

ACLE: a Software Package for SIMD Computer Simulation

O. G. PLATA, J. D. BRUGUERA, F. F. RIVERA, R. DOALLO AND E. L. ZAPATA

Department of Electronics, Faculty of Physics, University of Santiago de Compostela, 15706 Santiago de Compostela, Spain

This paper describes ACLE (Array C Language Emulator), a software package comprising an ACLAN-to-C translator and a library of simulation routines enabling the execution of programs written in ACLAN to be simulated on a conventional sequential computer. Array C LAnGuage (ACLAN) is a machine-independent programming language that extends C by endowing it with structures for programming array processors. ACLAN was successfully proven by developing many parallel algorithms for hypercube computers. An algorithmic solution for mapping algorithms onto these computers is explained.

Received July 1989

1. INTRODUCTION

The fundamentals of parallel programming are familiar to experienced programmers because most of the basic operations are part of today's sophisticated uniprocessor operating systems. Nevertheless, some methodologies are unique to parallel programming. The basic challenge that confronts the parallel programmer is how to transform the structure of the program at hand so that it matches the structure of the parallel computer that will run that program. To map a problem onto a parallel architecture, the programmer must first divide the problem into segments that will execute in parallel, and then determine how the processors will communicate and synchronize with one another.

According to Howe and Moxon,¹ the key to programming a parallel problem is determining the granularity, or level of parallelism which indicates how much computing each processor can do independently in relation to the time it must spend exchanging information with other processors. A coarse-grained application can be divided into logical parts made up of long independent processing sequences, with little synchronization or communication. On the other hand, in the fine-grained application, fewer instructions are executed between communication steps.

In this paper, we describe ACLE (Array C Language Emulator), a simulation package for executing ACLAN programs on sequential computers. ACLAN (Array C LAnGuage) is an array processor programming language, and we analyze the algorithm embeddability problem on hypercube computers.

2. THE PARALLEL PROGRAMMING LANGUAGE

ACLAN,² an array processor programming language, has been based on C³ because of C's power, popularity, portability and combination of the structured programming features characteristic of high-level languages with the provision of low-level operators and data types. There are two kinds of executable statements in ACLAN, scalar and parallel instructions. Scalar instructions process universal data or control program flow; they are executed in the control unit (CU) of the array processor and their form in ACLAN is just the same as in C. Parallel instructions which process the data

distributed among the processing elements (PEs) of the array, or the distribution of these data, are executed in the PEs, and are written using ACLAN extensions to C.

In most extensions of sequential languages for parallel processing, such as Actus,⁴ Actus-2,⁵ Latin,⁶ Vector C,⁷ Vectran,⁸ Parallel Pascal⁹ or Fortran 8X,¹⁰ parallel instructions are introduced by defining *vector* data types and modifying the existing sequential operators and syntactic structures so as to handle vector variables; the location of the variables in memory is controlled by the compiler, not by the programmer. In consonance with C's low-level features, ACLAN handles parallel execution by introducing new low-level operators and data types allowing optimal programming of algorithms in which runtime or occupied memory are critical factors to optimize.^{11,12}

For example, operators are provided which control local (intra-PE) and inter-PE data transfer, and the programmer can specify the PE registers or local memory locations in which data are to be stored. In spite of this low-level capability, ACLAN is completely machine-independent. This independence is based on the virtual array processor concept. Table 1 shows the basic parallel structures of ACLAN.

2.1 Declarative statements

The declaration syntax and meaning of the program variables depend on the context. Variables used exclusively in scalar instructions and hence only in the CU (called *scalar variables*), can have any of the types allowed by standard C. Variables that are used exclusively in parallel instructions, and which therefore refer to PE local memory (*parallel variables*), are declared in the virtual array processor in accordance with the restrictions that will be discussed in Section 3.1.1. *Hybrid variables* are the third kind of variables and are used in both scalar and parallel instructions. They represent universal data stored for economy in the CU and broadcast when necessary to the PEs, and their declaration is restricted to the basic data types, **char**, **int**, **float** and **double** (together, where appropriate, with the adjectives **unsigned**, **short** and **long**), since it is PE registers of these types that must receive them.

Table I. Basic parallel structures of ACLAN

Symbol	Meaning	Explanation
<code>:=</code>	Local assignment	Data transference between memory components belonging to the same node.
<code>==:</code>	Local interchange	Data swapping between memory components belonging to the same node.
<code><--</code>	Remote assignment	Data transference between memory components belonging to directly connected nodes.
<code><--></code>	Remote interchange	Data swapping between memory components belonging to directly connected nodes.
<code><==</code>	Central assignment	Data transference from host to nodes or viceversa.
<code>./:/</code>	Bit operator	Extract a bit or range of bits. (<code>//</code> means optional).
<code>in ./:/</code>	Set operator	Check if a value is in a certain range.
<code>#</code>	Parallel register	Identification of the index of the node.
<code>neigh[]</code>	Predefined vector	Identification of the interconnection functions.

2.2 Parallel executable statements

Parallel instructions in ACLAN are composed of two fields and conform to the following general syntax,

action {*mask*};

where the *mask* field is an optional expression and the *action* field represents the action to execute. Each PE, in parallel with the rest of the PEs, first evaluates the *mask* field, if present; if the *mask* is present and evaluates to zero the PE inhibits itself, otherwise the PE performs the *action*.

Expressions in ACLAN are basically similar to C expressions, though with certain logical restrictions and the provision of two extra operators. The allowed variables are limited to parallel and hybrid ones. The allowed operators are a subset of standard C operators, as follows, `()`, `!`, `~`, `-` (unary), `+` (unary), `*`, `/`, `%`, `+`, `-`, `<<`, `>>`, `<`, `<=`, `>`, `>=`, `==`, `!=`, `&`, `&`, `||`, `&&` and `!|`, together with the bit operator (`./:/`, where `//` means optional), used to extract a bit or a range of bits from an integer operand, and the set operator (`in ./:/`), used to check whether the value of a specified expression is in a certain range. For example, if a PE has the local memory structure of figure 2, the expressions `A.5:1` and `MASK1.3` extract the value contained in the register *A* from the bit 5 to the bit 1 (bit 0 is the LSB) and the most significant bit of the register *MASK1*, respectively. `#.0` returns 0 in even PEs and 1 in odd PEs (the symbol `#` is used to represent the logical identification register of the PEs). The expression `R in 3.6:24` tests if the content of the register *R* belongs to the real range [3.6,24] and the expression `# in 0:10:2` evaluates to 1 in the first six PEs with even logical indexes.

The memory components involved in a parallel action are specified by means of a multiple (parallel) expression. A multiple expression, that can return several values, is an expression (with the above mentioned restrictions) that can contain one or more multiple operands. A multiple operand is a local memory component (parallel variable) with a number of specified dimensions lower than the number declared, and/or with one or several dimensions ranging from two limits. These limits are specified as follows

initial_limit:final_limit:increment

where *final_limit* and *increment* are optional (by default, *initial_limit* and 1 are taken as values for *final_limit* and *increment*, respectively). For example, if the matrix `T[10][20]` is part of the local memory structure, then the multiple operands `T`, `T[3]` and `T[*][11]` represent the complete matrix, its fourth row and its twelfth column.

2.2.1. The action field of parallel instructions

All the actions specified by parallel instructions consist of value assignments. There are three kinds of actions, local assignments involving only the memory components of a single PE, remote assignments (routing assignments) transferring data among local memory components of different PEs, and central assignments involving data transfer among the CU and the PEs in either direction. In the examples that will follow in this section, we consider that the PEs have the local memory structure shown in figure 2 (see definition in Section 3.1.1.).

Local assignments. These assignments are expressed by means of the local operator `:=`, which assigns the value of a parallel expression to a local memory component, and the local swap operator `==:`, which exchanges the values of local memory components. Examples of valid local assignments include

`R := (RAM[4] & 7 + S)*exp(h-3);`

and

`RAM[2] :=: S {#.0 && # != 1};`

where *h* is a hybrid variable broadcast by the CU and `exp()` is an ALU operation of the PEs. The second of the examples is only executed by PEs with odd logical indexes except PE 1. The local action

`RAM[0:249] := RAM[250:499];`

makes a copy of the second half of the local *RAM* and stores it in the first half (all PEs execute this action with their local *RAM*s).

Remote assignments. The remote (routing) assignments are expressed by means of the one-way routing operator

$\langle \leftarrow$ and the two-way routing operator $\langle \leftrightarrow$). With the first operator, each PE evaluates a parallel expression and stores the value(s) in the data transfer buffer, whence it(they) is(are) sent to one of its neighbouring PEs identified by an interconnection function. The two-way routing operator expresses the swapping of the contents of local memory elements belonging to different PEs (but neighbours). The interconnection function that expresses the communication lines used in a message transference is specified by means of the predefined vector **neigh**[*i*]. For example, **neigh**[3] indicates that we use the interconnection function of number 3 (see definition of the interconnection network in Section 3.1.2).

As an example, consider a 6-dimensional hypercube network that is defined by assigning to the interconnection function number *k* the communication lines along the *k*-th dimension. The one-way routing assignment

```
RAM(neigh[0]) <-- RAM {!#.0};
```

makes a copy of the local RAMs of all the even PEs on the local RAMs of all the odd PEs (PE $2*i$ sends its local RAM to PE $2*i + 1$, for $i = 0, 1, \dots, 31$). If the mask is missing, then the even PEs interchange their local RAMs with the odd ones (PE *i* with PE *i* + 1). This last action can also be expressed by means of the two-way remote action

```
RAM(neigh[0]) <--> RAM {!#.0};
```

Central assignments. Central assignments are rather more complicated, and are specified by means of the single central assignment operator $\leq =$, that is used both to load local memory from the CU (distribution) and to report to the CU from local memory (recollection). As an example of the former use, consider a hybrid variable declared in the CU as follows

```
short int data[10][499];
```

The following central action of distribution,

```
RAM <= data;
```

tries to distribute the *data* matrix on the RAM local vectors of all the nodes. This is to say the content of *data*[0][0] is sent to PE 0 and stored in the RAM[0] element. The next value, *data*[0][1] is sent to the same node and stored in the RAM[1] element. The end of the distribution action is controlled by the receiver variable. Therefore, the content of *data*[9][499] is sent to PE 9 and the next action is to send again the content of *data*[0][0] but to PE 10. In general, the content of *data*[*i*][*j*] is sent to PE *i* and PE *i* + 10 and stored in the RAM[*j*] element.

The mechanism and control of data transfer in the reverse direction, from the PEs to the CU, is similar.

3. THE SIMULATION PACKAGE

The simulation package ACLE (Array C Language Emulator) allows a program written in ACLAN to be executed on a sequential computer. It consists basically of a translator, a library of simulation routines and virtual array processor block (see Figure 1). A source program written in ACLAN, which in an array processor would be completely stored in the CU or part of the program in the CU and the rest in the PEs, is converted by the translator into a standard C program, a process parameterized by the virtual array processor. During translation, statements declaring scalar and hybrid variables are left unchanged, except that a table of hybrid identifiers is constructed, while parallel variables declared as part of the processors definition are treated by conventional compiler methods during processing of this block. Scalar executable statements are also left intact, while parallel executable statements are analysed lexically and syntactically and replaced by calls to the library routines simulating the particular action involved in the statement.

3.1 Virtual array processor

For efficient array processor programming, the programmer must have the possibility of direct control over intra-PE operations and local memory and inter-PE

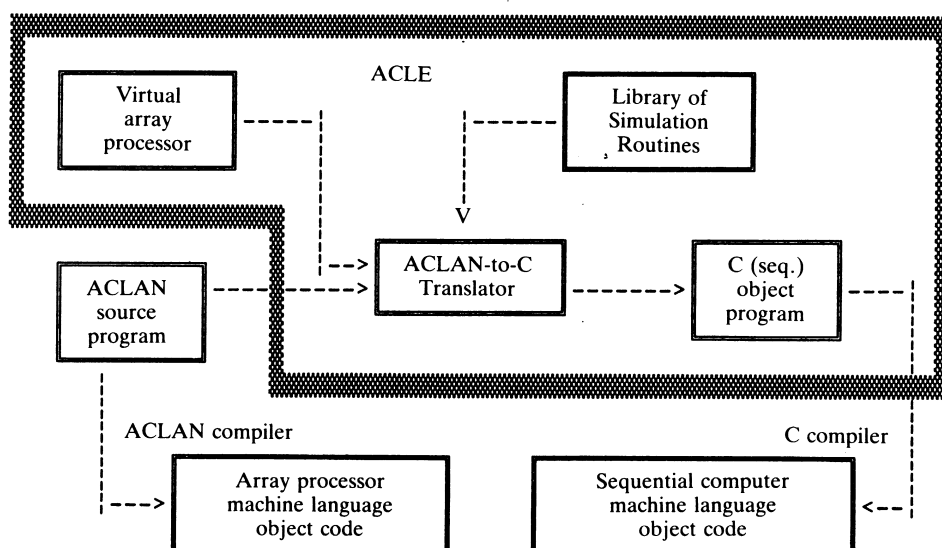


Figure 1. Structure of ACLE

communications. The programmer will in fact work with a conceptual scheme of the inter-PE communication network and the intra-PE structures that are accessible to him. This conceptual scheme constitutes a virtual array processor. In order to allow programs to be written which can be run on any machine, the programmer can define the virtual array processor he has in mind. Therefore, if the programmer wants to implement ACLAN on a particular array processor, he only needs to define the virtual array processor with the specific features of the array processor in question (apart from the machine-dependent ACLAN compiler).

3.1.1 Intra-PE definitions

Intra-PE definitions are of two kinds, definitions of local memory (including internal registers) and definitions of ALU operations. All PEs are assumed identical, so there will be just a single set of each kind.

Local memory definition. Local memory is defined by specifying, for each required memory component, an identifier associated with a data type indicating the kind of values to be held in the component. The programmer manipulates local PE memory by working with the appropriate identifiers. The local memory definitions conform to the syntactic rules of C.

In this section, we define all the local memory components required for each PE, which comprise registers (single-value storage elements) and arrays (storage elements comprising multiple values of the same data type, destined to hold the data distributed to the PEs for processing and which are presumably implemented by means of the local random-access-memory). Given the level at which we are working, the only data types allowed are **char**, **int**, **float**, **double** – all optionally together with the adjectives **unsigned**, **short** or **long** (with the same restrictions as in C) – and the new data type **bit** (as each identifier is bound to a memory element, the only admissible storage class is **external**,

which is accordingly assumed by default and need not be specified). **Bit** allows a memory element of a given number of bits to be defined. This data type is useful for defining mask and flag registers by means of declarations such as

```
bit MASK, CCR[12], COMPMASK:6;
```

which specifies a single-bit mask register (*MASK*), an array of 12 single-bit registers (*CCR*) and a 6-bit mask register (*COMPMASK*). Note that in practice the only difference between

```
bit VAR:n;
```

and

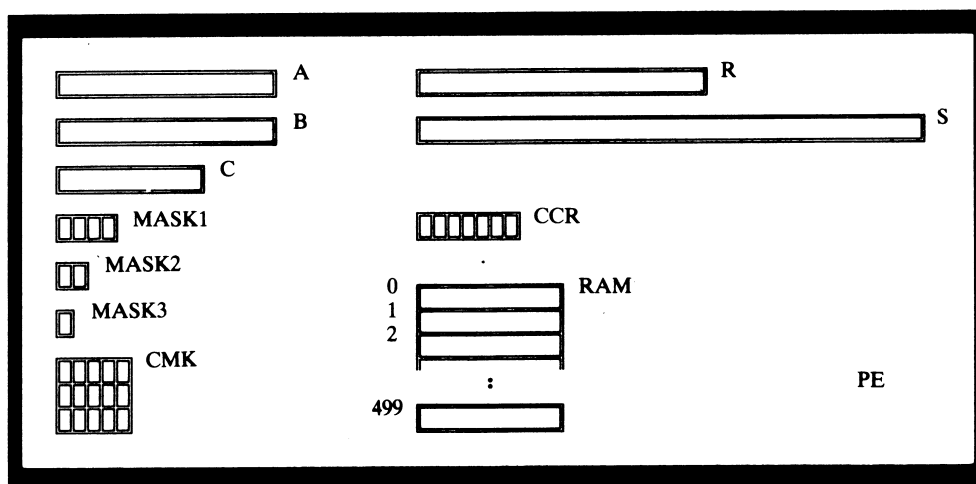
```
bit VAR[n];
```

(the former of which defines a single *n*-bit register and the latter an array of *n* single-bit registers) is that the latter allows access to a specific bit by means of an array index (e.g. *VAR[i]*), whereas the former requires the ACLAN bit operator.

Figure 2 illustrates the local memory structure of a possible conceptual PE together with the corresponding declaration of its components. Since the syntax of array dimension definitions allows for multi-dimensional arrays, the *RAM* component could equally have been defined, for example, by

```
short int RAM[10][50];
```

where *RAM[i][j]* is the physical element number ($50 \cdot i + j$). Though unrealistic as regards the physical nature of the memory component, this kind of definition is useful if the data to be distributed fall naturally into separate groups (such as the columns or rows of a matrix). Note that neither the logical identification register (LIR), whose contents identify the PE, nor the data routing register(s) DTR need to be specified. To refer to LIR, ACLAN instructions use the special identifier #, but the syntax of routing instructions is such as



```
int A, B;
short int C, RAM[500];
bit MASK1:4, MASK2:2, MASK3, CMK[3]:5, CCR[7];
float R;
double S;
```

Figure 2. PE local memory definition

to avoid any need for explicit mention of DTR registers. **ALU definition.** ALU requirements are specified by defining each of the operations as a C function. No extra syntax is involved, and a number of basic operations included as ACLAN operators do not need definition because the translator treats them as valid ALU operations when found in parallel instructions.

3.1.2 Inter-PE communication definition

An inter-PE communication definition begins by declaring the maximum number of PEs required. ACLAN programs will be able to work with any number of nodes, but always less than the number declared in this section. This is to say, we can restrict ourselves to any subarray without redefining the virtual array processor. The required inter-PE communication network is specified by defining a number of interconnection functions (IFs). An IF returns, for each PE index, the PE index of one of its neighbours (directly connected PEs). Each IF is identified by means of a name and a number. While the number characterizes the IF, the name can identify one IF or a group of related IFs. In the first case, we have nominal or directly indexed IFs, and in the second case we have indirectly indexed IFs. The actual calculation of an IF is performed by a C function referenced by the IF name. The IF identification number is used as a reference index by the ACLAN routing statements.

#16

```
IF0(node) = bit_0(node);
IF1(node) = bit_1(node);
IF2(node) = bit_2(node);
IF3(node) = bit_3(node);
```

```
long bit_0( pe )
long pe;
{ return( pe ^ 1 ); }
```

```
long bit_1( pe )
long pe;
{ return( pe ^ 2 ); }
```

```
long bit_2( pe )
long pe;
{ return( pe ^ 4 ); }
```

```
long bit_3( pe )
long pe;
{ return( pe ^ 8 ); }
```

(a)

#16

```
IFi(node) = hypercube(i, node), i in 0:3;
```

```
long hypercube( i, pe )
short i; long pe;
{ return( pe ^ (1 << i) ); }
```

(b)

Figure 3. Hypercube interconnection network definition
(a) Directly indexed definition,
(b) Indirectly indexed definition

The use of the two forms of the network definition is illustrated in Figure 3, which specifies the interconnection network of a 16-node hypercube computer.¹³ As we can realize, the indirectly indexed IF declarations specify the identification numbers by means of an indexing variable, whose range of possible values is specified in the same way

indexing_var in initial_value:final_value:increment

The indirectly indexed definition is specially suitable for declaring interconnection networks with a high degree of symmetry, such as a hypercube network, or networks that can be reconfigured by the programmer. For example, the Massively Parallel Processor (MPP),¹⁴ composed of 16384 PEs distributed in a 128*128 square, has an array communication topology similar to the Illiac IV; each PE communicates with its nearest neighbour (up, down, right and left). This network can be reconfigured, because the edges of the square can be left open or else, the opposite edges can be connected to each other. The top and bottom edges can either be left open or be connected, each PE with its opposite PE of the same column, and the right and left edges have four states of connectivity, open, cylindrical, open spiral and closed spiral. Figure 4 shows a possible definition of this network. If the network is completely left open, we must use the IFs of identification numbers 2, 3, 4 and 5, and if the top and bottom edges are connected and the right and left edges are connected as an open spiral, we must use the IFs of numbers 0, 1, 8 and 9.

#16384

```
IFi(node) = MPPup(node,i), i in 0:2:2,
IFi(node) = MPPdown(node,i), i in 1:3:2;
IFi(node) = MPPleft(node,i), i in 4:10:2;
IFi(node) = MPPright(node,i), i in 5:11:2;
```

```
long MPPup(pe,i)
long pe; short i;
{ if (i) return( (pe - 128) % 128 );
  return( (pe < 128) ? pe : (pe - 128) ); }
```

```
long MPPdown(pe,i)
long pe; short i;
{ if (i == 3) return( (pe + 128) % 128 );
  return( (pe > 16384) ? pe : (pe + 128) ); }
```

```
long MPPleft(pe,i)
long pe; short i;
{ if (i == 10) return( (pe - 1) % 16384 );
  else if (i == 8) return( (pe > 0) ? (pe - 1) : pe );
  else if (i == 6)
    return( (pe & 127) ? (pe - 1) : (pe + 127) );
  return( (pe & 127) ? (pe - 1) : pe ); }
```

```
long MPPright(pe,i)
long pe; short i;
{ if (i == 11) return( (pe + 1) % 16384 );
  else if (i == 9) return( (pe < 16383) ? (pe + 1) : pe );
  else if (i == 7)
    return( (pe & 127 == 127) ? (pe - 127) : (pe + 1) );
  return( (pe & 127 == 127) ? pe : (pe + 1) ); }
```

Figure 4. MPP interconnection network definition

3.2 Simulation routines

The sequential translator contained in ACLE uses a library of simulation routines to translate the ACLAN source program into a sequential C object program. The translator substitutes, in particular, each ACLAN parallel executable statement for a sequential C code routine that simulates the parallel statement. While the ACLAN parallel statement expresses an action that is executed by means of a certain group of PEs, the above mentioned sequential routine expresses the very same action, this time executed by the same group of PEs but one after the other. This is to say, the implicit parallelism associated with the ACLAN statement is transformed into a loop whose index moves along the logical identifications of the group of PEs that execute the action.

Now, we will see the sequential codes generated by the translator for the three kinds of parallel statements. In the examples that will follow, we will work on the basis of a hypercube array processor (see interconnection network in figure 3) but having the PE local memory structure shown in figure 2. The local parallel statement is the easiest one to be translated into a sequential statement because there is no interaction among PEs neither among PEs and CU. For example, the unidirectional local statement

$RAM[184] := A + 3*B;$

is substituted by the simulation code

```
for (_node_ = 0; _node_ < NumPrc; _node_ ++)
```

```
    RAM[_node_][184] = A[_node_] + 3*B[_node_];
```

where *NumPrc* is a predefined integer variable that is initialized to *_NumPrc_*, the number of PEs declared in the inter-PE definition section of the virtual array processor, but its content can be modified at execution time by the programmer. Note that each local memory component is redeclared with an additional dimension bound to *_NumPrc_*. The interchange local statement

$R := S;$

is translated into the statement

```
for (_node_ = 0; _node_ < NumPrc; _node_ ++)
```

```
    {tmpvarr_ = R[_node_];
```

```
    R[_node_] = S[_node_];
```

```
    S[_node_] = tmpvarr_;}
```

If we use multiple parallel operands and a mask, the sequential C code is more complex. For example, the local statement

$CCR[1:6:2] := CMK.2 \{ \# .0 \};$

is substituted by the code

```
for (_node_ = 0; _node_ < NumPrc; _node_ ++)
```

```
    if (_bitop_( _node_, 0 )){
```

```
        _k0_ = 1;
```

```
        _l0_ = 0;
```

```
        _continue_ = 1;
```

```
        while (_continue_){
```

```
            CCR[_node_][_k0_] =
```

```
                _bitop_(CMK[_node_][_l0_], 2);
```

```
            _k0_ = (_k0_ + 2 > 6)?1:(_k0_ + 2);
```

```
            if (_k0_ == 1) _continue_ = 0;
```

```
            _l0_ = (_l0_ + 1 > 2)?0:(_l0_ + 1);
```

```
            if (_l0_ == 0) _continue_ = 0;}}
```

where the call to the macro *_bitop_* replaces the bit parallel operator (*.*). Similar macros are used to simulate the operations represented by means of the bit and set ACLAN parallel operators.

The next one as regards complexity of translation is the remote parallel statement. In this case, there are message transferences among PEs. For example, the unidirectional remote action

$RAM(neigh[t]) <-- 2*RAM \{ MASK1.2,3 \};$

is transformed into the statement

```
for (_node_ = 0; _node_ < NumPrc; _node_ ++)
```

```
    if (_bitrop_(MASK1,2,3))
```

```
        {_l0_ = 0;
```

```
        _continue_ = 1;
```

```
        while (_continue_){
```

```
            _routbufl_[_node_][_l0_] =
```

```
                2*RAM[_node_][_l0_];
```

```
            _l0_ = (_l0_ + 1 > 499)?0:(_l0_ + 1);
```

```
            if (_l0_ == 0) _continue_ = 0;}}
```

```
for (_node_ = 0; _node_ < NumPrc; _node_ ++)
```

```
    if (_bitrop_(MASK1,2,3))
```

```
        {_neigh_ = _neighbour_( _node_, (short) t);
```

```
        _k0_ = _l0_ = 0;
```

```
        _continue_ = 1;
```

```
        while (_continue_){
```

```
            RAM[_neigh_][_k0_] =
```

```
                _routbufl_[_node_][_l0_];
```

```
            _k0_ = (_k0_ + 1 > 499)?0:(_k0_ + 1);
```

```
            if (_k0_ == 0) _continue_ = 0;
```

```
            _l0_ = (_l0_ + 1 > 499)?0:(_l0_ + 1);
```

```
            if (_l0_ == 0) _continue_ = 0;}}
```

where the data transference from the sending PEs to the corresponding reception PEs is carried out in two stages,

- first, a reading process, where the data is transferred from all the non-masked PEs and stored in a temporal buffer, and
- second, a writing process, where the data, read from the buffer, is placed in all the reception PEs.

For each node, the corresponding reception PE is calculated by means of predefined ACLAN function *_neighbour_()*, whose body depends on the interconnection network definition. In the case of the hypercube network (see figure 3 (b)), this function is defined as follows,

```
long _neighbour_(node, index)
```

```
long node;
```

```
short index;
```

```
{
```

```
    if (index >= 0 && index <= 3)
```

```
        return ( (long) hypercube(node, index) );
```

```
    return(-1);
```

```
}
```

A similar two stage mechanism is used for the bidirectional remote statement. In this way, the action

$A(neigh[2]) <--> A;$

is translated into the statement

```
for (_node_ = 0; _node_ < NumPrc; _node_ ++)
```

```
    {_routbufl1_[_node_] = A[_node_];
```

```

_neigh_ = _neighbour_( _node_, (short) 2);
_routbuff2_[_node_] = A[_neigh_];
for ( _node_ = 0; _node_ < NumPrc; _node_ ++ )
{ _neigh_ = _neighbour_( _node_, (short) 2);
  A[_neigh_] = _routbuff1_[_node_];
  A[_node_] = _routbuff2_[_node_];
}

```

Note that this two-stage process is reduced to a one-stage process if the left-hand side local variable of the remote operator does not appear in the right-hand side of it (nor in the mask).

The central parallel statement represents the most difficult one to be simulated because it expresses the interaction among the CU and a certain group of PEs. For example, if the CU has two hybrid variables declared as follows

```
short int data[200][10];
```

then the central action of distribution (loading)

```
RAM[100:200] <= data[*][4] {A in -3:16};
```

is substituted by the statement

```

_i0_ = 0;
for ( _node_ = 0; _node_ < NumPrc; _node_ ++ )
  if ( !_nop_(A[_node_], -3, 16) )
    for ( _i0_ = 100; _i0_ < 200; _i0_ ++ ) {
      RAM[_node_][_i0_] = data[_i0_][4];
      _i0_ = (_i0_ + 1) % 200;
    }

```

and the central statement of recollection (unloading)

```
data <= RAM {# == 6 ; ; # == 10};
```

is translated into the statement

```

_node_ = 0;
_continue_ = 1;
while ( !(_node_ == 6 ; ; _node_ == 10) &&
        _continue_ ) {
  _node_ = (_node_ + 1) % NumPrc;
  if ( _node_ == 0 ) _continue_--;
}
if ( _continue_ ) {
  _i0_ = 0;
  for ( _i0_ = 0; _i0_ < 200; _i0_ ++ )
    for ( _i1_ = 0; _i1_ < 10; _i1_ ++ ) {
      data[_i0_][_i1_] = RAM[_node_][_i0_];
      _i0_ = (_i0_ + 1) % 500;
      if ( _i0_ == 0 ) {
        _node_ = (_node_ + 1) % NumPrc;
        while ( !(_node_ == 6 ; ; _node_ == 10) )
          _node_ = (_node_ + 1) % NumPrc;
      }
    }
}

```

4. MAPPING ALGORITHMS ONTO HYPERCUBE COMPUTERS

In general, the programming of concurrent computers can be approached in two ways, the *ab initio* design of new parallel algorithms without reference to sequential algorithms for the problem to be solved; and the adaptation of existing optimal sequential solutions to concurrent architectures. In the case of problems whose sequential algorithms are very irregular or possess strong logical ordering, it is likely that the *ab initio* approach may be able to yield parallel algorithms that perform better than those resulting from direct adaptation of existing sequential solutions. There are nevertheless many other problems whose regularity allows

development costs to be saved by adapting existing sequential algorithms.

Three kinds of order may be distinguished in a sequential algorithm, the local order imposed by data dependence, i.e., by the fact that some operations use intermediate results that must be obtained before those operations can be performed; inessential sequencing imposed merely by the fact that a sequential Von Neumann machine is being used; and *semi-logical* ordering in which the nature of electronic processors means, for example, that the accumulation of a sum of many numbers must take place of an addend, but without there being any logical priority among the addends. Parallel processing can speed computation to the extent of removing inessential sequencing and minimizing the effects of semi-logical order.

A q -dimensional hypercube computer is a machine of $Q = 2^q$ processors interconnected like the vertices of a q -dimensional binary cube are interconnected by its edges.¹⁵ Thus each processor PE_r ($r = 0, 1, 2, \dots, Q - 1$) has unshared two-way communication links with the q processors $PE_{r^{(b)}}$ ($b = 0, 1, 2, \dots, q - 1$), where $r^{(b)}$ is the number whose binary representation differs from that of r only at bit b .

As in other multiprocessor systems, the hypercube computer architecture is determined by the memory organization (shared or distributed), the degree of concurrency (fine- or coarse-grained), the computing power of each individual processor, and the scheme adopted for the instruction flow (SIMD or MIMD). Reference 16 shows various examples of commercial hypercube concurrent computers.

4.1 Design procedure

In view of the above, the adaptation of a sequential algorithm for parallel processing will involve the following stages,

- (1) Analysis of the sequential algorithm to identify its nested loop dimensions, inessential loops and semi-logically ordered processes.
- (2) Partition of the hypercube dimensions into subsets associated with the inessential loops of the sequential algorithm.
- (3) Distribution of the data arrays used in the algorithm among the PEs in accordance with the chosen PE indexing and data distribution schemes.
- (4) Construction of the parallel algorithm.
- (5) Optimization of performance by optimizing the partition of stage 2.

We have employed this procedure successfully for the adaptation of numerous sequential algorithms for parallel processing on hypercube computers.¹⁵

4.2 ACLAN program example

We will present now, as an example, an ACLAN routine for multiplying matrices of unrestricted size on hypercubes with an arbitrary number of dimensions.¹³ We have chosen this example because although it is quite simple it nicely illustrates the distribution of data (using the cyclic scheme) and the principle that data distribution should be determined by processing distribution, not the other way round.

We wish to compute $C = [c_{ik}]$, the N^*L product matrix of the N^*M matrix $A = [a_{ij}]$ and the M^*L matrix $B = [b_{jk}]$. An elementary sequential algorithm for this computation is the following (in C language),

```
{for ( i = 0; i < N; i ++ )
  for ( k = 0; k < L; k ++ ){
    C[i][k] = 0;
    for ( j = 0; j < M; j ++ )
      C[i][k] = C[i][k] + A[i][j]*B[j][k];}
}
```

The loop index set of this routine is an N^*M^*L array of points, and we accordingly partition the q dimensions of our 2^q -node hypercube in three q_0 -, q_1 - and q_2 -membered subsets (with $q = q_0 + q_1 + q_2$) that allow the address r of each node to be represented by a vector (r_0, r_1, r_2) , where $r = r_2 + r_1 2^{q_2} + r_0 2^{(q_1+q_2)}$. We shall associate r_0 with the rows of A and C (the i loop), r_1 with the columns of A and the rows of B (the j loop), and r_2 with the columns of B and C (the k loop). Each node of address r will handle local submatrices LA , LB and LC of dimensions n^*m , m^*l and n^*l respectively, where $n = \lceil N/2^{q_0} \rceil$, $m = \lceil M/2^{q_1} \rceil$ and $l = \lceil L/2^{q_2} \rceil$. We have adopted the cyclic scheme of data distribution, which means that the matrices are distributed as follows,

- Matrix A: Element a_{ij} is stored in position $(\lceil i/2^{q_0} \rceil, \lceil j/2^{q_1} \rceil)$ of the local submatrix LA in all the 2^{q_2} nodes for which $r_0 = i \bmod 2^{q_0}$ and $r_1 = j \bmod 2^{q_1}$.
- Matrix B: Element b_{jk} is stored in position $(\lceil j/2^{q_1} \rceil, \lceil k/2^{q_2} \rceil)$ of the local submatrix LB in all the 2^{q_0} nodes for which $r_1 = j \bmod 2^{q_1}$ and $r_2 = k \bmod 2^{q_2}$.
- Matrix C: Element c_{ik} is stored in position $(\lceil i/2^{q_0} \rceil, \lceil k/2^{q_2} \rceil)$ of the local submatrix LC in all the 2^{q_1} nodes for which $r_0 = i \bmod 2^{q_0}$ and $r_2 = k \bmod 2^{q_2}$.

The ACLAN parallel algorithm that calculates the matrix multiplication on the hypercube is as follows,

```
void matrix_product( n, m, l, q2, q1 )
long n, m, l; short q2, q1;
{ long i, j, k;
  short t;
  for ( i = 0; i < n; i ++ )
    for ( k = 0; k < l; k ++ ){
      LC[i][k] := 0;
      for ( j = 0; j < m; j ++ )
        LC[i][k] := LC[i][k] + LA[i][j]*LB[j][k];
      for ( t = q2; t < q2 + q1; t ++ ){
        R1(neigh[t]) <-- LC[i][k];
        LC[i][k] := LC[i][k] + R1[j];}
    }
}
```

R1 is a general purpose register of the nodes. Each node performs the sequential algorithm to multiply its local submatrices, after which the resulting partial sums of $a_{ij} * b_{jk}$ products are added during a round of data transfers to obtain the final c_{ik} .

The complete program that uses the ACLAN algorithm shown above, and which includes input/output and data distribution and recollection actions, is the following,

```
main()
{
  /* Input Actions */
  input_dimension_of_hypercube( &dim );
  NumPrc = 1 << dim; /* Number of Nodes */
  input_partition( &q[0], &q[1], &q[2] );
  input_dimensions_of_matrices( &N, &M, &L );
  mdim[0] = N; mdim[1] = M; mdim[2] = L;
  /* Local Dimensions Calculation */
  for ( i = 0; i < 3; i ++ ){
    po2[i] = 1 << q[i];
    nml[i] = mdim[i]/po2[i] + ( mdim[i]%po2[i] ? 1 : 0 );
  }
  /* Cyclic distribution of matrix A */
  k0 = k1 = 0;

  for ( p = 0; p < NumPrc; p += po2[2] ){
    LA[0:nml[0]-1][0:nml[1]-1] <= =
    A[k0:nml[0]*po2[0]-1:po2[0]][k1:
    nml[1]*po2[1]-1:po2[1]]
    -1:po2[1]][#.(dim-1):q[2] <= =
    p.(dim-1):q[2]];
    k1 = (k1 + 1 == po2[1]) ? 0 : k1 + 1;
    if ( k1 == 0 ) k0 = (k0 + 1 == po2[0]) ? 0 :
    k0 + 1;
  }
  /* Cyclic distribution of matrix B */
  /* ... Similar to matrix A ... */
  /* Matrix Multiplication */
  matrix_product( nml[0], nml[1], nml[2], q[2], q[1] );
  /* Cyclic recollection of matrix C */
  k0 = k1 = 0;
  for ( p0 = 0; p0 < po2[0]; p0 ++ )
    for ( p2 = 0; p2 < po2[2]; p2 ++ ){
      C[k0:nml[0]*po2[0]-1:po2[0]][k1:nml[2]*po2[2]-1:
      po2[2]] <= =
      LC[0:nml[0]-1][0:nml[2]-1]
      {# <= = (p0 << q[1] + q[2]) + p2};
      k1 = (k1 + 1 == po2[2]) ? 0 : k1 + 1;
      if ( k1 == 0 ) k0 = (k0 + 1 == po2[0]) ? 0 :
      k0 + 1;
    }
  /* Output Actions */
  output_result_matrix( C );
}
```

Note that the current number of nodes of the target machine is expressed by means of the *NumPrc* ACLAN predefined variable.

If we want to execute the above mentioned program on a sequential computer we need to use the ACLAN-to-C translator described in section 3.2. Our matrix multiplication program is translated into the sequential C code shown in figure 5. In particular, figure 5 (a) illustrates the C code corresponding to the cyclic distribution of the matrix A, figure 5 (b) shows the C code associated with the recollection of the matrix C, and figure 5 (c) shows the code corresponding to the matrix multiplication routine.

5. CONCLUSIONS

The simulation package ACLE has been developed with standard C specifications³. Since the only external routines it requires are a library of standard I/O and dynamic memory allocation functions, it is fully portable to other machines.

```

k0 = k1 = 0;
for ( p = 0; p < NumPrc; p += po2[2] ){
{
    _i0_ = k0;
    _i1_ = k1;
    for (_node_ = 0; _node_ < NumPrc; _node_++)
    if (_bitrop_(_node_, (dim - 1), q[2]) == _bitrop_(p, (dim - 1), q[2]))
        for ( _i0_ = 0; _i0_ <= (nml[0] - 1); _i0_++)
            for ( _i1_ = 0; _i1_ <= (nml[1] - 1); _i1_++){
                LA[_node_][_i0_][_i1_] =
                    A[_i0_][_i1_];
                _i1_ = (_i1_ + (po2[1] > (nml[1]*po2[1] - 1))?(k1):(_i1_ + (po2[1])));
                if (_i1_ == (k1))
                    _i0_ = (_i0_ + (po2[0] > (nml[0]*po2[0] - 1))?(k0):(_i0_ + (po2[0])));
                k1 = (k1 + 1 == po2[1]) ? 0 : k1 + 1;
                if ( k1 == 0) k0 = (k0 + 1 == po2[0]) ? 0 : k0 + 1;
            }
        }
}

```

(a) Cyclic distribution of the matrix A

```

k0 = k1 = 0;
for ( p0 = 0; p0 < po2[0]; p0++ )
    for ( p2 = 0; p2 < po2[2]; p2++ ){
        {
            _node_ = 0;
            _continue_ = 1;
            while (!(_node_ == (p0 << q[1] + q[2]) + p2) && _continue_){
                _node_ = (_node_ + 1) % NumPrc;
                if ( _node_ == 0 ) _continue_ = 0;
            }
            if ( _continue_ ){
                _i0_ = 0;
                _i1_ = 0;
                for ( _i0_ = k0; _i0_ <= (nml[0]*po2[0] - 1); _i0_ = _i0_ + (po2[0]))
                for ( _i1_ = k1; _i1_ <= (nml[2]*po2[2] - 1); _i1_ = _i1_ + (po2[2])){
                    C[_i0_][_i1_] = LC[_node_][_i0_][_i1_];
                    _i1_ = (_i1_ + 1 > (nml[2] - 1))?(0):(_i1_ + 1);
                    if (_i1_ == (0)){
                        _i0_ = (_i0_ + 1 > (nml[0] - 1))?(0):(_i0_ + 1);
                        if (_i0_ == (0)){
                            _node_ = (_node_ + 1) % NumPrc;
                            while (!(_node_ == (p0 << q[1] + q[2]) + p2))
                                _node_ = (_node_ + 1) % NumPrc;
                        }
                        k1 = (k1 + 1 == po2[2]) ? 0 : k1 + 1;
                        if ( k1 == 0) k0 = (k0 + 1 == po2[0]) ? 0 : k0 + 1;
                    }
                }
            }
        }
    }
}

```

(b) Cyclic recollection of the matrix C

```

void matrix_product( n, m, l, q2, q1)
{
    for ( i = 0; i < n; i++ )
        for ( k = 0; k < l; k++ ){
            {for ( _node_ = 0; _node_ < NumPrc; _node_++ )
                LC[_node_][i][k] = 0;}
            for ( j = 0; j < m; j++ )
                {for ( _node_ = 0; _node_ < NumPrc; _node_++ )
                    LC[_node_][i][k] = LC[_node_][i][k] + LA[_node_][i][j]*LB[_node_][j][k]; }
            for ( t = q2; t < q2 + q1; t++ ){
                {for ( _node_ = 0; _node_ < NumPrc; _node_++ )
                    { _neigh_ = _neighbour_(_node_, (short) t);
                        R1[_neigh_] = LC[_node_][i][k]; }}
                {for ( _node_ = 0; _node_ < NumPrc; _node_++ )
                    LC[_node_][i][k] = LC[_node_][i][k] + R1[_node_]; }}
            }
        }
}

```

(c) Matrix multiplication routine

Figure 5. Sequential C code for the matrix multiplication program

The new structures introduced in ACLAN constitute a natural extension of standard C to parallel programming and are sufficiently general as to make ACLAN programs largely machine-independent. The direct programming of local memory and inter-PE data transfer make ACLAN especially useful in applications in which local memory capacity or execution time are critical. Most of the current parallel systems lack an inherently parallel programming language. Generally, these systems are programmed with a familiar sequential language (such as Fortran, C or Pascal) which is helped by means of a library of special functions that permit use of the specific parallel features of the machine. Some typical functions of this library are those which specify transferences of information between different nodes or which create or destroy processes. Obviously, the great difference between the syntactic structures of sequential languages and the parallel features of the machine make this programming method hard and error-prone. A parallel language such as ACLAN, whose syntactic structures allow a direct expression of the problem parallelism, eases the programming of parallel systems since parallel programs can be expressed in a natural way. One of the main drawbacks of this new way of programming is that the user is forced to think in *parallel*.

Recently, we have implemented ACLAN on the NCUBE/10 hypercube concurrent computer and have developed an ACLAN to C translator for it.^{12,17} This implementation allows us to program the NCUBE/10 system as an array processor. Now we are implementing ACLAN on a transputer network.

REFERENCES

1. C. D. Howe and B. Moxon, How to Program Parallel Processors. *IEEE Spectrum*, **24** (9) September, 36–41 (1987).
2. O. G. Plata, *ACLAN, A Parallel Language for Multiprocessor Systems* (in Spanish), Ph.D. Dissertation, Univ. Santiago de Compostela, Spain, February (1989).
3. B. W. Kernighan and D. M. Ritchie, *The C Programming Language*. Prentice Hall, Englewood Cliffs, N.J. (1978).
4. R. H. Perrott, A Language for Array and Vector Processors. *ACM Transactions on Programming Languages and Systems*, **1** (2), October, 177–195 (1979).
5. R. H. Perrott, R. W. Lyttle and P. S. Dhillon, The Design and Implementation of a Pascal-based Language for Array Processor Architecture. *Journal on Parallel and Distributed Computing*, **4** (3), June, 266–287 (1987).
6. D. Crookes, P. J. Morrow, P. Milligan, P. L. Kilpatrick and N. S. Scott, An Array Processing Language for Transputers Network. *Parallel Computing* **8** (1–3), August, 141–148 (1988).
7. K.-C. Li and H. Schwetman, Vector C: A Vector Processing Language. *Journal of Parallel and Distributed Computing* **2** (2), May, 132–169 (1985).
8. G. Paul, VECTRAN and the Proposed Vector/Array Extensions to ANSI FORTRAN for Scientific and Engineering Computation, in *Proc. IBM Conf. Parallel Computers and Scientific Computations* Rome, Italy, March, 143–162 (1982).
9. A. P. Reeves, Parallel Pascal for Parallel Computers, *Journal of Parallel and Distributed Computing* **1** (1), August, 64–80 (1984).
10. American National Standards Institute, *Fortran 8X Version 98 X3J3*. New York (1986).
11. O. G. Plata, F. F. Rivera, A. L. Zapata and I. Benavides, A Parallel Programming Language for Signal Processing, in *1989 Int. Mediterranean Electrotechnical Conference*, Lisbon, Portugal, April, 261–264 (1989).
12. O. G. Plata, F. Argüello, J. D. Bruguera and E. L. Zapata, An Array Processing Language for Real Time Programming of Hypercube Concurrent Computers. *IEEE 1989 Sec. Int. Conf. on Software Engineering for Real Time Systems*, Cirencester, UK, September, 141–155 (1989).
13. C. L. Seitz, The Cosmic Cube. *Communications of the ACM*, **28** (1), January, 22–33 (1985).
14. K. E. Batcher, Design of a Massively Parallel Computer. *IEEE Transactions on Computers*, **C-29** (9), September, 836–841 (1980).
15. E. L. Zapata, F. F. Rivera and O. G. Plata, On the Partition of Algorithms into Hypercubes. *Advances on Parallel Computing*, D. J. Evans, Ed., JAI Press, UK (1990) (to appear).
16. T. Johnson and T. Durham, *Parallel Processing: the Challenge of New Computer Architectures*. Ovum. Ltd. Press, London (1987).
17. O. G. Plata, E. L. Zapata, F. F. Rivera and R. Peskin, An Array Processing Language for Message-Passing Hypercubes. *Advances on Parallel Computing*, D. J. Evans, Ed., North-Holland, Amsterdam (1990) (to appear).