INTRODUCTION TO VECTOR PROCESSING

H. Lipps

CERN, Geneva, Switzerland

Abstract

Vector processing is seen as an entry into the wider field of parallel computing. After reviewing the need for it, we introduce data dependence, the major new concept essential to parallel computing, in chapter 2. The basic problems related to the dependence graph of a program are discussed with emphasis on Fortran. Vector instructions available to the Fortran programmer are presented in chapter 3, and typical machine instructions in chapter 4. Chapter 5 shows how the typical elements of vector machines fit into the overall design of the von Neumann computer, and what are the key elements of machine performance. The final chapter deals with loop vectorization in Fortran programs, the concept of recurrence, and some programming measures to alleviate its consequences.

1. MOTIVES FOR VECTOR PROCESSING

Vector processing is a complication to computing, invented to make number crunchers go faster. Is there a need for computers to become faster and faster? And if so, must fast computers also be more complicated?

Science and technology have many problems for which progress depends on more powerful computers. The electronic computer was invented to cover such needs, and since then more powerful machines have been built all the time in order to perform calculations that had been impossible before. Computing a weather forecast for five days is not worthwhile as long as the calculation takes a week to carry out.¹ Elaborate models of quantum chromodynamics are still beyond the reach of to-day’s computers.

There are two basic methods to make computers go faster: To use faster circuits, or to use more of them in parallel. Every form of parallel computing, of which vector processing is just one, poses formidable software problems, and software might anyway be considered a major bottleneck in the development of complicated applications. Progress in parallel computing does not, however, depend in any fundamental way on the further progress in circuit speeds. A program that vectorizes on our present machines may be expected to execute faster again on the next generation of vector computers.

¹ And yet the computation of a one-day weatherforecast was already tried on the ENIAC in 1950! [3]
1.1 On circuit development

The introduction of electronic circuits forty years ago reduced the time for addition from the second to the millisecond — thus providing a thousand times the speed of mechanical and relay computers. To-day the arithmetical operations can be performed within nanoseconds. The following table illustrates how circuits have developed over the years. As a measure of circuit speed we give the time of multiplication for a selection of "supercomputers" of their times; as a measure of circuit density (and cost) we give the main memory capacity of these machines.

<table>
<thead>
<tr>
<th>Year</th>
<th>System</th>
<th>Time (nsec)</th>
<th>Memory Capacity</th>
</tr>
</thead>
<tbody>
<tr>
<td>1951</td>
<td>UNIVAC I</td>
<td>2,000,000</td>
<td>1 K words</td>
</tr>
<tr>
<td>1964</td>
<td>CDC 6600</td>
<td>1,000</td>
<td>128</td>
</tr>
<tr>
<td>1968</td>
<td>CDC 7600</td>
<td>140</td>
<td>512</td>
</tr>
<tr>
<td>1976</td>
<td>Cray 1</td>
<td>90</td>
<td>&lt;4 000</td>
</tr>
<tr>
<td>1985</td>
<td>Cray 2</td>
<td>20</td>
<td>256 000</td>
</tr>
</tbody>
</table>

The development of multiplication times makes evident that progress towards faster circuits has slowed down considerably. In contrast, the growth of memory capacities demonstrates that density of circuits still increases at a very high rate. What will be the future? Circuit speed is limited by the speed of light — 30 centimeters per nanosecond in vacuum — and by the speed of switching. Switching times have recently been reduced below 0.5 picoseconds in the laboratory. Faster switching speeds in commercially available computers may be expected from the use of new types of semiconducting materials, in particular of gallium arsenide. The limit on transmission speed means that faster computers must become smaller. Advances are made through large scale integration of circuits: 4-megabit storage chips are already available, and still denser chips will come. But then, the interconnections on a 256-megabit chip would be only a few hundred atoms wide! So again, multi-layer chips are being investigated.

Yet electronic computing — i.e. electric current — consumes energy, which transforms into heat. Cooling becomes more difficult as circuits are being packed more densely. Speeding up circuitry thus reaches another physical limit. Further speedup through use of still smaller elements would come, however, if supraconducting currents could be used economically, since these would reduce the cooling problem. The hopes based on supraconducting currents have been heightened recently, since supraconducting materials have been found for temperatures above seventy degrees Kelvin, where cooling through liquid nitrogen becomes possible.

While further improvement to the speed of electronic circuits may thus be expected, the rate of progress is likely to be modest when compared to the early times of electronic computing. Otherwise, parallel operation in one form or another will be the only way to build more powerful computers.

1.2 Vector processing, one form of parallelism

As science and technology depend heavily on mathematics, the usefulness of computers in these fields depends largely on their usefulness in numerical analysis. In turn, numerical analysis is strongest in the area of linear analysis, since quite generally, the linearization of a nonlinear problem is usually the first step to its solution — and sometimes even the only approximation that can be computed at all. It is therefore natural that algorithms of linear algebra are important for the computational treatment of a wide range of scientific and technical problems. As a consequence, the algorithms of linear
algebra take a large share of the computer time that is used in technical calculations. The mathematicians also call their linear spaces vector spaces, since a linear space is made up of vectors. Computers that perform vector operations efficiently have therefore gained the market of large numerical problems from science and technology. Vector computers currently dominate in applications which require the solution of large sets of linear equations, as they arise e.g. from partial differential equations: Structural analysis, weather forecasting, etc.

However, the need for number crunchers has also arisen with scientific and technical applications that have only peripheral connections, if any, to linear algebra — like experimental particle physics or the generation of motion pictures. It remains a key question for the future of vector processors to what extent they can be applied to wider classes of problems, not dominated by linear algebra.

2. THE LOGIC OF PARALLEL PROCESSING

Most of the elementary vector operations consist of a series of independent calculations for all elements of the operand vectors, and so may be performed in parallel. Vector processing may thus be seen as one particular form of parallel computing. Other forms of parallel processing exist, with individual operations at various levels of complexity: At one extreme, multi-tasking allows several processors to work concurrently on different, arbitrarily complex sections of a program. At the other extreme, even the simplest CPU is wired so as to perform several of its most basic logic functions during the same clock period. Since some forms of parallel computing will unavoidably dominate the future of high-performance computers, we shall now examine the common basic constraints that logic imposes on all forms of parallel computing.

2.1 Fundamental concepts of parallel processing

When we move from sequential to parallel processing we put into question some basic conventions that we practice subconsciously whenever we program. There are two ways to look at a program, one for the user and one for the programmer. The user considers his program as a tool to obtain some result (record) from some data (record). He may perceive an execution of his program as a transformation of memory contents in a single, atomic step.

In this context, a matter of definition must be clarified. Consider the following program:

\[
T = A_1 \\
A_1 = B_1 \\
B_1 = T \\
T = A_2 \\
A_2 = B_2 \\
B_2 = T
\]

A user of this program may regard it as swapping vectors \((A_1,A_2)\) and \((B_1,B_2)\). He may thus be interested only in the contents of these four memory locations, and not in the fact that the run leaves the old value of \(A_2\) in the temporary location \(T\). He would therefore accept
T = A2
A2 = B2
B2 = T
T = A1
A1 = B1
B1 = T

as an equivalent program, whereas a user who considers T as part of his memory would not. In short: whether, as a user, we shall be prepared to accept some reordering of statements in our program may depend on the use we intend to make of it.

The programmer sees the execution of his program at a finer level of granularity. He sees each instruction of his program as an atomic item. Every executable instruction (except possibly a control instruction) references some data elements (its operands) and defines or redefines some corresponding result in memory. The instruction is to the programmer what the entire program is to the user: a tool to transform memory contents.

The programmer perceives his instructions in two ways, as he distinguishes between the program per se and a run of it. The program itself consists of a sequence of instructions “in space”, where each instruction is identified, say, by the line number which the editor assigns to it. This ordering of instructions in space is essential to the meaning of the program. Did you ever drop a deck of cards?

The programmer, however, also regards his program as a prescription to carry out an algorithm. A sequence of instructions “in time” (sometimes called an instruction trace) develops during each run of the program. Thus the program is understood as a sequence of transformations of memory that unfolds in time. Each instruction is seen as having completed its transformation of memory before the next one performs its task, and so each instruction operates on the memory contents left behind by its forerunners. The memory contents at the end of the run represents the result of the entire calculation, as seen by the user.

The instruction trace of a run depends not only on the program, but also on the data for this particular run. The instruction trace for a particular data record may not, in general, be predicted without actually performing the run. Two things occur in parallel. While each instruction that is called up in turn for execution performs its transformation of memory contents, the instruction to be executed next is located. We may imagine a program counter P that identifies the line number of the instruction being executed. Initially P is set to the starting line of the program. When an assignment statement is executed, P advances by one; control statements permit P to depart from this linear sequence.

Now, to open the way to parallel computing, we should like to retain this interpretation of our program, and thus retain its meaning, but obtain a degree of freedom in its execution: Can we change the order of instructions in time without risking a different result? How can we analyze the problem systematically? The answer will be contained in the dependence graph of the execution.²

² The discussion of dependence becomes confused unless a clear distinction is made between instructions “in space” and instructions “in time”. Here we deal only with dependence of instructions in time, which is basic. Vectorization of loops will bring us to dependence of instructions in space. (See chapter 6)
Although we are mainly interested in the parallel execution of Fortran statements, it should be clear that the same logical problem and the same method of analysis prevail also at other levels of detail. Sequential code at a higher level of granularity poses the problem of parallel execution on the next lower level. In particular, the Fortran programmer is not concerned whether the machine instructions that result from his program will be executed in strict sequence. We shall see later that some degree of parallel execution is implemented in all "sequential" machines.

2.2 Example: Floating point multiplication

We underline the generality of the problem and its solution with an example taken from the hardware level. If we assume that floating point numbers are represented with the help of binary exponents, say

\[ A = a \cdot 2^\alpha, \quad B = b \cdot 2^\beta, \quad C = c \cdot 2^\gamma, \]

then the product

\[ A = B \cdot C \]

might be formed in eight steps, as follows:

s: \quad B = \text{normalize}(B)  \\
t: \quad C = \text{normalize}(C)  \\
u: \quad (b, \beta) = \text{unpack}(B)  \\
v: \quad (c, \gamma) = \text{unpack}(C)  \\
w: \quad a = b \cdot c  \\
x: \quad \alpha = \beta + \gamma  \\
y: \quad A = \text{pack}(a, \alpha)  \\
z: \quad A = \text{normalize}(A)  

Operation t does not depend on the result of operation s, and may thus be performed concurrently. Operation u must await the outcome of operation s only, and v must wait only for the termination of t. Thereafter, w and x require the results from both u and v, but x need not wait for w. Step y needs the result from both w and x. Finally z can only be performed when y has been completed. All these conditions are expressed by the dependence graph in Figure 1.

![Dependence graph for floating point product](image)

*Figure 1: Dependence graph for floating point product*
2.3 Aspects specific to Fortran

When we consider the problem of dependence for a series of executable Fortran statements, we shall always assume that no input/output is involved. Note that, by definition, a Fortran variable is always a "simple" variable, not an array element. Following the Fortran 8x proposal [2] we call a variable or array element a scalar. A scalar is said to become defined or redefined when it is assigned a value. Any use of a scalar as operand or subscript is called a reference to the scalar.

The Fortran language presents a number of traps for the construction of dependence graphs. This leads to ambiguous situations, which compilers cannot possibly resolve on their own. As a result, a number of seemingly innocent program constructs will fail to vectorize. Some such obstacles to vectorization may be removed with the help of compiler directives — i.e. Fortran comment lines of a particular, compiler dependent format whose contents are recognized and taken into account during compilation.

For one, given a subprogram, it is impossible to say whether two of its dummy arguments correspond to the same or different actual arguments if the dummy arguments are of the same type. For example,

```
CALL SUB(A, A)
```

would associate the same variable A with both dummy arguments of

```
SUBROUTINE SUB(X, Y)
```

Any call to a subroutine may of course affect the contents of all common locations defined in the calling program, and also all the actual arguments of the call. This will normally prevent the vectorization of a series of statements which includes a CALL statement.

Next, the use of common blocks permits external functions to have side-effects. If F is an external function, referenced by the program

```
COMMON B(2)
...
A = B(1)
C = F(D) + B(2)
E = B(1)
```

then the function subprogram F might possibly include the common array B. The reference to F might then alter the value of array element B(1) — without this change being evident from the present code. Note, however, that it would be illegal for function F to redefine B(2), given that B(2) and F are referenced within the same expression. Also, argument D of function F might be changed during the evaluation of F.

The expression (B + C + D) permits several orders of evaluation that are arithmetically equivalent. IBM Fortran [6] (p. 38) insists on evaluation from left to right; Cray (and VAX) Fortran do not.
The execution of a Fortran assignment statement defines the scalar on the left of the equal sign and (in general) references all its other scalars, including those that appear in subscript expressions. However, references to scalars also occur during the execution of control statements. Moreover, each execution of a DO statement also defines or redefines the control variable of the loop.

2.4 The three types of data dependence

Data dependence exists between a statement s that assigns a value to the scalar A and any statement t that defines or references A. Thus there are three cases.

- **True Dependence (or Flow Dependence):** Statement s that defines A is followed by a statement t that references A.

  
  s: \[ A = ... \]
  
  \[ \ldots \]
  
  t: \[ \ldots = \ldots A \ldots \]

  The execution of statement t must await the completion of statement s, so that the value assigned to A by s is available as operand to t.

- **Antidependence:** Statement s that defines A is preceded by a statement t that references A.

  
  t: \[ \ldots = \ldots A \ldots \]
  
  \[ \ldots \]
  
  s: \[ A = \ldots \]

  The execution of statement s must await the completion of statement t, so that the operand value A to be used by t is not destroyed by the execution of s.

- **Output dependence:** Statement s that defines A is followed by a statement t that also defines A.

  
  s: \[ A = \ldots \]
  
  \[ \ldots \]
  
  t: \[ A = \ldots \]

  Statement s must be executed before statement t, so that the result assigned to A by t is retained in memory.

The dependences generated by the above three cases will normally link other pairs of statements by implication, as may be seen from the following example:

  
  s: \[ \ldots = \ldots A \ldots \]
  
  \[ \ldots \]
  
  t: \[ A = \ldots B \ldots \]
  
  \[ \ldots \]
  
  u: \[ B = \ldots \]
The antidependence \( s \to t \) (caused by \( A \)) and the antidependence \( t \to u \) (caused by \( B \)) together imply that statement \( u \) can be executed only after statement \( s \).

The replacement of an operand:

\[
A = \ldots A \ldots
\]

is an action that can be reduced to the previous cases through introduction of a dummy variable, not used otherwise in the program:

\[
\begin{align*}
\text{DMY} &= \ldots A \ldots \\
A &= \text{DMY}
\end{align*}
\]

Read-only scalars of a program — i.e. scalars that are never defined during the execution of the program (segment) — can never generate any data dependence. The two statements

\[
\begin{align*}
A &= B + 1 \\
C &= B - 1
\end{align*}
\]

may be executed in any order although they share the operand \( B \).

A complication arises with the determination of dependences that are caused by the definition of array elements: A true dependence exists between the two statements

\[
\begin{align*}
A(I) &= \ldots \\
\ldots &= \ldots A(K) \ldots
\end{align*}
\]

if and only if \( K \) equals \( I \); yet it might be difficult, or even impossible to decide the validity of this equality without actually executing the program. It may hold in certain runs of the program, and not in others.

Note that the dependence graph of a run may sometimes be changed by apparently trivial changes to a program. In the sequence

\[
\begin{align*}
s: & \quad A = B + C \\
t: & \quad D = A + 5 \\
u: & \quad A = E - F
\end{align*}
\]

variable \( A \) causes a true dependence \( s \to t \), an antidependence \( t \to u \) and an output dependence \( s \to u \). The equivalent program

\[
\begin{align*}
D &= (B + C) + 5 \\
A &= E - F
\end{align*}
\]

has no dependence at all.
A list of all dependences between the statements in the instruction trace of a run is equivalent to the dependence graph of the run. It requires the instruction trace itself, a list of all scalars that are defined by the run, and where each scalar is defined and referenced. The dependence graph of a run provides full information on the possible orderings of statements that will not affect the results of the run — i.e. will leave the same values in all memory locations at the end of the run. Any or all statements to which no dependence exists may be executed at once, concurrently or in any order, and subsequently deleted from the dependence graph. Thereafter, the dependence graph for the remaining statements may be used in the same way to determine the next lot of statements to be executed.

The example given in section 1 of this chapter involves the following dependences:

\[ \begin{align*}
  s_1 : & \quad T = A1 \\
  t_1 : & \quad A1 = B1 \\
  u_1 : & \quad B1 = T \\
  s_2 : & \quad T = A2 \\
  t_2 : & \quad A2 = B2 \\
  u_2 : & \quad B2 = T
\end{align*} \]

\[ \begin{align*}
  A1 : & \quad s_1 \rightarrow t_1 \\
  B1 : & \quad t_1 \rightarrow u_1 \\
  A2 : & \quad s_2 \rightarrow t_2 \\
  B2 : & \quad t_2 \rightarrow u_2 \\
  T : & \quad s_1 \rightarrow u_1 \rightarrow s_2 \rightarrow u_2
\end{align*} \]

Thus we obtain the dependence graph

\[ s_1 \rightarrow t_1 \rightarrow u_1 \rightarrow s_2 \rightarrow t_2 \rightarrow u_2 \]

— i.e. the statements must be executed precisely in the order in which they are written.³

3. VECTOR INSTRUCTIONS IN FORTRAN

Although it is clear what vector operations are to linear algebra, we must see what they are to computing. While one wants to extend vector processing beyond the realms of linear algebra, it is not clear at present how this can best be done. Cray, Control Data, and others have put vector instructions into their large CPUs for over a decade now, but every Fortran compiler still has trouble putting the new instructions to good use. In fact, some disagreement persists as to what is a suitable set of vector instructions for a computer.

How can we discuss the vectorization of Fortran programs as long as we don’t know what vector instructions are? One answer, to be given in this chapter, comes from the compiler people. To reduce their plight, between our Fortran programs and their vector machines, they have invented the biological compiler — us! We are given a set of “Fortran vector instructions”, as additions to our Fortran 77 machine, and we are asked to employ them whenever possible in place of the standard Fortran constructs. The Fortran vector instructions have been designed by the compiler writers so as to match up more easily than standard Fortran with the vector instructions of their machines. But there is disagreement even at the Fortran level, and even within the Fortran 8x standards committee. What follows

³ The conclusion would evidently be different if the scalar \( T \) were expanded into a vector \((T_1,T_2)\), with \( T_i \) taking the place of \( T \) in statements \( s_i \) and \( u_i \).
below is mainly from the Fortran 8x proposal — that encompasses not only vector machines, but also array processors — and occasionally from Fortran dialects created by some manufacturers. The selection is for illustration only; it has a strong bias from personal preference.

3.1 Array sections

According to the Fortran 8x proposal we call the number of dimensions of an array the rank of the array, and the number of elements along any dimension the extent or size of this dimension. The shape of an array A is the rank 1 integer array whose elements are the extents of the dimensions of A. For example, the array defined by

\[
\text{DIMENSION } A(0:4,3)
\]

has rank 2 and shape (5,3).

Consider an arithmetic progression (of integers):

\[
K = M, M+I, M+2I, \ldots, M+(L-1)I
\]

Computer jargon for its increment (I) is stride. Variable K is called induction variable. When the arithmetic progression is generated by a DO loop, say

\[
\text{DO } K = M, N, I \\
\ldots \\
\text{ENDDO}
\]

the length \( L = \max\{ (N - M + 1) / I, 0 \} \) is called the trip count, and K is called the control variable of the loop.

The Fortran 8x proposal calls an arithmetic progression of subscript values a subscript triplet. As such, the arithmetic progression above is denoted M:N:I. If M is omitted, its value is assumed to be the lower bound of the dimension. Similarly, omission of N implies the upper bound of the dimension. Omission of I implies a stride of 1.

If A is an array of rank \( n \), an expression \( A(s_1,s_2,\ldots,s_n) \) denotes an array section provided at least one of the subscripts \( s_i \) denotes a subscript triplet. The concepts of extent, rank, and shape carry over naturally from arrays to array sections. Conformable array sections have the same shape. Array sections of rank 1 are called vectors. \( A(9,2:4,1:3:2) \) means the rank 2 array section

\[
\begin{align*}
A(9,2,1) & \quad A(9,2,3) \\
A(9,3,1) & \quad A(9,3,3) \\
A(9,4,1) & \quad A(9,4,3)
\end{align*}
\]

* We take advantage of the new, simplified syntax for DO loops proposed for Fortran 8x.
The X3J3 Committee could not bring itself to agree also on vector valued subscripts. These are vectors of type integer. Given the specifications

\[
\text{DIMENSION A(4,3), I(5), J(2)}
\]
\[
\text{DATA I / 2,3,4,4,2 /, J / 3,1 /}
\]

The array section \( A(\cdot,:) \) is

\[
A(2,3) \quad A(2,1) \\
A(3,3) \quad A(3,1) \\
A(4,3) \quad A(4,1) \\
A(4,3) \quad A(4,1) \\
A(2,3) \quad A(2,1)
\]

The example shows that, using vector valued subscripts, an array section size may exceed the size of the corresponding array dimension.

3.2 Array assignment statements

Using array sections and scalars, array assignment statements may now be made:

\[
A(1:N) = \sin( B(1:N,I) ) + C*B(0:N-1,J)
\]

means

\[
\begin{align*}
\text{DO } K &= 1, N \\
A(K) &= \sin( B(K,I) ) + C*B(K-1,J) \\
\text{ENDDO}
\end{align*}
\]

All array sections of an array expression must be conformable. Addition, subtraction, etc of terms applies to corresponding elements; scalar operands apply to all elements. Intrinsic\(^5\) functions permit array sections as arguments; the function value is a conformable array. Array assignment must be unique; i.e.

\[
\text{DATA I / 1, 2, 1 /}
\]
\[
A( I(1:3) ) = \ldots
\]

is not allowed since array element \( A(1) \) would be defined twice by the same array assignment statement.

By definition, array expressions are evaluated before any assignment is made. Thus

- Output dependence may not arise since each array element is assigned a value only once.
- Antidependence is resolved by definition.
- True dependence is avoided by definition.

\(^5\) For simplicity of presentation, we forgo the 8x proposal for array-valued external functions.
The significance of these rules may be seen from the following simple example:

\[
\text{DIMENSION } A(2), I(2) \\
\text{DATA I / 2,1 /} \\
A( I(: ) ) = A(:)
\]

The assignment exchanges the two elements of array A. Thus there exist vector assignment statements that cannot be expressed by a one-statement DO-loop. Another such case is

\[
B(1:5) = B(5:1 - 1)
\]

On the other hand, there exist also one-statement DO-loops that cannot be expressed by vector assignment statements:

\[
\text{DO } K = 1, N \\
S = S + A(K) \\
\text{ENDDO}
\]

is a case in point. All such cases of discrepancy between vector assignment statements and DO-loops arise from the two conventions: On one hand each executable Fortran statement must complete execution before the next one begins — thus possibly introducing data dependences between successive (scalar) assignment statements. On the other hand each array expression is evaluated before the assignment is made — with the consequences on data dependence just stated.

3.3 Special vector functions

Can we extend vector processing (and array processing) to algorithms with innate dependences? The Fortran 8x proposal covers a number of cases that arise frequently, even in linear algebra, by a series of new intrinsic functions. We present only a few of them for illustration.

3.3.1 MAXVAL( ARRAY [,DIM] [,MASK] ) and MINVAL

The arguments DIM and MASK of function MAXVAL are optional. If absent, the value of MAXVAL is that of the largest element of ARRAY. If MASK is present, it must be a logical array that is conformable with ARRAY. Elements of ARRAY for which the corresponding elements of MASK are false will then be ignored during the computation of the maximum value. If ARRAY is of rank n, argument DIM may be an integer in the range [1,n]. In this case MAXVAL is the array of rank n−1 whose elements are the maxima of those elements of ARRAY along dimension DIM that correspond to true elements of MASK. Given

\[
\text{DIMENSION } A(3,3) \\
\text{DATA } A / 1,2,3,4,5,6,7,8,9 /
\]

we have

\[
\text{MAXVAL}(A) = 9 \\
\text{MAXVAL}(A,1) = [3,6,9] \\
\text{MAXVAL}(A,2,A.L.E.5) = [4,5,3]
\]

The function MINVAL is defined similarly to MAXVAL, to provide minimal values.
3.3.2 MAXLOC(ARRAY [,MASK]) and MINLOC

The argument MASK of function MAXLOC is optional. If present, MASK must be a logical array that is conformable with ARRAY. If ARRAY is of rank n, the value of MAXLOC is a vector of n integers \(i_1,i_2,...\) such that \(ARRAY(i_1,i_2,...) = \text{MAXVAL}(ARRAY,\text{MASK})\). Thus, for the example above,

\[
\text{MAXLOC}(A) = [3,3] \\
\text{MAXLOC}(A,A.LT.5) = [1,2]
\]

The function MINLOC is defined similarly to MAXLOC, to provide the locations of minimal values.

3.3.3 ANY(MASK [,DIM]) and ALL

The function ANY has the value true if any element of array MASK [along dimension DIM] is true. The function ALL has the value true if all elements of array MASK [along dimension DIM] are true.

3.3.4 SUM(ARRAY [,DIM] [,MASK]) and PRODUCT

The function SUM accumulates the elements of ARRAY. As in MAXVAL, the arguments DIM and MASK are optional, and their significance is analogous. We thus find

\[
\text{SUM}(A) = 45 \\
\text{SUM}(A,2) = [12,15,18] \\
\text{SUM}(A,1,A.LT.5) = [6,4,0]
\]

The function PRODUCT is defined similarly to SUM, to multiply rather than add the elements of ARRAY.

3.3.5 DOTPRODUCT(X,Y)

The arguments X and Y of this function are vectors. The function has the value \(\text{SUM}(X*Y)\) provided the arguments are integer or real, or \(\text{SUM}(\text{CONJG}(X)*Y)\) if X is complex. Both vectors may also be logical, in which case the function assumes the value ANY(X.AND.Y). For example,

\[
\text{DOTPRODUCT}(A(:,1),A(:,2)) = 4 + 10 + 18 = 32
\]

3.3.6 COUNT(MASK [,DIM])

Function COUNT gives the number of elements in MASK [along dimension DIM] that have value true.

3.3.7 PACK(ARRAY, MASK, [,VECTOR])

The function is a vector whose type agrees with that of ARRAY. If present, VECTOR must be a vector also of this same type, and of a size n that is no less than the number m of true elements in
MASK. The vector PACK then consists of those m elements of ARRAY that correspond to true elements in MASK, followed by the n − m last elements of VECTOR. If VECTOR is not given, the function consists only of the m elements of ARRAY designated by MASK.

3.3.8 UNPACK(VECTOR, MASK, FIELD)

The function is an array whose type agrees with that of VECTOR and whose shape agrees with that of MASK. VECTOR must be a vector whose size n is no less than the number m of true elements in MASK. FIELD must be of the same type as VECTOR, and must be either an array that is conformable with MASK, or a scalar. The array UNPACK holds the first m elements of VECTOR in the positions designated by MASK; the remaining positions of the array are filled with the corresponding positions of FIELD.

Given
DIMENSION A(5), M(5), V(4)
DATA A / 1,2,3,4,5 /
DATA M / .TRUE.,.FALSE.,.FALSE.,.TRUE.,.FALSE. /
DATA V / 6,7,8,9 /
we have
COUNT(M) = 2
PACK(A,M) = [1,4]
PACK(A,M,V) = [1,4,8,9]
UNPACK(V,M,A) = [6,2,3,7,5]

3.4 Assignment under mask

Array assignments may be made under the conditions of a mask. Consider for instance the following DO-loop:

DO K = 1, N
   IF(A(K) .GE. 0) THEN
      B(K) = SQRT( A(K) )
   ELSE
      B(K) = 0
   ENDIF
ENDDO

Cray Fortran vectorizes this construct by means of the intrinsic function CVMGT ('conditional vector merge on true'). The above loop is rewritten as

DO K = 1, N
   B(K) = CVMGT( SQRT(A(K)), 0, A(K).GE.0 )
ENDDO

i.e. the scalar function CVMGT assumes the value of its first or second argument, depending on whether its third argument is true or false. Similarly, the Fortran 8x proposal introduces the array-valued function MERGE, which solves our problem by

B(1:N) = MERGE( SQRT(A(1:N)), 0, A(1:N).GE.0 )
If several array assignments must be made under the same mask then the WHERE-construct of the Fortran 8x proposal may be used. In our case this yields

```fortran
WHERE( A(1:N) .GE. 0 )
B(1:N) = SQRT( A(1:N) )
ELSEWHERE
B(1:N) = 0
ENDWHERE
```

In general, only assignment statements may occur inside the WHERE-blocks. In our particular case, though longer, this solution is probably more readable than that obtained through use of the MERGE function.

4. MACHINE INSTRUCTIONS

Given any standard Fortran program, we may compile and execute it on our vector computer. We might never even consider what performance we obtain from it; or we might be happy to learn from experiment that we run twice as fast as on a comparable scalar computer. However, to develop efficient code, we would like to know how far the actual performance of a program differs from the optimal use of our machine.

All manufacturers provide elaborate tools to examine program performance, but there are several schools of thought. A somewhat extreme position is probably held by a manufacturer who insists you must not ask about the instruction repertoire of their machines. My personal attitude is very much the opposite. Designing efficient Fortran code for a vector computer, the problem seems to be twofold. We must first conceive our algorithm in terms of the vector operations which our machine can perform, and then must write our Fortran program in such a manner that the compiler will produce the machine code that we had in mind in the first place. (This can surely be a frustrating exercise!) We cannot judge actual system performance unless we know what our machine can do. To deprive us from this knowledge is telling us to shoot in the dark.

To illustrate the facilities typically available on actual vector computers, this chapter presents a variety of vector instructions, and a few scalar instructions for comparison. To avoid irrelevant detail, we present them with a Fortran-like syntax; i.e. we imagine a very primitive Fortran, but augmented by vector instructions that correspond to instructions on real machines. Our picture of a register machine fits e.g. the Cray computers, the Fujitsu VP 200 line, and the IBM vector facility, but not the Control Data Star 100 computer and its derivatives, the Cyber 205 and the ETA computers. The Control Data machines take the operands of their vector instructions directly from main memory, and place the results back there. This motivates some differences in the instruction repertoire; in particular, it provides for vectors of arbitrary length. But then vector start-up times tend to be lengthy.

4.1 Scalar machine instructions

We envisage a Fortran-like language with variables and constants of three types only: INTEGER, REAL, and 64-bits. Arrays are permitted, but may have only one dimension. In addition, there shall
be eight scalar registers S₁,...,S₈ that can hold data of any one type. Memory access — that is communication between variables and arrays on one hand, and the S-registers on the other hand — is accomplished by the following four instructions:

\[
\begin{align*}
S_n &= u, & S_n &= z(m) & \text{"load"} \\
u &= S_n, & z(m) &= S_n & \text{"store"}
\end{align*}
\]

where \( m \) denotes an S-register which holds an integer value, \( u \) is any simple variable, and \( z \) is any one-dimensional array. The load instructions copy the contents of a scalar to a register, the store instructions provide for transfer in the opposite direction.

The following forms of assignment statement are permitted:

\[
S_n = S_j, \quad S_n = \text{.NOT.} S_j
\]

copy the contents of \( S_j \), or its logical negation (bit 0 for bit 1 and vice versa), into register \( S_n \). The assignments\(^7\)

\[
S_n = S_j + S_k, \quad S_n = S_j - S_k, \quad S_n = S_j \times S_k, \quad S_n = S_j / S_k
\]

perform integer or floating point arithmetic,

\[
S_n = S_j \text{.AND.} S_k, \quad S_n = S_j \text{.OR.} S_k, \quad S_n = S_j \text{.EQV.} S_k
\]

perform logical operations on bit strings, and

\[
S_n = \text{FLOAT}(m), \quad m = \text{INT}(S_j)
\]

convert integers to floating point and vice versa.

The only control statements are the unconditional jump

\[
\text{GOTO } p
\]

and the conditional jumps

\[
\text{IF( } S_j \text{ .rel. } S_k \text{ ) GOTO } p
\]

where the relational operator .rel. must be .EQ., .NE., .GE., .GT., .LE., or .LT., and where \( p \) denotes a statement label in the program, called the jump address.

---

\( ^6 \) Real machines tend to use several sets of registers to hold different types of data. We need not consider this complication here.

\( ^7 \) To be logically consistent, we should use different arithmetic operators to distinguish between integer and real arithmetic.

198
The Fortran program

\[
\text{DO } K = 1, N \\
\quad A(K) = B(K) + C(K) \\
\text{ENDDO}
\]

might then be translated into the language of our scalar computer as

\[
\begin{align*}
S1 & = \text{ONE} & \text{! ONE holds 1} \\
S2 & = S1 & \text{! } S2 = K = 1 \\
S3 & = N \\
10 & \text{IF( S2.GT. S3 ) GOTO 20} \\
S4 & = B(S2) \\
S5 & = C(S2) \\
S6 & = S4 + S5 & \text{! } S6 = B(K) + C(K) \\
A(S2) & = S6 \\
S2 & = S2 + S1 & \text{! } K = K + 1 \\
\text{GOTO 10} \\
20 & \text{CONTINUE}
\end{align*}
\]

4.2 Vector machine instructions

We now turn our computer into a vector machine through addition of vector registers and appropriate instructions to use them. The eight vector registers V1(64),...,V8(64) shall consist of 64 elements each, and like the scalar registers, hold data of any of the three types defined above. In addition, we require two ancillary registers: The vector length register L which can hold any integer in the range 0 \leq L \leq 64, and the vector mask register M which holds a string of 64 bits. L determines the effective length of the vector registers; e.g. a vector instruction \( V1 = V2 + V3 \) that is executed when \( L = 19 \) will only operate on the first 19 elements of its vectors. Vector instructions which involve the use of the mask register M associate the i-th bit of M with the i-th vector element. Thus the instructions

\[
\begin{align*}
V_n & = z(m::i) & \text{"load vector register"} \\
z(m::i) & = V_n & \text{"store vector register"}
\end{align*}
\]

will transfer data from the array elements \( z(m), z(m+1), \ldots, z(m+i(L-1)) \) to elements \( V_n(1), V_n(2), \ldots, V_n(L) \), or vice versa, whereas

\[
\begin{align*}
\text{IF(M) } V_n & = z(m::i) & \text{"load under mask"} \\
\text{IF(M) } z(m::i) & = V_n & \text{"store under mask"}
\end{align*}
\]

will carry out the transfer only for those vector elements for which the corresponding bit in the mask register M is set to 1.

Another form of memory access is obtained with the instructions

\[
\begin{align*}
V_n & = z(m + V_j) & \text{"gather"} \\
z(m + V_j) & = V_n & \text{"scatter"}
\end{align*}
\]

199
which assume that vector register \( V_j \) holds a set of integers, say \( i_1, i_2, \ldots \). The gather instruction copies \( L \) array elements \( z(m + i_1), z(m + i_2), \ldots \) to the vector elements \( V_n(1), V_n(2), \ldots \); the scatter instruction performs the invers transfers.

There are instructions to load the \( L \) and \( M \) registers, and to examine the contents of \( M \):

\[
\begin{align*}
L &= S_j \\
M &= S_j \\
Sn &= M \\
Sn &= \text{COUNT}(M)
\end{align*}
\]

"load vector length register"

"load vector mask register"

"store vector mask register"

"population count of ones"

For data transfer between scalar and vector registers, optionally under control of a mask in \( M \), we have the instructions

\[
\begin{align*}
[\text{IF}(M)] & \ V_n = V_j & \text{"copy } V_j \text{ to } V_n" \\
[\text{IF}(M)] & \ V_n = S_j & \text{"broadcast } S_j \text{ to } V_n" \\
[\text{IF}(M)] & \ V_n = \text{.NOT.} V_j & \text{"copy complement of } V_j \text{ to } V_n" \\
[\text{IF}(M)] & \ V_n = \text{.NOT.} S_j & \text{"broadcast complement of } S_j \text{ to } V_n" \\
[\text{IF}(M)] & \ V_n = \text{INT}(V_j) & \text{"convert integers in } V_j \text{ to reals"} \\
[\text{IF}(M)] & \ V_n = \text{FLOAT}(V_j) & \text{"convert reals in } V_j \text{ to integers"}
\end{align*}
\]

Arithmetical and logical operations similar to those on \( S \)-registers may be performed on \( V \)-registers:

\[
\begin{align*}
[\text{IF}(M)] & \ V_n = V_j,\text{op.} V_k \\
[\text{IF}(M)] & \ V_n = S_j,\text{op.} V_k
\end{align*}
\]

The contents of two vector registers, or of a scalar and a vector register, may be compared using the instructions

\[
\begin{align*}
M &= V_j,\text{rel.} V_k \\
M &= S_j,\text{rel.} V_k
\end{align*}
\]

where \( \text{rel.} \) is one of the relational operators \( \text{.EQ.}, \text{.NE.}, \ldots \). These instructions set the \( i \)-th bit of \( M \) to 1 if the relation \( V_j(i),\text{rel.} V_k(i) \) or \( S_j,\text{rel.} V_k(i) \) holds, and to 0 otherwise. The contents of two vector registers, or of a scalar and a vector register, may be merged under control of the vector mask register using the instructions

\[
\begin{align*}
V_n &= \text{MERGE}(V_j, V_k, M) \\
V_n &= \text{MERGE}(S_j, V_k, M)
\end{align*}
\]

Element \( V_n(i) \) (\( i = 1, 2, \ldots, L \)) is set to \( V_j(i) \) or \( S_j \) if the \( i \)-th bit of \( M \) is one; otherwise \( V_n(i) \) is set to \( V_k(i) \).

Finally, the instruction

\[
V_n = \text{IOTA}(M) \quad \text{"compress index"}
\]

provides a simplified version of the \( \text{PACK} \) function proposed for Fortran 8x:
IOTA(M) = PACK([1,2,3,...], M)

i.e. if register M has k of its bits set to 1 then the first k elements of Vn will be set to the k integers that specify the positions of the 1-bits in M.

Using the instructions above, the Fortran program

```fortran
REAL A(15,10), B(10), C
DO K = 1, 10
   A(3,K) = B(K) * C
ENDDO
```

might be translated into the language of our vector computer as

```fortran
REAL A(150), B(10), C
L = 10
S1 = C
V1 = B(1::1)
V2 = S1*V1
A(3:15) = V2 ! A(3) = C*B(1), A(18) = C*B(2),...
```

To pack the nonzero elements of an array A into array B, we would have in Fortran 8x

```fortran
REAL A(9), B(9)
LOGICAL M(9)

M(:) = A(:) .NE. 0
K = COUNT( M(:) )
B(1:K) = PACK( A(:), M(:) )
```

and in the language of our vector machine:

```plaintext
L = 9
V1 = A(1::1)
S1 = 0
M = S1 .NE. V1
S2 = COUNT(M)
V2 = IOTA(M)
L = S2
V3 = A(V2)
B(1::1) = V3
```

5. MACHINE ORGANIZATION

In this chapter, our main goal is to establish some basic understanding of those features of a large computer that determine its speed of calculation. In essence, we must see how the sequence of instructions that is determined by the execution of a program is implemented by the hardware of the CPU,
and how instruction execution develops in time as the run of the program progresses. A major difficulty arises from the fact that machine instructions are not really executed one by one, but with much overlap in time. Present-day performance of large computers, vector or scalar, depends essentially on this overlap.

5.1 Overview of instruction processing

To-day even fairly small mainframes consist of multiple memories and several processors (CPUs). The Cray 2 has up to four CPUs, each with a small local memory of 16K words, and with access to a common memory of up to 256M words. The ETA10 has up to eight CPUs, each with 4M words of local memory, and with access to a common memory of up to 256M words. The IBM 3090, with up to six CPUs that each include a (non-programmable) buffer memory, as well as the Cray X-MP and the Cray Y-MP also provide large secondary memories to back up their main memories. To obtain a schematic diagram of the basic CPU organization, we shall disregard the complications that arise from the use of multiple memories and consider only one CPU in its relation to main memory. In fact, the existence of a non-programmable cache memory between the CPU and main memory, as it exists on the 3090, turns out to be more of a hindrance than a help to vector processing.

Typically, the CPU is divided into two main units, sometimes referred to as the instruction interpretation or I-unit, and the instruction execution or E-unit. (See Figure 2) In short, the E-unit performs the programmer's instructions while the I-unit establishes the flow of control and obtains the

![Diagram](image)

*Figure 2: Instruction processing*
instruction stream from memory. The E-unit itself is often divided into a number of functional units, where each unit takes care of a class of similar operations. Instructions that transmit data between memory and CPU registers are performed by a kind of functional unit called memory port. The I-unit examines instructions in the order in which they must be executed and then passes each one to the appropriate functional unit. (The simplicity of this scheme — fully valid for all computers designed by S. Cray since the CDC 6600 — is somewhat blurred when an architecture provides for arithmetic and logical operations that have some of their operands in memory.)

5.2 Clock pulses and clock rates

A synchronous CPU works on a single clock that emits pulses of fixed length at regular time intervals. The time interval from one raising clock pulse to the next one is called a clock period of the CPU. (See Figure 3)

```
  CP 0   CP 1
```

*Figure 3: Clock periods*

The length of the clock period is chosen in accordance with circuit speed, but it depends also on the amount of computation that shall be performed per clock period. It is therefore an indication, but not an absolute measure of circuit speed. Table 1 gives the clock times for a few high-performance computers of their times to indicate the development during the past three decades.

*Table 1: Clock times for some computers*

<table>
<thead>
<tr>
<th>Computer</th>
<th>Year</th>
<th>Clock Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ferranti Mercury</td>
<td>1958</td>
<td>T = 60 000 nsec</td>
</tr>
<tr>
<td>CDC 6600</td>
<td>1964</td>
<td>T = 100 nsec</td>
</tr>
<tr>
<td>CDC 7600</td>
<td>1968</td>
<td>T = 27.5 nsec</td>
</tr>
<tr>
<td>Cray 1</td>
<td>1976</td>
<td>T = 12.5 nsec</td>
</tr>
<tr>
<td>Cray 2</td>
<td>1985</td>
<td>T = 4.1 nsec</td>
</tr>
</tbody>
</table>

The logical significance of the clock period resides in the fact that all registers (programmable or not) of the machine always hold their information constant throughout the clock period; information in a register may change only during the rise of a new clock pulse, as a result of the logic functions performed in the static networks during the preceding clock period. This is the lowest level of parallelism in the machine.
5.3 Memory organization

The basic memory operations require an operand register which holds the word being read or written, and an address register which contains the address of the memory location being referenced. These registers plus associated control logic are often shared between several memory modules. Together, they form a memory bank. For the convenience of addressing, the capacity of a memory bank is always a power of 2. Memory cycle time is the time required by a memory bank to read or write one word. Memory cycle time is not to be confused with memory access time, which is the time required to transfer data between a memory location and a scalar or vector register. A bank conflict is a request for memory access to a bank that currently performs a memory cycle for an earlier request. In such cases the second memory cycle is always delayed until the first one has been completed.

Memory cycle and memory access times are usually expressed as multiples of the CPU clock period. For example, the Cray X-MP (bipolar) memory has a cycle time of 4 clock periods (38 nsec) and an access time of 14 clock periods (133 nsec) for access to the scalar registers. It will be clear from these numbers that a single memory bank could not adequately feed even a single CPU — let alone more. Several memory banks are therefore employed by large mainframes, especially by vector computers, to provide an appropriate memory bandwidth.

A memory of, say, 16 megawords needs addresses of 24 bits. If this memory consists of $2^8$ banks of $2^{16}$ words each, then 8 address bits are needed for bank selection, and 16 bits to address a word within a given memory bank. If the top eight bits of the 24-bit address were used for bank selection, then each memory bank would hold a range of consecutive addresses. As a result, small programs might reside completely within a single memory bank, and even large ones would reference some banks much more often than others, thus causing delays from memory bank conflicts for no good reason. The low-order bits of the memory address are therefore used for bank selection and the high-order bits serve as address within the bank, so that successive memory addresses cycle through the memory banks. This addressing scheme is called interleaving of memory banks. It is common for general purpose computers; for vector machines it has the particular advantage that vectors stored with a stride of one may be read and written with full speed. Vectors whose stride is divisible by a power of two, however, may not be accessed efficiently, especially when this power is large enough. Given, for example, a stride divisible by four, access to the vector would only use one fourth of all memory banks.

Fortran programs should, if possible, take into account that systematic delays from memory bank conflicts become likely when e.g. a twodimensional array A of shape (p,q) is accessed row-wise and the first dimension p of A is even. As a general rule, the sizes of all array dimensions other than the last one should therefore be odd numbers.

5.4 The I-unit: Instruction flow

We now examine more closely the working of the I-unit. (See Figure 4 and Table 2.) Its key elements are a register sometimes called the current instruction word (CIW-) register and a register usually called the program address (P-) register. The CIW-register holds the machine instruction to be executed next. The P-register holds the memory address of this instruction.

Provided the instruction in the CIW is not a jump instruction then the I-unit will increment the contents of P by one during the execution of this instruction, so that P now holds the memory address
Figure 4: I-unit: Instruction flow

Table 2: I-unit action for sample program

<table>
<thead>
<tr>
<th>Register contents</th>
<th>Action by I-unit</th>
<th>Action by E-unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>P CIW</td>
<td></td>
<td></td>
</tr>
<tr>
<td>217 S1 = B</td>
<td>Issue CIW to port</td>
<td>Load B into S1</td>
</tr>
<tr>
<td></td>
<td>P = P + 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Fetch instruction at P</td>
<td></td>
</tr>
<tr>
<td>218 S2 = C</td>
<td>Issue CIW to port</td>
<td>Load C into S2</td>
</tr>
<tr>
<td></td>
<td>P = P + 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Fetch instruction at P</td>
<td></td>
</tr>
<tr>
<td>219 S3 = S1 + S2</td>
<td>Issue CIW to add unit</td>
<td>Put S1 + S2 into S3</td>
</tr>
<tr>
<td></td>
<td>P = P + 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Fetch instruction at P</td>
<td></td>
</tr>
<tr>
<td>220 A = S3</td>
<td>Issue CIW to port</td>
<td>Store S3 into A</td>
</tr>
<tr>
<td></td>
<td>P = P + 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Fetch instruction at P</td>
<td></td>
</tr>
<tr>
<td>221 GOTO 136</td>
<td>Identify jump</td>
<td></td>
</tr>
<tr>
<td></td>
<td>P = 136</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Fetch instruction at P</td>
<td></td>
</tr>
</tbody>
</table>

of the following instruction. If the CIW holds an unconditional jump instruction, the jump address is moved into the P-register, so that instruction execution will continue from there. A conditional jump is executed in two steps: First the jump condition is evaluated, and then either the jump address or the value P + 1 is entered into P, depending on whether or not the jump condition is satisfied. The new value of P thus always holds the address of the following instruction. The I-unit uses this information

---

8 Since we wish to present only the basic principles of instruction sequencing, we simply assume here that each machine instruction occupies one full memory word. In practice, different instructions may well differ in size, and might also be stored across word boundaries. Such complications to the addressing of instructions need not concern us here.

205
to bring this instruction to the CIW, perhaps from some instruction buffer if it is already contained there, or else directly from memory as soon as the present instruction has been issued to the E-unit for execution.

Assume that the Fortran program

```
20    CONTINUE
...
10    A = B + C
      GOTO 20
```

has been translated and stored in memory such that label 10 corresponds to memory address 217 and label 20 to memory address 136, and that the corresponding sequence of machine instructions is that given by Figure 4. The status of the I-unit is then as shown in this figure once program execution has arrived at statement label 10. Holding the instruction \( S1 = B \) in the CIW, and its address 217 in P, the I-unit recognizes a memory fetch instruction. Accordingly, the instruction is issued to the memory port for execution, P is incremented by one, giving \( P = 218 \), and the instruction at this address read into the CIW-register. The operation and those that follow are summarized by Table 2.

### 5.5 The E-unit: Functional units

The subdivision of the E-unit into several functional units is not limited to vector computers; it has in fact been practiced for more than twenty years in order to let instructions of general purpose computers execute in parallel. The functional units of the Cray X-MP are shown together with their execution times in Table 3.

**Table 3: Functional units of the Cray X-MP**

(A second value for the execution time is shown in parentheses where the functional unit takes longer for some instructions than for the others.)

<table>
<thead>
<tr>
<th>Clock periods</th>
<th>Unit</th>
<th>Operand sizes</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>Floating add</td>
<td>64 bits</td>
</tr>
<tr>
<td>7</td>
<td>Floating multiply</td>
<td>64 bits</td>
</tr>
<tr>
<td>14</td>
<td>Floating reciprocal</td>
<td>64 bits</td>
</tr>
<tr>
<td>3</td>
<td>Scalar integer add</td>
<td>64 bits</td>
</tr>
<tr>
<td>3</td>
<td>Vector integer add</td>
<td>64 bits</td>
</tr>
<tr>
<td>1</td>
<td>Scalar logical</td>
<td>64 bits</td>
</tr>
<tr>
<td>2</td>
<td>Vector logical</td>
<td>64 bits</td>
</tr>
<tr>
<td>2(3)</td>
<td>Scalar shift</td>
<td>64 bits</td>
</tr>
<tr>
<td>3(4)</td>
<td>Vector shift</td>
<td>64 bits</td>
</tr>
<tr>
<td>4</td>
<td>Scalar population count</td>
<td>64 bits</td>
</tr>
<tr>
<td>5</td>
<td>Vector population count</td>
<td>64 bits</td>
</tr>
<tr>
<td>2</td>
<td>Address add</td>
<td>24 bits</td>
</tr>
<tr>
<td>4</td>
<td>Address multiply</td>
<td>24 bits</td>
</tr>
</tbody>
</table>
A CPU that has separate functional units for floating addition and floating multiplications can overlap an addition with a multiplication. How about overlapping two additions? One solution to this problem is to provide two or more copies of the same functional unit within the same CPU, and use them e.g. in a round-robin fashion. The IBM 3090 vector facility includes three identical floating add units. Each of them can perform a floating point addition in three clock periods, and so a new addition may be started every clock period.

Operations of the same kind may also be performed concurrently within a single functional unit, provided this unit is segmented. A fully segmented functional unit has separate, logically independent circuits for the actions required during different clock periods of the same operation. The partial results obtained during the first clock period are held during the second clock period in a set of registers that feed an independent set of circuits which perform the next stage of the operation, and so on. Table 4 illustrates the principle, called pipelining, for a three-segment adder that performs the vector addition \( V1 = V2 + V3 \) for vectors of length \( L = 5 \).

<table>
<thead>
<tr>
<th>CP</th>
<th>Segment 0</th>
<th>Segment 1</th>
<th>Segment 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>( V2(1) + V3(1) )</td>
<td>( V2(1) + V3(1) )</td>
<td>( V2(1) + V3(1) )</td>
</tr>
<tr>
<td>1</td>
<td>( V2(2) + V3(2) )</td>
<td>( V2(2) + V3(2) )</td>
<td>( V2(2) + V3(2) )</td>
</tr>
<tr>
<td>2</td>
<td>( V2(3) + V3(3) )</td>
<td>( V2(3) + V3(3) )</td>
<td>( V2(3) + V3(3) )</td>
</tr>
<tr>
<td>3</td>
<td>( V2(4) + V3(4) )</td>
<td>( V2(4) + V3(4) )</td>
<td>( V2(4) + V3(4) )</td>
</tr>
<tr>
<td>4</td>
<td>( V2(5) + V3(5) )</td>
<td>( V2(5) + V3(5) )</td>
<td>( V2(5) + V3(5) )</td>
</tr>
</tbody>
</table>

5.6 Memory access

We now consider the execution of a vector load instruction, \( V1 = a(m:i) \), in order to illustrate the working of a memory port.\(^9\) When the I-unit issues the instruction, the port receives the initial address of \( a(m) \) (the first word to be read), the vector stride \( i \), and the total number \( L \) of vector elements to be read from memory. The port will then request the appropriate memory bank to read from address \( a(m) \). While this occurs, the port successively generates the addresses of successive vector elements \( a(m + i), a(m + 2i), \) etc. and passes read requests for these addresses one by one to the appropriate banks. The request rate can be one request per clock period, as long as no delays are encountered. Delays may, however, occur either because of bank conflicts or because of conflicts in the communication network. In either case the port will repeat the current request until it can be communicated to the corresponding memory bank, and all further requests will be delayed accordingly.

A bank conflict occurs when the port requires use of a memory bank that is currently engaged in a memory cycle either for the present instruction, or for some other instruction of the program that

\(^9\) We assume that the CPU obtains its data from main memory directly, without a cache memory as intermediate storage.
executes concurrently, or for some other CPU, or also for an input/output operation. A network conflict arises when the port cannot communicate with the required bank to forward the next request in sequence during the current clock period. A memory port may fail to connect to a memory bank, although this bank is currently free, because the interconnection network, like the ordinary telephone network, typically shares some of its links between several of its nodes. This situation arises more or less at random due to the traffic between memory ports and memory banks.

Since the memory banks require a fixed number of clock periods to process a read request and forward the data to the memory port, at most one vector element will arrive during a given clock period. Since the requests for \( a(m) \), \( a(m+i) \), etc are generated in sequence, the elements also arrive in sequence at the memory port and then at the vector register.

5.7 Timing of vector instructions

Each time the I-unit issues a vector instruction to a functional unit or a memory port a delay of several clock periods occurs before the first element of the result vector arrives at the result register. This delay is called the vector start-up time of the vector instruction. For a vector load instruction we have just seen that the delay results from the time taken to submit the request for the first vector element to the memory bank, plus the time taken by the memory bank to perform the memory cycle, plus the time to move the vector element from the bank operand register to the vector register. For a vector addition the first pair of operands must be moved from the vector registers to the add unit, the unit then needs several clock periods to perform the addition (see e.g. Table 4), and finally the resulting sum must be transferred to the result register. In general the vector start-up time will exceed the functional unit time by whatever other delays occur in the transmission of operands and results between the vector registers and the functional unit. For example, a vector multiply instruction of the Cray X-MP has a start-up time of 12 clock periods, whereas functional unit time is only 7 clock periods. Start-up time for a vector load is a minimum of 17 clock periods, whereas memory cycle time is only 4 clock periods.

The start-up time of vector instructions will usually be greater than, or at least equal to the total execution time of the corresponding scalar instructions. The time savings due to the use of vector instructions result from the fact that pairs of operands will be sent to the functional unit at a rate of, typically, one pair per clock period, and results will arrive at the result register at the same rate. Execution rate of a vector instruction is the number of results delivered per clock period once the start-up time has elapsed. Typically, the execution rate is equal to the number of segmented functional units (or pipes) used to perform the operation. Exceptions arise, however, for vector instructions that involve memory access, because the execution rate may then well be affected by memory bank conflicts or network conflicts, as discussed in the preceding section. With this exception, the total execution time \( T \) of a vector instruction is linear with vector length \( L \):

\[
T(L) = s + L/r
\]

where \( s \) denotes the vector start-up time and \( r \) the execution rate.

Thus the full speed \( r \) of the pipeline is reached only asymptotically, as the vector length \( L \) increases:

\[
r \approx L/T(L) \quad \text{for} \ L \to \infty
\]

208
Vector start-up times delay overall execution speed, and the delays become more significant as the vector length \( L \) decreases. In order to judge whether the vectors of a given problem are long enough to permit efficient execution it is useful to remember a simple rule of thumb: Only half of the asymptotic execution speed \( r \) will be reached when the vector length is \( L' = rs \):

\[
L'/T(L') = rs/(s + rs/r) = r/2
\]

Large values of \( L' \) signal that the computer will not be efficient on a problem unless the vector length of the problem is correspondingly large. Assuming, for example, \( s = 50 \) and \( r = 4 \) we would have \( L' = 200 \).

### 5.8 Overlap of instruction execution

We return once more to the execution of scalar instructions. Table 3 shows that the arithmetic operations take several clock periods to execute. To these times must be added the time needed to transfer the operands from their registers to the functional units, and also the time for the result to return to its register. On the other hand, the I-unit can generally interpret the instruction in the CTW within one or two clock periods, and therefore could issue instructions to the E-unit at this rate as long as the instruction buffer can keep up the supply of instructions. Overall speed of the CPU thus hinges on the number of arithmetic operations that may be performed concurrently by the E-unit.

Let us assume, for example, that the following two instructions shall be executed in sequence:

\[
S1 = S2 + S3 \\
S4 = S5 \times S6
\]

The I-unit might issue these instructions in successive clock periods, the first instruction to the add and the second to the multiply unit. The two instructions would then execute concurrently (in different functional units) and the result of the addition would arrive in \( S1 \) well after the multiplication has started. This would not affect the outcome of the multiplication, however, since no dependence exists between the two instructions. If, on the other hand, the second instruction were \( S4 = S5 \times S1 \) then the multiply unit would be supplied with a value in \( S1 \) that has not yet been updated by the previous instruction, and the result of the multiplication would be wrong.

### 5.9 S-register reservation

It is thus clear that a computer cannot execute several (scalar) arithmetic instructions concurrently unless the I-unit can recognize dependences between the instructions it issues. For scalar instructions (involving only S-registers) the problem is solved by a reservation system that provides a one-bit flag for every S-register and acts as follows:

(A) **Action at instruction issue**: The result register has its reservation flag set. This flag is cleared as soon as the result of the instruction enters the result register.

(C) **Condition for instruction issue**: Issue of an instruction is delayed as long as the reservation flag is set for any register used by this instruction.
To see the effect of the S-register reservation system, say on register S1, we consider two instructions s and t, not necessarily consecutive, that involve S1. Four cases arise:

1) Instructions s and t both use S1 as operand register:

   s: ... = ...S1...
   ...
   t: ... = ...S1...

No dependence arises from the use of S1 as an operand in instructions s and t. Since S1 does not become reserved upon issue of instruction s, instruction t may issue without delay from S1.

2) Instruction s defines S1, and then t uses S1 as operand register:

   s: S1 = ...
   ...
   t: ... = ...S1...

True dependence arises from the use of S1. The S-register reservation system will set the reservation flag for S1 as instruction s issues, and this flag remains set until the result from instruction s has arrived in register S1. If instruction t should enter the CIW before instruction s has been completed, its operand register S1 will be found reserved, thus preventing instruction t from being issued before instruction s has been completed.

3) Instruction s and t both define S1:

   s: S1 = ...
   ...
   t: S1 = ...

Output dependence arises from the use of S1. The S-register reservation system will set the reservation flag for S1 as instruction s issues, and this flag remains set until the result from instruction s has arrived in register S1. If instruction t should enter the CIW before instruction s has been completed, its result register S1 will be found reserved, thus preventing instruction t from being issued before instruction s has been completed.

4) Instruction s uses S1 as operand register, and then t redefines S1:

   s: ... = ...S1...
   ...
   t: S1 = ...

Antidependence arises from the use of S1. This case is not covered by the S-register reservation system since instruction s will issue without reserving its operand register S1, and so instruction t might issue while s is still in progress. To cover antidependence, all scalar operands are sent to the functional unit as the the instruction issues. The old value of S1 thus enters the functional unit that executes instruction s as soon as instruction s issues, and instruction t is thus free to enter a new value into S1 immediately. No reservation of registers is needed.
Example: \[ S7 = (S1+S2)*(S3*S4) \]

This expression might be evaluated by the instruction sequence:

- s: \[ S5 = S1+S2 \]
- t: \[ S6 = S3*S4 \]
- u: \[ S7 = S5*S6 \]

If we assume that additions take three and multiplications four clock periods, the computation is performed as follows:

<table>
<thead>
<tr>
<th>CP 0</th>
<th>S1 + S2</th>
</tr>
</thead>
<tbody>
<tr>
<td>CP 1</td>
<td>S1 + S2</td>
</tr>
<tr>
<td>CP 2</td>
<td>S1 + S2</td>
</tr>
<tr>
<td>CP 3</td>
<td>S3*S4</td>
</tr>
<tr>
<td>CP 4</td>
<td>S3*S4</td>
</tr>
<tr>
<td>CP 5</td>
<td>S5*S6</td>
</tr>
<tr>
<td>CP 6</td>
<td>S5*S6</td>
</tr>
<tr>
<td>CP 7</td>
<td>S5*S6</td>
</tr>
<tr>
<td>CP 8</td>
<td>S5*S6</td>
</tr>
</tbody>
</table>

Instruction s issues from the CIW at the beginning of clock period 0, and instruction t takes its place in the CIW. Clock periods 0, 1, and 2 are spent to compute the sum S1+S2; the result enters register S5 at the end of clock period 2. Result register S5 is reserved in the meantime. Instruction t is held in the CIW during clock period 0 and is found to be ready since none of the three registers involved in it are reserved. Thus instruction t issues from the CIW at the beginning of clock period 1, and instruction u takes its place in the CIW. Clock periods 1, 2, 3, and 4 are spent to compute the product S3*S4; the result enters register S6 at the end of clock period 4. Result register S6 is reserved in the meantime. Instruction u is held in the CIW during clock period 1 and is found to be not ready since both its operand registers S5 and S6 are currently reserved. The reservation on S5 is cleared upon completion of clock period 2, but the reservation on S6 is held until completion of clock period 4. Thus instruction u issues from the CIW only at the beginning of clock period 5, leaving its place in the CIW to the next instruction in sequence. S7 is reserved during clock periods 5, 6, 7, and 8, preventing any further instructions from reference to S7 in the meantime.

It should be noted that although the S-register reservation system permits independent (and anti-dependent) instructions to execute concurrently, it does not affect the order in which instructions pass through the CIW. Whenever any instruction is delayed in the CIW because of some dependence, all further instructions are blocked behind it, and thus are also prevented from execution no matter whether or not they are involved in any dependences.

5.10 V-register reservation and chaining

It is possible to extend concurrent execution of instructions to vector instructions, but two new problems arise here. One concerns the use made of segmented functional units. Two independent scalar additions that must be performed by two successive scalar instructions can pass through a segmented add unit in successive clock periods, but two independent vector additions (or a vector addition followed by a scalar addition) cannot be performed concurrently by the same functional unit since
the first vector addition will require all duty cycles of the add unit to cope with the stream of operands of the first vector instruction. Any further add instruction, whether vector or scalar, must therefore await the completion of the present vector addition unless the CPU happens to have a second segmented add unit. Moreover, antidependence between vector registers cannot be dealt with in the same manner as antidependence between S-registers since the functional unit accepts only one pair of operands per clock period. The contents of operand vectors must therefore be protected against change by a later instruction as long as the vector instruction is in progress. Several methods have been invented to let some vector instructions execute concurrently. The solution adopted for the Cray X-MP may serve for illustration.

Two one-bit flags, called operand flag and result flag respectively, are available for every V-register. These flags act as follows:

(A) Action at instruction issue:

(A1) The result flag is set on the result register. This flag is cleared as soon as the last element of the result enters the result register.

(A2) The operand flag is set on each vector operand register. This flag is cleared as soon as the last element of the operand has been sent to the functional unit.

(C) Condition for instruction issue:

(C1) Issue of an instruction is delayed as long as any reservation flag is set for the result register of this instruction.

(C2) Issue of an instruction is delayed as long as an operand flag is set for an operand register of this instruction.

To see the effect of this V-register reservation system, say on register V1, we consider two instructions s and t, not necessarily consecutive, that involve V1. Four cases arise:

1) Instruction s uses V1 as operand register, and then t redefines V1:

s: ... = ...V1...

... t: V1 = ...

Antidependence arises from the use of V1. The V-register reservation system will set the operand flag for V1 as instruction s issues. (Action A2) If instruction t should enter the CIW before instruction s has sent all operands of V1 to its functional unit, the operand flag of register V1 will be found set. Condition C1 then delays issue of instruction t until the operand flag for V1 will be cleared.

2) Instructions s and t both define V1:

s: V1 = ...

... t: V1 = ...

212
Output dependence arises from the use of V1. The V-register reservation system will set the result flag for V1 as instruction s issues, and this flag remains set until all results from instruction s have arrived in register V1. (Action A1) If instruction t should enter the CIW before instruction s has been completed, the result flag of register V1 will be found set. Condition C1 then delays issue of instruction t until the result flag for V1 will be cleared.

3) Instructions s and t both use V1 as operand register:

\[
\begin{align*}
  s: & \quad \ldots = \ldots V1 \ldots \\
  t: & \quad \ldots = \ldots V1 \ldots
\end{align*}
\]

No dependence arises from the use of V1 as an operand in instructions s and t. Nevertheless, the V-register reservation system will set the operand flag for V1 as instruction s issues and this flag remains set until all operands of V1 have been sent to the functional unit. (Action A2) If instruction t should enter the CIW before instruction s has sent all operands of V1 to its functional unit, the operand flag of register V1 will be found set. Condition C2 then delays issue of instruction t until the operand flag for V1 will be cleared. Thus the same V-register of the Cray X-MP can never serve concurrently as operand register for two instructions. The second instruction will be delayed accordingly.

4) Instruction s defines V1, and then t uses V1 as operand register:

\[
\begin{align*}
  s: & \quad V1 = \ldots \\
  t: & \quad \ldots = \ldots V1 \ldots
\end{align*}
\]

True dependence arises from the use of V1. The V-register reservation system will set the result flag for V1 as instruction s issues, and this flag remains set until all results from instruction s have arrived in register V1. (Action A1) Even so, if instruction t should enter the CIW before instruction s has been completed, it may issue since the operand flag for V1 has not been set. Thus the V-register reservation system does not cope with true dependence between vector instructions.

Additional logic therefore exists in the Cray X-MP to cope with true dependence of vector instructions, which controls true dependence at the level of individual vector elements. Successive elements of the common vector register are released to the functional unit of the second instruction as the corresponding results of the first instruction arrive from its functional unit. This procedure is called chaining of instructions. It permits a vector instruction t that is linked to an earlier instruction s by true dependence, to start execution as soon as the vector start-up time has elapsed for instruction s. More than two instructions might be chained in this manner, always provided that each of them uses a different functional unit.

Example:

\[
\begin{align*}
  s: & \quad V1 = S1 + V2 \\
  t: & \quad V3 = V1 \cdot V4 \\
  u: & \quad V5 = 1 / V3
\end{align*}
\]

Issue of instruction s sets the result flag of V1 and the operand flag of V2. These flags do not prevent instruction t from issue, but chaining occurs via V1: Result elements that are generated by the add unit for instruction s are passed to the multiply unit (together with the corresponding elements
from V4). As instruction t issues, the result flag of V3 and the operand flags of V1 and V4 are set. Again these flags do not prevent instruction u from issue, but chaining occurs via V3: Result elements that are generated by the multiply unit for instruction t are passed to the reciprocal unit.

The example shows the vector computer at its best. Several operations (in this case an addition, a multiplication, and a division) are performed every clock period, while a strictly sequential scalar computer might require tens of clock periods for every single operation and would not start the next one before the preceding one has been completed. Thus, provided conditions are favourable, the vector architecture might well gain an order of magnitude in speed over the equivalent scalar machine.

6. LOCAL VECTORIZATION OF FORTRAN CODES

6.1 Vectorization of innermost loops

Let

\[
\begin{align*}
&\text{DO } K = M, N, I \\
&s: \quad \ldots \\
&t: \quad \ldots \\
&\ldots \quad \ldots \\
&z: \quad \ldots \\
&\text{ENDDO}
\end{align*}
\]

be an innermost loop which consists of assignment statements only, and let

\[
L = (N-M+1) / I
\]

be its positive trip count. Execution of this loop produces the instruction trace

\[
\begin{align*}
&s_1 \quad s_2 \quad s_3 \quad \ldots \\
&t_1 \quad t_2 \quad t_3 \quad \ldots \\
&\ldots \\
&z_1 \quad z_2 \quad z_3 \quad \ldots
\end{align*}
\]

where \(x_i\) represents the execution of statement \(x\) during the \(i\)-th trip round the loop, and where (scalar) execution proceeds column-wise from top to bottom and from left to right through the rectangular scheme above.

Can we replace the loop partly or fully by vector assignment statements? If statement \(x\) were replaced by a vector assignment then statements \(x_1,x_2,x_3,\ldots\) would be executed in parallel; full vectorization of the loop may thus be seen as proceeding row-wise through the rectangular scheme of statements. A weaker requirement is to split the loop into two successive ones, with some of the statements \(s_1,t_1,\ldots,z\) in the first loop, and the remainder of the statements in the second loop. The problem can be solved with the help of the dependence graph.

We have already established the instruction trace, except that the value of the trip count \(L\) may be data-dependent. (Control statements inside the loop might prevent us from establishing the instruction
However, we may be unable to establish the full list of dependences between all the statements in the loop. For one, the assignment statements may define dummy arguments of the subprogram in which our loop occurs, and these may not all correspond to different actual arguments. The statements may also define or reference array elements that occur with subscript expressions whose values cannot be identified. In addition, the loop may define or reference scalars in a common block and may reference external functions, so that unpredictable side-effects could occur. In any such situation, the compiler can only assume the worst unless we can help it with appropriate directives.

6.2 Induction variables

One way to simplify the dependence graph of a loop, and thus to simplify the vectorization problem for the loop, is to calculate in advance all values of the loop control variable, and of all other variables that go with it, in particular all induction variables. Most compilers will make these changes automatically behind the back of the Fortran programmer. Consider the following example:

\[ K1 = M1 \]
\[ DO K = M, N, I \]
\[ J = 5*K**2 - 7*K + 3 \]
\[ \ldots \]
\[ K1 = K1 + I1 \]
\[ \ldots \]
\[ ENDDO \]

Note that now \( K = M + L*I \)

\( K \) and (assuming that \( I1 \) is not redefined within the loop) \( K1 \) are induction variables that run through \( L + 1 \) values of an arithmetic progression. Although \( J \) is not itself an induction variable, its \( L \) values are fully determined by the first \( L \) values of \( K \). Note that any reference to \( K1 \) inside the loop requires the value of \( K1 \) that is defined during the current trip through the loop if the reference follows the definition. However, if the reference precedes the definition then the value referenced is that from the preceding trip through the loop.

\( K, K1, \) and \( J \) may be evaluated by vector instructions, given the auxiliary arrays

\[
\begin{align*}
\text{DIMENSION NN}(0:L), \; KV(0:L), \; K1V(0:L), \; JV(0:L-1) \\
\text{DATA NN} / \; 0,1,2,3,\ldots,L / \\
\text{KV}(0:L) = M + I*NN(0:L) \\
\text{K1V}(0:L) = M1 + I1*NN(0:L) \\
\text{JV}(0:L-1) = 5*KV(0:L-1)**2 - 7*KV(0:L-1) + 3
\end{align*}
\]

Since the arrays \( KV, K1V, \) and \( JV \) are needed only during execution of the loop the compiler might not assign storage to them in main memory, but hold them in vector registers.
6.3 An illustration of recurrence

The loop

\[
\begin{align*}
\text{DO } K &= 1, N \\
\text{s: } A(K) &= 2 \cdot A(K) + C(K - 1) \\
\text{t: } B(K) &= A(K) - B(K + 1) \\
\text{u: } C(K) &= A(K - 1) + D(K - 1) \\
\text{v: } D(K) &= D(K - 1) - 3
\end{align*}
\]

ENDDO

expands into the following instruction trace:

\[
\begin{align*}
\text{s}_1: & \quad a_1 = 2a_1 + c_0 \\
\text{t}_1: & \quad b_1 = a_1 - b_2 \\
\text{u}_1: & \quad c_1 = a_0 + d_0 \\
\text{v}_1: & \quad d_1 = d_0 - 3 \\
\text{s}_2: & \quad a_2 = 2a_2 + c_1 \\
\text{t}_2: & \quad b_2 = a_2 - b_3 \\
\text{u}_2: & \quad c_2 = a_1 + d_1 \\
\text{v}_2: & \quad d_2 = d_1 - 3
\end{align*}
\]

The definition of \(a_1, b_2, c_1,\) and \(d_1\) produces the following dependences (\(b_1\) is assigned a value, but never referenced before or after):

\[
\begin{align*}
\text{a}_1: & \quad s_1 \rightarrow t_1 \quad s_1 \rightarrow u_2 \\
\text{b}_2: & \quad t_1 \rightarrow t_2 \\
\text{c}_1: & \quad u_1 \rightarrow s_2 \\
\text{d}_1: & \quad v_1 \rightarrow u_2 \quad v_1 \rightarrow v_2
\end{align*}
\]

The definition of further array elements generates the corresponding dependences, and thus need not be considered further.

The dependence graph of the loop (Figure 5) for statements \(s, t, u,\) and \(v\) is obtained as the sum total of all dependences of the execution:

\[
\begin{align*}
s &\rightarrow t \rightarrow t \\
v &\rightarrow v \rightarrow u \rightarrow s \rightarrow u
\end{align*}
\]

![Figure 5: Dependence graph of loop](image-url)
The only statement which does not depend on any other statement is \( v \), and may therefore be executed first. Thereafter, \( s \) and \( u \) depend on each other, forming a cycle in the graph. Once they have been executed, \( t \) may be executed last. Thus we arrive at the following split of the loop:

\[
\begin{align*}
\text{DO } & K = 1, N \\
\text{v:} & \quad D(K) = D(K-1) - 3 \\
\text{ENDDO} \\
\text{DO } & K = 1, N \\
\text{s:} & \quad A(K) = 2A(K) + C(K-1) \\
\text{u:} & \quad C(K) = A(K-1) + D(K-1) \\
\text{ENDDO} \\
\text{DO } & K = 1, N \\
\text{t:} & \quad B(K) = A(K) - B(K+1) \\
\text{ENDDO}
\end{align*}
\]

A cycle in the dependence graph is called a \textit{recurrence}. The statements in the loop cannot be executed alone, and thus cannot be replaced by vector assignment statements. The loop around statement \( t \) may obviously be replaced by the vector assignment statement

\[ B(1:N) = A(1:N) - B(2:N+1) \]

The loop around statement \( v \), on the other hand, is a special case of

\subsection*{6.4 Linear recurrence}

which is expressed by the loop

\[
\begin{align*}
\text{DO } & K = 1, N \\
\text{} & \quad A(K) = A(K-1)B(K) + C(K) \\
\text{ENDDO}
\end{align*}
\]

This loop generates the following series of assignments and (true) dependences:

\[
\begin{align*}
s_1: & \quad a_1 = a_0 b_1 + c_1 & a_1: & \quad s_1 \rightarrow s_2 \\
s_2: & \quad a_2 = a_1 b_2 + c_2 & a_2: & \quad s_2 \rightarrow s_3 \\
s_3: & \quad a_3 = a_2 b_3 + c_3 & \text{...} \\
\end{align*}
\]

It is therefore not at all equivalent to the vector assignment

\[ A(1:N) = A(0:N-1)B(1:N) + C(1:N), \]

which, by definition, has no dependences and replaces the vector \((a_1,a_2,...)\) by the vector \((a_0b_1+c_1,a_1b_2+c_2,...)\). A linear recurrence can therefore not, in general, be expressed by a vector assignment statement.
A special case arises for \( B(K) = 1, C(K) = C \) where \( K = 1, \ldots, N \) and where \( C \) is any constant. Here we obtain from successive substitutions

\[
\begin{align*}
s_1: \quad a_1 &= a_0 + c \\
s_2: \quad a_2 &= a_0 + 2c \\
s_3: \quad a_3 &= a_0 + 3c \\
&\vdots
\end{align*}
\]

and these results may also be obtained by the vector assignment

\[
A(1:N) = A(0) + C*NN(1:N)
\]

in which the array \( NN \) of natural numbers is defined as in section 6.2. Thus the loop around statement \( v \) in section 6.3 may be vectorized.

Other special cases of linear recurrence arise from reduction:

\[
\begin{align*}
A &= \ldots \\
\text{DO} \quad K &= 1, N \\
A &= A*B(K) + C(K) \\
\text{ENDDO}
\end{align*}
\]

If, in particular, \( B(K) = X \) then we arrive at the evaluation of a polynomial by Horner’s method:

\[
\begin{align*}
A &= C(0) & a &= c_0 \\
\text{DO} \quad K &= 1, N \\
A &= A*X + C(K) & a &= c_0x + c_1 \\
\text{ENDDO} \\
& & a &= c_0x^2 + c_1x + c_2 \\
& & & \quad \vdots
\end{align*}
\]

For \( C(K) = 0 \) we obtain the product \( A = \text{PRODUCT}( B(1:N) ) \):

\[
\begin{align*}
A &= 1 & a &= 1 \\
\text{DO} \quad K &= 1, N \\
A &= A*B(K) & a &= b_1 \\
\text{ENDDO} \\
& & a &= b_1b_2 \\
& & & a = b_1b_2b_3
\end{align*}
\]

and for \( B(K) = 1 \) we obtain the sum \( A = \text{SUM}( C(1:N) ) \):

\[
\begin{align*}
A &= 0 & a &= 0 \\
\text{DO} \quad K &= 1, N \\
A &= A + C(K) & a &= c_1 \\
\text{ENDDO} \\
& & a &= c_1 + c_2 \\
& & & a = c_1 + c_2 + c_3
\end{align*}
\]

6.5 Vector sectioning

Sometimes it is advantageous, or even unavoidable in order to achieve vectorization, to restrict the length of vectors to some upper limit \( L \). This limit is called the section size of the vector. In par-
ticular, vector sectioning is always performed by compilers behind the back of the Fortran programmer if the target computer is equipped with vector registers. The loop

\[
\text{DO } K = M, N \\
\ldots \\
\text{ENDDO}
\]

is equivalent to the double loop

\[
\text{DO } J = M, N, L \\
\quad L1 = \text{MIN}(L-1,N-J) \\
\text{DO } I = 0, L1 \\
\quad K = J + I \\
\quad \ldots \\
\text{ENDDO} \\
\text{ENDDO}
\]

For example, the generalized linear recurrence

\[
\text{DO } K = 4, N \\
\quad A(K) = A(K-4)B(K) + C(K) \\
\text{ENDDO}
\]

\[
\begin{align*}
a_4 &= a_0 b_4 + c_4 \\
a_5 &= a_1 b_5 + c_5 \\
a_6 &= a_2 b_6 + c_6 \\
a_7 &= a_3 b_7 + c_7 \\
a_8 &= a_4 b_8 + c_8 \\
a_9 &= a_5 b_9 + c_9 \\
\vdots
\end{align*}
\]

can be reprogrammed as

\[
\text{DO } J = 4, N, 4 \\
\quad L1 = \text{MIN}(3,N-J) \\
\text{DO } I = 0, L1 \\
\quad A(J+I) = A(J+I-4)B(J+I) + C(J+I) \\
\text{ENDDO} \\
\text{ENDDO}
\]

Here, the innermost loop unfolds into the statements

\[
\begin{align*}
A(J) &= A(J-4)B(J) + C(J) \\
A(J+1) &= A(J-3)B(J+1) + C(J+1) \\
A(J+2) &= A(J-2)B(J+2) + C(J+2) \\
A(J+3) &= A(J-1)B(J+3) + C(J+3)
\end{align*}
\]

which are clearly independent and thus permit vectorization.
6.6 Example: Partial sums

Although a sum of more than two terms may not be expressed as a vector assignment statement, it is nevertheless possible to accumulate it partly by means of vector assignments, using the technique of vector sectioning. The scalar product loop

\[
A = 0 \\
\text{DO } K = 1, N \\
A = A + B(K) \cdot C(K) \\
\text{ENDDO}
\]

can first be transformed into

\[
D(1:N) = B(1:N) \cdot C(1:N) \\
A = 0 \\
\text{DO } K = 1, N \\
A = A + D(K) \\
\text{ENDDO}
\]

using the auxiliary array D. Now, to accumulate the elements of D, we may split D into sections of at most length L and form L partial sums in an array E of length L. Thus we obtain for \( L = 3 \)

\[
E(0:2) = 0 \\
\text{DO } J = 1, N, 3 \\
L1 = \text{MIN}(2, N - J) \\
\text{DO } I = 0, L1 \\
E(I) = E(I) + D(J + I) \\
\text{ENDDO} \\
\text{ENDDO} \\
A = E(0) + E(1) + E(2)
\]

Note that the innermost loop is vectorizable. It produces

\[
(e_0, e_1, e_2) = (d_1, d_2, d_3) + (d_4, d_5, d_6) + (d_7, \ldots)
\]

At this point we may ask once more what is meant by "vectorization of a loop". If the intrinsic function \( \text{SUM} \) is considered part of our repertoire of vector instructions then we may conclude that the summing up of a set of numbers (or an inner product of two vectors) \textit{can} be vectorized -- i.e. vectorized with regard to Fortran 8x. Whether the appropriate loops are indeed recognized as being equivalent to a reference of function \( \text{SUM} \) -- and thus whether the loops \textit{will} be vectorized -- depends on our compiler. Whether the instruction repertoire of our computer includes any special hardware facilities that help the accumulation of sums (or products) is yet a different matter: While e.g. the Cray 1 computers and the IBM Vector Facility offer instructions that accumulate partial sums, the Cray X-MP and Cray 2 computers do not. The technique of loop sectioning just presented may be applied by the compiler behind the back of the Fortran programmer in any case. On the other hand, the full speed of a segmented floating point add unit can never be used to accumulate the partial sums given the dependences that remain between these operations, no matter how the program is coded.
6.7 Example: Partial histograms

Vector sectioning can also be useful to alleviate the dependences that prevent histogramming from being vectorized. The code below exemplifies the basic problem:

\[
\begin{align*}
\text{REAL } & \ A(N), B(\text{LOW:IGH}) \\
& B(\cdot) = 0 \\
& \text{DO } K = 1, N \\
& \quad M = \text{INT}( A(K) ) \\
& \quad B(M) = B(M) + 1 \\
& \text{ENDDO}
\end{align*}
\]

The loop involves a true dependence from the first statement (s) to the second (t) and an antidependence from t to s:

\[
\begin{align*}
s_1: & \quad m = [a_i] \\
t_1: & \quad b(m) = b(m) + 1 \quad s_1 \rightarrow t_1 \\
s_2: & \quad m = [a_i] \\
t_2: & \quad b(m) = b(m) + 1 \quad t_1 \rightarrow s_2 \quad s_2 \rightarrow t_2 \\
& \quad \ldots
\end{align*}
\]

Expansion of the scalar M to the vector M(0:4), and the corresponding expansion of array B to array C plus vector sectioning lead to the following code with a vectorizable inner loop.

\[
\begin{align*}
\text{REAL } & \ A(N), B(\text{LOW:IGH}), C(0:4,\text{LOW:IGH}) \\
\text{INTEGER } & \ M(0:4) \\
& C(\cdot,\cdot) = 0 \\
\text{DO } J = 1, N, S \\
& \quad L1 = \text{MIN}(4,N-J) \\
& \quad \text{DO } I = 0, L1 \\
& \quad \quad M(I) = \text{INT}( A(J+I) ) \\
& \quad \quad C(I,M(I)) = C(I,M(I)) + 1 \\
& \quad \text{ENDDO} \\
& \text{ENDDO} \\
& B(\cdot) = C(0,\cdot) + C(1,\cdot) + C(2,\cdot) + C(3,\cdot) + C(4,\cdot)
\end{align*}
\]

Clearly, the overhead inherent in this code will not make it efficient unless the size of A is sufficiently large compared to the size of B.

\[
\ast \quad \ast \quad \ast
\]

221
REFERENCES


