report D4.3.1, University of Southampton, October 1990.

[3] J. H. Merlin. ADAPTing Fortran 90 array programs for distributed memory architectures. In *1 st Intl Conf of the ACPC*, Salzburg, October 1991.

[4] M. Metcalf and J. Reid. *Fortran 90 Explained*. Oxford Science Publications. OUP, 1990. ISBN 0-19-853772-7.

[5] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C.-W. Tseng, and M.-Y. Wu. Fortran D language specification. Technical Report TR90-141, Dept of Computer Science, Rice University, December 1990.

[6] A. D. Aho and J. D. Ullman. *Principles of Compiler Design*. Addison Wesley, 1977. ISBN 0-201-10073-8.

[7] Thinking Machines Corp., Cambridge, MA. *CM Fortran reference manual, version 5.2-0.6*, September 1989.

[8] M. S. McCarthy. Fortran 90 in lattice QCD. PUMA Output O4.3.3, University of Southampton, December 1991.

[9] H. Zima, H.-J. Bast, and M. Gerndt. SUPERB, a tool for semi-automatic MIMD/SIMD parallelisation. *Parallel Computing*, 6:1–18, 1988.

[10] D. B. Carpenter. Array communication library: *Introduction* and *Reference Manual*. PUMA Output O5.1.1, University of Southampton, May 1991.

[11] L. Bomans, D. Roose, and R. Hempel. The Argonne/GMD macros in FORTRAN for portable parallel programming. *Parallel Computing*, 15:119–132, 1990.

A ADLIB procedure interfaces

A.1 Overview

This appendix specifies procedure interfaces for part of the ADLIB library, namely for the data redistribution routines and the procedures corresponding to the Fortran 90 array intrinsic functions. The procedures for I/O, storage allocation, checking and initialisation are not included.

All of the procedures in ADLIB share some common features, which we briefly describe here.

Firstly, every array argument is accompanied in the procedure's argument list by an *attribute vector*, which contains all the essential information about the array. This is in accordance with the scheme universally employed by ADAPT for array arguments (see section 3.2). As well as the datatype and rank of the array, the attribute vector specifies, for each data dimension i:

- the total *extent* of that dimension;
- the process array dimension over which that data dimension is distributed (which is called the *process mapping*);
- the *section parameters*, ($sectn(i)$, $sectl(i)$, $sects(i)$), which specify whether the procedure is to act on a regular section of the dimension rather than the full dimension.

(In addition, the attribute vector specifies the extent and bounds the local segment, which are redundant in that they can be calculated from the above parameters, but are included for convenience). These parameters completely specify the distribution of elements in a regular section of an array dimension, and are called the *dimensional parameters*, or simply the *distribution*, of that dimension. The attribute vectors have the specification:

```
INTEGER, DIMENSION (:), INTENT (IN)  :: attr
```

They are global (i.e. not distributed) and occupy contiguous storage.

Every array argument of an ADLIB procedure that is not an attribute vector has assumed shape and process mapping (and often assumed type and rank too). This means that the corresponding actual argument is in general a *regular section* of the array that is passed-in, which may itself have any shape and process mapping (all of which are specified by its dimensional parameters). Therefore, throughout the following descriptions, 'array' must be understood to mean the specified *subset* of elements of the dummy argument array on which the procedure operates. As a limiting case, this may be a single element or no elements at all!

Incidentally, for many ADLIB procedures we state that two or more array parameters have the 'same distribution' in certain dimensions. This does not necessarily mean that their dimensional parameters are *identical*, but rather that they are *equivalent*, meaning that they specify regular sections containing the same number of elements (i.e. have the same *sectn*), that they are distributed over the same process grid dimension d, and that corresponding elements of the different sections are mapped onto the same d coordinate of the process grid. In other words, their selected elements are 'co-distributed' in the given dimensions. Subject to these constraints, arrays with the same 'distribution' may have different dimensional parameters, as indicated in sections 4.5.3 and 4.6.

In general the output argument or arguments of a procedure are at the beginning of the argument list. In the case of an array output argument, only the elements within the specified regular section are defined.

Unless otherwise stated, there is no relation between the attributes of different array arguments.

Unfortunately it has not been possible to implement ADLIB in the timescale of the PUMA project.

A.2 Data redistribution

Except for `s$gather` and `s$scatter`, these procedures *copy* an input array `from` to an output array `to`, in general changing its distribution. The procedures `s$gather` and `s$scatter` *re-arrange* data from `from` into `to`, which may also involve data redistribution.

The following argument specifications are general to all of the data redistribution procedures:

```
ANYTYPE, DIMENSION (...),                               &
         DISTRIBUTION (...),                            &
         INTENT (OUT)           :: to
ANYTYPE, DIMENSION (...),                               &
         DISTRIBUTION (...),                            &
         INTENT (IN)            :: from
INTEGER, DIMENSION (:),                                 &
         INTENT (IN)            :: attr_to, attr_from
INTEGER, INTENT (IN)            :: dim, dim1, dim2
```

where `ANYTYPE` means any data type, and `(...)` means an assumed shape or process mapping of any rank (subject to the stated constraints).

Gather-scatter

```
SUBROUTINE s$gather (to,   attr_to,          &
                     from, attr_from,        &
                     v,    attr_v,    dim)
SUBROUTINE s$scatter(to,   attr_to,          &
                     from, attr_from,        &
                     v,    attr_v,    dim)
   INTEGER, DIMENSION (:),                   &
            DISTRIBUTION (),                 &
            INTENT(IN)        :: v
   INTEGER, DIMENSION (:),                   &
            INTENT(IN)        :: attr_v
```

These procedures re-arrange an irregular section in data dimension dim into a regular section, or vice versa, as follows:

procedure	operation performed
s$gather	$to(\ldots\ \underline{V}_{dim}\ \ldots) = from(\ldots\ \{d\}_{dim}\ \ldots)$
s$scatter	$to(\ldots\ \{d\}_{dim}\ \ldots) = from(\ldots\ \underline{V}_{dim}\ \ldots)$

(where $\{d\}_{dim}$ means a subscript triplet in dimension *dim*, $\underline{V}_{dim}$ means a vector subscript in dimension *dim*, and ... refers to optional subscripts or subscript triplets in other dimensions, which are the same on both sides of the assignment).

Arrays to and from have the same rank and distribution except in data dimension dim. In that dimension, to, from and v have the same process mapping, and, as required for conformance, the same extent *sectn* is shared by from and v (in the case of s$gather) or to and v (s$scatter). The other dimensional parameters in dimension dim may differ between the 3 arrays.

Vector v contains 'normalised global' subscript values for dimension dim of to (in the case of s$gather) or from (s$scatter). If any subscript value is not in range, the procedure writes an error message and halts the program.

Dimensional redistribution

```
SUBROUTINE  s$broadcast (to,   attr_to,         &
                         from, attr_from, dim)
SUBROUTINE  s$rotate    (to,   attr_to,         &
                         from, attr_from, dim)
SUBROUTINE  s$transpose (to,   attr_to,         &
                         from, attr_from, dim1, dim2)
```

These procedures copy array from into to, changing the process mapping of data dimension dim or, in the case of transpose, transposing the process mappings of data dimensions dim1 and dim2, as shown in the table below.

The other dimensional parameters may also be changed in that (those) data dimensions, although *sectn* is unchanged (as required for conformance). `to` and `from` have the same distribution in their other dimensions.

procedure	process mapping of data dimension *dim* or (*dim1*,*dim2*) array `from`	array `to`	conditions
`s$broadcast`	μ	0	$\mu \neq 0$
`s$rotate`	μ	ν	$\mu \neq \nu \neq 0$
`s$transpose`	(μ, ν)	(ν, μ)	$\mu \neq \nu$

Notes:

- If `s$broadcast` selects a single element in dimension `dim` of `from`, this dimension may be omitted from `attr_to` (i.e. `s$broadcast` may be a rank reducing function). In all other cases the rank is preserved.
- For `s$transpose`, one of the process dimensions (μ, ν) may be 0.

Sectional redistribution

```
SUBROUTINE s$resection (to,   attr_to,         &
                        from, attr_from, dim)
```

This procedure copies array `from` into `to`, performing a linear transformation from one regular section to another in data dimension `dim` *without* changing its process mapping, viz:

$$to(\ldots\ l' : u' : s'\ \ldots) = from(\ldots\ l : u : s\ \ldots)$$

The regular sections in dimension `dim` may select a single element or the whole dimension. In dimension `dim`, *sectn* is unchanged (as required for conformance), as is the process mapping, but the other dimensional parameters may be changed. `to` and `from` have the same distribution in their other dimensions.

Elemental broadcast

```
SUBROUTINE  s$broadcast_elem (elem, from, attr_from)
   ANYTYPE, INTENT (OUT) :: elem
```

This broadcasts a single element of array `from` globally over the process array, writing its value to the global scalar variable `elem`, which must have the same data type as `from`. Array `from` can have any rank, data type and distribution, but it must have just a single element selected, that is, it must have *sectn* $= \pm 1$ in every dimension.

Generalised re-distribution

```
SUBROUTINE s$redistr (to, attr_to, from, attr_from)
```

This is used to copy data from array `from` into array `to` when either or both have variable shape and/or process mapping, so that the appropriate redistribution scheme cannot be determined statically.

The procedure first checks that the *sectn* vectors of `from` and `to` are the same – that is, that the arrays (or array sections) conform – if not, it writes an error message and halts the program. Apart from this requirement, the dimensional parameters of `from` and `to` may be arbitrarily different, and the procedure copies the specified regular section of `from` into the specified regular section of `to`, performing all of the communications necessary (if any) to perform this operation.

A.3 Fortran 90 array intrinsic functions

The following ADLIB procedures correspond to the Fortran 90 array intrinsic functions (see, e.g. chapter 8 of [4]). Although they are called 'functions' here, they are actually implemented as subroutines, with storage for the result pre-allocated and passed-in as the first parameter. The purpose of these procedures, and the specification of their arguments and results, is given by the Fortran 90 standard.

The arguments correspond to those specified by the Fortran 90 standard, with the following additions:

- the first argument is always the `result` of the procedure call;
- every array argument (or potential array argument) is followed by an attribute vector, `attr_`*name*;
- an argument that is specified as *optional* in the Fortran 90 standard is normally followed by an extra argument of `LOGICAL` type called `present_`*name*.

Unless otherwise stated, there is no relation between the attributes of different array arguments except for relations between their types and shapes (i.e. *sectn* parameters) specified by the Fortran 90 standard.

Vector and matrix manipulation functions

```
SUBROUTINE s$dot_product (result,                &
                          vector_a, attr_a,      &
                          vector_b, attr_b )
SUBROUTINE s$matmul      (result,   attr_result, &
                          matrix_a, attr_a,      &
                          matrix_b, attr_b )
```

Array reduction functions

```
SUBROUTINE s$all    (result, attr_result,  &
                     mask, attr_mask,      &
                     dim, present_dim )
SUBROUTINE s$any    (result, attr_result,  &
                     mask, attr_mask,      &
                     dim, present_dim )
SUBROUTINE s$count  (result, attr_result,  &
                     mask, attr_mask,      &
                     dim, present_dim )
SUBROUTINE s$maxval (result, attr_result,  &
                     array, attr_array,    &
                     dim, present_dim )
SUBROUTINE s$minval (result, attr_result,  &
                     array, attr_array,    &
                     dim, present_dim )
SUBROUTINE s$product(result, attr_result,  &
                     array, attr_array,    &
                     dim, present_dim )
SUBROUTINE s$sum    (result, attr_result,  &
                     array, attr_array,    &
                     dim, present_dim )
```

Constraints: If `present_dim` has value `TRUE`, then `result` has the same distribution as `mask` or `array` excluding data dimension `dim`. Otherwise, `result` is a scalar.

Reshaping an array

```
SUBROUTINE s$reshape (result, attr_result,              &
                      source, attr_source,              &
                      shape,  attr_shape,               &
                      pad,    attr_pad,   present_pad,  &
                      order,  attr_order, present_order)
```

Replication function

```
SUBROUTINE s$spread (result, attr_result,   &
                     source, attr_source,   &
                     dim, ncopies)
```

Constraints: The distribution of `source` is the same as that of `result` with data dimension `dim` omitted. In dimension `dim` of `result`, *sectn* equals `ncopies`.

Array shifting functions

```
SUBROUTINE s$cshift  (result, attr_result,     &
                      array,  attr_array,      &
                      shift, dim)
SUBROUTINE s$eoshift (result, attr_result,     &
                      array,  attr_array,      &
                      shift,                   &
                      boundary, attr_boundary, &
                      dim)
```

Constraints:

- `array` and `result` have the same distribution.
- In procedure `eoshift`, `boundary` is either a scalar or an array whose distribution is the same as that of `array` with data dimension `dim` omitted.

Matrix transpose

```
SUBROUTINE s$transpose (result, attr_result, &
                        matrix, attr_matrix)
```

Transformational functions for geometric location

```
SUBROUTINE s$maxloc (result, attr_result,    &
                     array,  attr_array,     &
                     mask,   attr_mask, present_mask)
SUBROUTINE s$minloc (result, attr_result,    &
                     array,  attr_array,     &
                     mask,   attr_mask, present_mask)
```

Constraints: The `result` vector is global.

SNARC Technical Reports

Editor: David Pritchard

- [ ] 91-01 I. Glendinning (ed): *Load-balancing and Performance Monitoring for MIMD Computers*
- [ ] 91-02 I. Glendinning (ed): *The PUMA and GENESIS Projects*
- [ ] 91-03 D.L. Gee and A.J.G. Hey: *Parallel Universal Message-passing Architectures*
- [ ] 92-01 Ian Glendinning, Stephen Hellberg and Piers Shallow: *Tools for Parallel High Performance Systems – Comparative Evaluation*
- [ ] 92-02 John Merlin: *Techiniques for the Automatic Parallelisation of 'Ditributed Fortran 90'*

Computer Science Technical Reports (selected)

Editor: Andy Gravell

Design and Production: Sebastian Rahtz

- [ ] 89-2 Dave De Roure: *QPL2*
- [ ] 89-4 Hugh Glaser, Stuart Cox and Mike Reeve: *Compiling Functional Languages for Arrays of Transputers*
- [ ] 90-4 Andy King: *Granularity analysis of concurrent logic*
- [ ] 90-6 Andy King and Paul Soper: *Granularity control for concurrent logic programs*
- [ ] 90-7 Ken Thomas: *An Object Oriented CSP interpreter*
- [ ] 90-14 Mike Surridge: *ECCL: a general communication harness and configuration language*
- [ ] 90-17 Andrew Verden & Hugh Glaser: *Independent AND-Parallel Prolog for Distributed Memory Architectures*
- [ ] 90-19 Tony Hey: *Experience with MIMD Message Passing Systems — Toward General Purpose Parallel Computing*
- [ ] 90-20 David DeRoure: *QPL3 — Continuations, Concurrency and Communication*
- [ ] 90-21 David DeRoure: *Experiences with Lisp and Distributed Systems*
- [ ] 91-06 JM Bishop, MJ Hasling & KS Thomas: *Experience with Multi-transputer Ada*
- [ ] 91-07 Hugh Glaser & Pieter Hartel (Editors): *Proceedings of the Workshop on the Parallel Implementation of Functional Languages*

Order Form

Name: . . . .

Position: . . . .

Organisation: . . . .

Signature: . . . .

Date: . . . .

Address: . . . .

. . . .

. . . .

[ ] Please send me the previously issued reports that I have ticked overleaf.

[ ] Please send me the Project Directory *Parallel Computing at Southampton*.

[ ] Please add me to your list to be mailed with future newsletters

Correspondence concerning SNARC Techical Reports should be sent to:

Novel Architecture Research Centre
Department of Electronics and Computer Science
University of Southampton
SO9 5NH
United Kingdom

E-mail `snarc@uk.ac.soton.ecs` (Janet)

The Department of Electronics and Computer Science

The Department of Electronics and Computer Science was formed in 1986 as part of a major expansion in Information Technology within the University of Southampton. It was created by combining the Department of Electronics (which was established in 1947—the first such department in any UK University) and the Department of Computer Studies. It is the largest department of its kind with more than sixty academic staff including eight professors. Its research and teaching ranges includes the fabrication of integrated circuits, parallel processing and computing systems, optical electronics and communications, signal processing and radio communications, and program and algorithm design.

We were one of two Universities chosen by the Science and Engineering Research Council (SERC) to establish a microelectronics fabrication facility; this provides advice and integrated circuit fabrication facilities to industry and other Universities. We also host the Institute of Transducer Technology and a Transputer Support Centre. We have also recently been awarded an Interdisciplinary Research Centre (IRC) by the SERC. These aim to be major research centres covering scientific and technological disciplines and only 9 have been established so far within the UK. The Southampton IRC will concern optoelectronic technology and will be housed in the new Electronics building.

The department publishes its own Research Journal annually, with short articles featuring each of its research areas. If you would like to be included on the mailing list for this Research Journal, please write, giving full details, to:

The Research Journal Editor
The Department of Electronics and Computer Science
University of Southampton
SO9 5NH