DAP-Algol: A Development System for Parallel Algorithms

L. M. DELVES AND S. C. MAWDSLEY

Department of Statistics and Computational Mathematics, University of Liverpool

This paper describes a programming language and environment, 'DAP-Algol', which provides convenient facilities for developing, testing, and timing parallel algorithms for an SIMD machine such as the ICL Distributed Array Processor, and which can be run on any serial machine supporting the language (Algol 68) in which the system is coded. The facilities provided are modelled closely on those provided in DAP-FORTRAN, and so the system is particularly convenient as a development system for DAP programmes; however, the principles involved in providing the facilities apply equally well to other parallel languages of similar structure, such as ACTUS or FORTRAN 8X.

1. INTRODUCTION

With the introduction of vector processors (Cray 1, Cyber 205, AP 120b) and of multi-processor SIMD machines (ICL DAP) into routine user service, there is growing interest in the development of specialised (parallel) algorithms for such systems. This interest stems from the fact that a given algorithm may perform well in a serial environment but poorly on a parallel machine; an algorithm which runs well on the DAP may run poorly on the Cray 1, and vice versa; and in general the choice of a 'best' (i.e. fastest) algorithm for a given task may depend strongly on the machine on which it is intended to run it. The minimum testbed requirements for the development of a new parallel algorithm are: a language allowing ready expression of the algorithm; a compiler for this language; timing software to test the effectiveness of the code and a machine on which to run. At the present time, none of these requirements is easy to fill. Several 'parallel' languages have been developed; for example ACTUS;[1] DAP-FORTRAN;[2] while others already have a significant parallel and/or array – processing ability (Algol 68) or proposed ability (FORTRAN 8X). None of these languages provides an ideal medium of expression: ACTUS can deal only with one-dimensional parallelism; DAP-FORTRAN contains features specific to the ICL DAP and hence is not portable; while neither of these contains features suitable for MIMD machines. The other two languages are 'general purpose' rather than 'parallel'; and FORTRAN 8X, although promising to contain many features useful at expressing parallelism, is still rather far from being available. We return to Algol 68 below. It is often also difficult to provide a machine on which to test a proposed algorithm. This is especially true of the 'supercomputer' architectures: Cray 1, Cyber 205, DAP. Machines of this class have so far been sold in ones rather than hundreds; remote access, or no access, is the norm. One possible answer to these problems of language and machine availability is to provide some sort of simulator. This has been done, for example, for the ICL DAP, for which a complete bit-level simulator exists which will run on any ICL 2900 machine. However, such a simulator is likely to be slow: orders of magnitude slower than the direct execution of code on the machine hosting the simulator. It is also rather likely to be itself not portable. An alternative approach to portability is to embed, in an existing high-level language, features which are needed to express parallel algorithms. DAP-FORTRAN does this by adding new facilities to FORTRAN; but this cannot be done directly in a portable manner because FORTRAN was not designed to accept such extensions. This lack of portability can be overcome in one of two ways: either by writing a pre-processor for the language extensions, or by adding similar facilities directly to a language which was already designed to accept extensions. We take the latter course here, by describing briefly a package (a 'prelude' in the jargon of Algol 68) which adds DAP-FORTRAN-like facilities to Algol 68 and in addition includes a 'pseudo timer' which maintains a clock designed to return the timings which we would obtain were the code run on the current 64×64 production DAP. Because the language was designed to accept user-defined extensions, these extensions can be themselves written wholly in Algol 68; and this has three advantages:

(1) The extensions are portable;

(2) programs written in the extended language run as fast as those in the base language;

(3) the effort involved in producing the extensions is relatively small, and certainly much less than would be required to write a pre-processor of similar capabilities.

2. DAP-ALGOL

The extensibility of Algol 68 lies in the ability of the user to define new data types ('modes'), and to define new operators acting between variables of either new or existing types. A simple introduction to these facilities is given in Ref. 3; however, it is not necessary to know the language to read what follows, since this paper is intended to outline the principles involved and to demonstrate by example the use of the facilities, rather than to serve as a manual for DAP-Algol (for which see Ref. 4). In this section, then, we recall briefly the additional facilities introduced into FORTRAN by DAP-FORTRAN, and describe their equivalents in DAP-Algol.

2.1. Array operations

The most basic feature of the DAP is its ability to process whole arrays (of size up to 64*64) in parallel. DAP-FORTRAN reflects this ability by allowing the user to write whole-array operations, and operations on subsections of an array ('slices'). Table 1 gives examples, together with the equivalent code in DAP-Algol. We make the following comments on these examples:

(1) Apart from trivial representational differences (: = for assignment, square brackets for suffixes), the facilities are the same in the two languages.

(2) Whole array assignment, and the ability to refer to or to name a slice of an array, are already part of the Algol 68 language. Arithmetic operations between arrays are not predefined, but the ability to define them is there. Thus, in the last four lines of Table 1, the operators *, +, − and / have been defined as extensions to the language. The Algol 68 code defining the operation * between two real matrices is given in Table 2 to illustrate how simple it is to do. (Non-Algol 68 users need not read this table; those who are expert in the language will note that checks on the input parameters to the operator, which are included in the actual code, have been omitted in Table 2 for simplicity.) We note that the operation is defined as pair-wise multiplication between matrix elements, rather than as an algebraic multiplication of the two matrices, because that is how it is defined in DAP-FORTRAN. Note also that the result of the multiply operation is another matrix, space for which is generated automatically; this ability to generate storage as required, and to define functions returning matrix-valued results, is crucial to the DAP and to DAP-FORTRAN; it is already present in Algol 68. Similar facilities are available for Integer and Logical matrices, and for one-dimensional vectors.

(3) The DAP-Algol operators are defined for any (compatibly sized) matrices or vectors. A global variable DAPSIZE is provided for compatibility with DAP-FORTRAN, with a default value of 64 (resettable by the user!); but there is no restriction to the use of DAPSIZEd vectors and matrices. (The restriction in DAP-FORTRAN is imposed to make efficient implementation on the DAP easier.)

2.2. Systems functions

DAP-FORTRAN provides a large number of operations via 'system functions': pre-defined functions which can be called by the user. These are all very useful, but from the point of view of the present paper not very interesting: we merely have to provide routines which carry out the same tasks, in Algol 68. The only feature of note is that DAP-FORTRAN allows the 'overloading' of procedure names; that is, a given procedure name can refer to two or more procedures which expect different types of results. This facility allows, for example, the same name

Table 1. Array facilities in DAP-FORTRAN and DAP-Algol. A, B, C are assumed to be real or integer matrices, U, V are one-dimensional vectors

Operation	DAP FORTRAN	DAP Algol
Array assignment	A = B	A: = B
Array addition	A = B+C	A: = B+C
Array multiplication	A = B*C	A: = B*C
Slicing a row	U = V+A (I,)	U: = V+A[I,]
Slicing a column	U = V+A (,I)	U: = V+A[,I]

Table 2. The Algol 68 code required to define the operation '*' between two real matrices

```
OP* = (REF [,] REAL a,b)REF [,] REAL:
BEGIN
    £ Experienced Algol 68 users will note that, here
    and elsewhere in this paper, we used 'Reserved
    Word Stropping' for Algol 68 keywords. They will
    also note that this coding assumes that the two
    arguments have the same bounds, with a lower
    bound of one. In the DAP-Algol 68 prelude this is
    checked for by the code. This checking of the
    bounds is omitted here for simplicity £
    INT n = 1UPB a;
    HEAP [1:n,1:n]REAL result;
    FOR i TO n DO
      FOR j TO n DO
        result [i,j]: = a[i,j] * b[i,j]
      OD
    OD;
    result £ returns the result matrix £
END £ of operator definition £
```

to be used for the procedure to sum the elements of a real matrix (SUM) as for the versions to sum the elements of an integer or logical matrix, or of a real integer or logical vector. Such procedures are said to be 'generic'. In Algol 68 procedures cannot be generic, but operators may; most of the system functions have therefore been implemented as generic operators. This leads to a notational difference for two parameter generic functions; which are accessed in DAP-Algol using the infix notation:

	function (DAP-FORTRAN)	Operator (DAP-Algol)
one parameter:	fix(im)	fix(im) or fix im
two parameters:	and(lm, ln)	lm and ln

A partial list of system operators and functions containing some of the more commonly used ones, is given in Table 3; a full list is given in Ref. 4. Note that the last entries in this table relate to a limitation in Algol 68: Operators can only have two arguments so that three-argument generic facilities are not available. Thus the three-argument function MERGE cannot be provided as a single operator. We provide two alternative forms in DAP-Algol; the first is as a set of procedures with different names for each type of argument (bvmerge, bmmerge, rvmerge, rmmerge); the alternative form, which is fully generic, uses the two operators MERGE and MASK, with the reasonably comfortable syntax

answer = arg1 MERGE arg2 MASK arg3

The operator MASK appears again later in Section 2.5.

2.3. Subscripting facilities

In DAP-FORTRAN, the FORTRAN concept of a subscript is generalised. In addition to the traditional use to specify a particular element of a vector or a matrix

A(i,j) ; V(i)

it is possible to specify a row or a column, and to provide integer and logical vectors and matrices as suffixes. The resulting facilities are extremely useful for specifying quite general loops without having to introduce an explicit DO loop. It is not possible within Algol 68 to use the

Table 3. A partial list of system functions. The allowable types of argument are indicated as follows: s = scalar, v = vector, m = matrix, e = any of these; r = real, i = integer, l = logical, a = any of these

DAP-FORTRAN	DAP-Algol	Type of result	Comments
ABS(ae)	ABS(ae)	same as argument	
EXP ATAN SIN SQRT COS LOG			are also provided
FIX(re)	FIX(re)	ie	
FLOAT(ie)	FLOAT(ie)	re	
ALL(le)	ALL(le)	ls	logical AND of components
ANY(le)	ANY(le)	ls	logical OR of components
AND(le,le)	le AND le	le	logical AND of components
NOT(le)	NOT(le)	le	logical NOT of components
MERGE(am,am,lm))		am	
MERGE(av,av,lv)		av	merges the first two
BVMERGE(lv,lv,lv)		lv	depending on the value
BMMERGE(lm,lm,lm)		lm	of the third argument
RVMERGE(rv,rv,lv)		rv	
RMMERGE(rm,rm,lm)		rm	
or am MERGE am MASK lm		am	
av MERGE av MASK lv		av	

DAP-FORTRAN syntax as it stands; however, it is straightforward to define new operators SUB, SUBR, SUBC which accept logical and integer vector and matrix arguments, and perform the same selecting actions as the 'extended suffixing' provisions of DAP-FORTRAN. A list of the facilities is given in Table 4, together with their DAP-Algol equivalent.

2.4. Shift facilities

In DAP-FORTRAN there are facilities which enable users to perform datashifts on vector and matrix values. These shifts are performed by a set of pre-defined functions, each of which shifts either vector or array data between processors in the DAP processor array, in a horizontal ('EAST/WEST') or vertical ('NORTH/SOUTH') direction. The length of the shift is a parameter of the function; a simpler facility ('shift-indexing' – see below) is provided for shifts of length 1. Shifting a DAP-sized row of data one place to the right (say) introduces a blank in position 1, and shifts the rightmost data element out of the DAP processor array. How these edge effects are treated depends on the setting of what DAP-FORTRAN refers to as the GEOMETRY:

CYCLIC GEOMETRY: data shifted out are wrapped round and shifted back in at the other end.
PLANE GEOMETRY: data shifted out are lost; zeros are shifted in at the other end

At any time there is a standard geometry, set separately (and resettable) for the N–S and E–W direction. The shift operations themselves either impose an explicit temporary geometry, or use the default geometry. The facilities provided are listed in Table 5, with DAP-Algol equivalents; the naming convention adhered to is the following. Shift operator names start with the letters SH. The direction of the shift, and the mode of the argument is given by the next letter of the operator name. N, S, E & W, which stand for North, South, East and West respectively, are matrix shifts, and L & R, which stand for Left and Right, are vector shifts. The last letter of the operator name gives the type of geometry used by the shift. If the shift name ends with a P the shift uses Planar geometry, while if the shift name ends with a C the shift uses Cyclic geometry. DAP-FORTRAN also provides Left and Right shifts for matrices; these treat the matrix as a long vector. This concept has not been mimicked by DAP-Algol, and is not necessary since the vector shift operations in DAP-Algol work for any length vector.

DAP-FORTRAN also permits the use of + and − as array subscripts to indicate a shift. This facility is included in DAP-Algol by defining operators NORTH, SOUTH, EAST and WEST. The geometry of these shifts is given by the current geometry, as set in the global boolean variables NSGEO and EWGEO. These can be altered at any time by either a straightforward assignment of the form:

NSGEO: = PLANE (or CYCLIC)

or by calling a procedure GEOMETRY which takes a boolean vector argument of any size and sets either the NS, or both the NS and EW geometries (see Ref. 4).

2.5 Masked assignments

The concept of a 'Logical Mask' is a very important one in DAP-FORTRAN. A logical mask is a matrix of logical

Table 4. Subscript facilities available in DAP-FORTRAN and DAP-Algol 68. (LV is a logical vector, LA a logical matrix, IV an integer vector)

DAP-FORTRAN	DAP-Algol 68	Meaning
A(,i)	A [,i]	iTH column of A
A(i,)	A [i,]	iTH row of A
A(LA,)	A SUBR LA	{These forms each
A(,LA)	A SUBC LA	return of vector
A(,IV)	A SUBR IV	whose components
A(IV,)	A SUBC IV	are a selection
V(LV)	V SUBR LV	from the elements
	V SUBC LV	of A or V. For details,
or	V SUB LV	see [4]}

Table 5. Shift operators. The allowable types of argument are as follows: s = scalar, v = vector, m = matrix, a = any of these, r = real, i = integer, b = boolean, e = any of these

DAP-FORTRAN	DAP-Algol*	Operation
SHNC(me,is)	me SHNC is	Shift North Cyclic
SHNP(me,is)	me SHNP is	Shift North Planar
SHSC(me,is)	me SHSC is	Shift South Cyclic
SHSP(me,is)	me SHSP is	Shift South Planar
SHWC(me,is)	me SHWC is	Shift West Cyclic
SHWP(me,is)	me SHWP is	Shift West Planar
SHEC(me,is)	me SHEC is	Shift East Cyclic
SHEP(me,is)	me SHEP is	Shift East Planar
SHLC(ve or me,is)	ve SHLC is	Shift Left Cyclic
SHLP(ve or me,is)	ve SHLP is	Shift Left Planar
SHRC(ve or me,is)	ve SHRC is	Shift Right Cyclic
SHRP(ve or me,is)	ve SHRP is	Shift Right Planar
A(+,)	A NORTH 1 or NORTH A	
A(−,)	A SOUTH 1 or SOUTH A	suffixed elements
A(,+)	A EAST 1 or EAST A	move in the stated
A(,−)	A WEST 1 or WEST A	direction with the
V(+)	V EAST 1 or EAST V	default geometry
V(−)	V WEST 1 or WEST V	
A(−,+)	(A SOUTH 1) EAST 1	etc. etc.

* Note. Our current implementation of DAP-Algol does not include integer vector or matrix shifts.

values which is used to determine which of the DAP processors shall be active during a given operation; examples of the use of such masks during subscripting operations are given in the previous section. Equally important is their use during assignments: it is very common to find that time can be saved by computing a whole matrix of values, and then throwing away the unwanted ones during the assignment of the results to storage. In DAP-FORTRAN the syntax for such a masked assignment is not distinguished from that for masked suffixing; the difference is detected by the compiler from the context. We believe that this double use of the suffix notation is not desirable, and originally we used an alternative syntax for this feature which emphasises the difference. In DAP-FORTRAN, a masked assignment takes the form:

A(MASK) = matrix-expression

which assigns the values of the components of the matrix-expression to the elements of the matrix A, but only for those elements for which the corresponding element of the logical matrix MASK is TRUE. We associate the mask with the right-hand side of this statement by introducing the operator MASK:

expression MASK logical-mask.

With this notation, we would write a masked assignment in the form:

a: = expression MASK logical-mask

However, this assignment statement cannot be implemented within Algol 68; the result of the operation 'a MASK logical-mask' is a structure containing the information necessary for the assignment, but one is not allowed by the language to give a new or extended meaning to the assignment symbol : =, which is treated as a primitive operation and not as an operator. We therefore find it necessary to introduce a new operator BECOMES and to write the assignment in the form:

a BECOMES expression MASK logical-mask

which does the required job but is certainly more verbose than the DAP-FORTRAN equivalent. The introduction of BECOMES also has the side effect that it is possible to provide a version which accepts the alternative syntax:

A MASK logical-mask BECOMES expression

which is rather closer to the DAP-FORTRAN syntax. Our current implementation does in fact accept either form of the assignment. We note that the operator MASK is identical with that introduced in section 2.2. to describe the MERGE operation

DAP-FORTRAN also accepts the constructs:

DAP-FORTRAN	DAP-ALGOL
A(IV,): = expr	A MASK COL(IV) BECOMES expr
A(,IV): = expr	A MASK ROW(IV) BECOMES expr
A(LV,): = expr	A MASK COL(ELN LV) BECOMES expr
A(,LV): = expr	A MASK ROW(ELN LV) BECOMES expr

and other variants. In these variants, the suffix LV or IV is first implicitly expanded to the logical matrix mask, and hence, as shown, all of these can be easily written in DAP-Algol using operators already introduced. Since these constructs seem less often used, we have not provided special operators for them (though we certainly would if the constructs A(LV,) or A(,LV) were often needed!)

2.6. Other features

DAP-FORTRAN contains other facilities (see Ref. 2). Most of these have close equivalents in DAP-Algol (see Ref. 4), with the following exceptions.

(1) Variable length reals and integers are not supported.

(2) The debug facilities are not supported.

(3) There is no equivalent of a FORTRAN COMMON block, and the distinction between DAP and HOST code is not maintained. Therefore, the conversion routines between DAP and HOST formats are not mimicked, although they could be provided in dummy form so that the conversion time would be taken into account in the psuedo-timer.

3. EXAMPLES

DAP-Algol can be run on any machine for which an Algol 68 compiler is available (this does not currently include the DAP itself). Its purpose as a vehicle for developing parallel algorithms requires that it feel as comfortable in use as does DAP-FORTRAN; and that the pseudo-timer returns reasonably accurate DAP timings. We illustrate the correspondence between DAP-Algol and DAP-FORTRAN and the use of the pseudo-timer, with two short examples.

The first solves a system of N equations of the form

$$Ax = b$$

for N < = 64, using the Gauss-Jordan method (the 'recommended' method for full matrices on the DAP).

DAP-FORTRAN version, taken from Ref. 5.

```
  REAL VECTOR FUNCTION GAUSS-JORDAN (A,B,N)
  DIMENSION A(,) B( )
  REAL MULT ( ),S
  DO 1 I = 1,N
  S = A(I,I)
  MULTS = A(,I)/S
  A(.NOT.ROW(I)) = A – MATR(A(I,))*MATC(MULTS)
  B(.NOT.EL(I)) = B – B(I)*MULTS
  A(I,) = A(I,)/S
1 B(I) = B(I)/S
  GAUSS-JORDAN = B
  RETURN
  END
```

This is of course a rather simplified program; it was included in Ref. 5 as a programming example rather than a 'production solver'. Note that the solution is returned as the vector value of the function GAUSS-JORDAN; the ability to return vectors and arrays is an essential part of DAP-FORTRAN.

DAP-Algol version

```
PROC GAUSS-JORDAN = (MATRIX A, VECTOR B,
INT N) VECTOR:
BEGIN
    [1:N] REAL MULTS; REAL S;
    FOR I TO N
    DO
      S: = A[I,I];
      MULTS: = A[,I]/S;
      A MASK (NOT ROW (I))
           BECOMES A-MATR(A[I,])*MATC(MULTS);
        B MASK (NOT EL(I)) BECOMES B-B[I]*MULTS;
      A[I,]: = A[I,]/S;
      B[I]: = B[I]/S
    OD;
    B
END;
```

This example shows how closely the facilities in DAP-Algol correspond to those in DAP-FORTRAN; the two codes correspond on a line-by-line basis. For those not used to Algol 68: the keywords DO...OD delimit the FOR loop, and the apparently 'floating' B at the end represents the Algol 68 mechanism for returning a value; this line corresponds to the line GAUSS-JORDAN = B in the Fortran code.

The second example compares two programs which perform a bubble sort on $N <= \text{dapsize}\uparrow 2$ positive elements. For DAP-FORTRAN, these are assumed provided in the first N locations of a DAPSIZE × DAPSIZE matrix and padded out with zeros.

DAP-FORTRAN version, taken from [6]

```
   REAL MATRIX FUNCTION BUBBLE (VALUE)
   REAL VALUE(,)
   LOGICAL MSK(,),CHANGE(,)
   MSK = .NOT.ALTR(1)
1  CHANGE = VALUE.LT.VALUE(+)
   IF(.NOT.ANY(CHANGE)) GOTO 10
   CHANGE = CHANGE.AND.MSK
   CHANGE = CHANGE.OR.CHANGE(–)
   VALUE(CHANGE) =
                 MERGE(VALUE(+),VALUE(–),MSK)
   MSK = .NOT.MSK
   GOTO 1
10 BUBBLE = VALUE
   RETURN
   END
```

DAP-Algol Version

```
PROC BUBBLE = (VECTOR VALUE) VECTOR:
BEGIN
  [1: UPB VALUE] BOOL MSK,CHANGE;
  MSK: = NOT ALT(1);
  CHANGE: = TRUE;
  WHILE ANY(CHANGE)
  DO
    CHANGE: = VALUE < WEST VALUE;
    CHANGE: = (CHANGE AND MSK);
    CHANGE: = CHANGE OR EAST CHANGE
    VALUE MASK CHANGE BECOMES
        RVMERGE(WEST VALUE,EAST VALUE,MSK);
    MSK: = NOT MSK
  OD;
  VALUE
END;
```

Again the correspondence between the two codes is very close, with the exception that we have used a WHILE loop in DAP-Algol rather than the GOTOs of DAP-FORTRAN. Note however that the DAP-FORTRAN code assumes that the data to be sorted is in a DAP matrix, which it then treats as a vector ('long-vector' in DAP notation). This standard trick will be familiar to all FORTRAN users, but comes as a surprise to Algol programmers; in DAP-ALGOL, arbitrary length vectors are accepted and the data required can therefore be provided in a vector.

4. TIMINGS

As mentioned above, DAP-Algol includes a pseudo-timer; a global variable called TIME is updated whenever any of the DAP-Algol procedures or operators are entered, by the time taken for the equivalent facility on the DAP. Interrogating the timer then allows estimates of the speed of the corresponding DAP code. For the examples given here, with DAPSIZE = 64, we obtain the times given in Table 6. We see that for example 1 the pseudo-times are about 10% too fast; for example 2 about 10% too slow. This is quite good enough accuracy to compare algorithms, since a 10% change in the speed of an algorithm is rarely significant in practice. The actual measured DAP times themselves are not completely

Table 6. Timing results. DAP-FORTRAN times obtained on the ICL DAP at QMC, London; DAP-Algol, run on the ICL 1906S at the University of Liverpool

Code	DAP-FORTRAN time	DAP-Algol pseudo-time
Gauss-Jordan, N = 4	5.10	4.48
8	10.20	8.96
16	20.40	17.92
32	40.80	35.84
64	81.60	71.68
Bubble sort, N = 256	79.15	86.032
512	158.30	172.048
1024	316.60	344.080
2048	633.20	688.144
4096	1266.40	1376.272

consistent; the same program run a number of times can give a fluctuation of between 5 and 10%.

5. CONCLUSIONS

The facilities provided by DAP-Algol are, as the examples show, sufficiently close to those in DAP-FORTRAN that it is possible to develop DAP algorithms quite naturally in DAP-Algol and then translate line-by-line. The development is in practice aided considerably by the relatively good accuracy of the pseudo-timer.

We achieve this timing accuracy because most DAP programmes consist mainly of calls to array features, which are trapped by the pseudo-timer, and have relatively few sections of 'serial' code in them which are replaced in DAP-Algol by standard Algol 68 and hence are not timed. This in turn reflects the success of DAP-FORTRAN in expressing the operations which are needed for parallel processes on this type of machine.

We believe the success of the exercise can be interpreted a little more generally. The success depends upon the suitability of Algol 68; if FORTRAN had been similarly suitable, then DAP-FORTRAN would itself be portable. As machine architectures proliferate, with each having its own strengths and hence suggesting its own high-level language features to utilise these strengths, we need either local and non-portable extensions of FORTRAN, or a widely used and suitable extensible language. The authors hope that FORTRAN 8X will include not only built-in array facilities suited for current SIMD machines, but sufficient extensibility to cope with architectures which the language designers do not know about in advance.

Acknowledgements

The research reported here was funded in part by the U.S. Army European Research Office via grant No DAJA45-82-C-0010; and by the U.K. Science and Engineering Research Council via grant no. GR/A48990 to the University of Liverpool.

REFERENCES

1. R. H. Perrott, ACTUS. *ACM Transactions in Programming Languages*, TOPLAS (1979), 177.
2. DAP-FORTRAN Manual ICL Technical Publication 6918.
3. L. M. Delves, *ALGOL 68 For FORTRAN programmers*. Preprint, Department of Statistics and Computational Mathematics, University of Liverpool (1983).
4. S. C. Mawdsley, *DAP-ALGOL Users Manual*, Internal Report, Department of Statistics and Computational Mathematics, University of Liverpool (1983).
5. R. W. Gostick, *Introduction to DAP-FORTRAN*. ICL Document Number AP 20 (1978).
6. R. W. Gostick, Software and Algorithms for the Distributed Array Processor. *ICL Technical Journal* 116–135 (1979).