stichting
mathematisch
centrum

MC

AFDELING NUMERIEKE WISKUNDE NN 28/82 NOVEMBER
(DEPARTMENT OF NUMERICAL MATHEMATICS)

B.P. SOMMEIJER

EFFICIENT FORTRAN TECHNIQUES FOR VECTOR PROCESSORS; REPORT ON A SEMINAR

kruislaan 413 1098 SJ amsterdam

Printed at the Mathematical Centre, 413 Kruislaan, Amsterdam.

The Mathematical Centre, founded the 11-th of February 1946, is a non-profit institution aiming at the promotion of pure mathematics and its applications. It is sponsored by the Netherlands Government through the Netherlands Organization for the Advancement of Pure Research (Z.W.O.).

69C12

1980 Mathematics subject classification: 68A05

Efficient FORTRAN techniques for vector processors; report on a seminar

by

B.P. Sommeijer

KEY WORDS & PHRASES: *vector processing, FORTRAN, supercomputers*

PROLOGUE

From September 14-17, 1982 I attended a seminar on: "Efficient FORTRAN techniques for vector processors", held at the Stockholm University Computer Center (QZ), Sweden. This seminar was organized by Pacific-Sierra Research corporation (PSR) in conjunction with QZ. PSR is a private scientific corporation based in Los Angeles, California and is "specialized in all areas of high-speed computation".

The course was designed for the programmer interested in what is called "supercomputers" and the programming techniques for obtaining the highest performance on these machines. The most well-known supercomputers nowadays are the CDC CYBER 205, the CRAY-1 and the CRAY X-MP.

The major aim of this report is to give an overview of the most significant subjects discussed on the seminar, in order to offer my colleagues of the Mathematical Centre a hopefully readable introduction to the efficient use of vector machines. A guideline for this overview was a voluminous work-book (over 500 pages) which was included with the seminar. An introductory part in which the basic ideas of supercomputers are explained is followed by three sections, each describing a particular aspect of using these machines. Connected with each section there is an appendix containing more detailed information about the particular subject. Hence, readers only interested in a rough outline can skip the appendices.

The work-book has been written by John Levesque and Richard Friedmann, both of PSR, who were the instructors of the seminar as well.

I take the liberty to borrow some phrases, examples and timings from this work-book whenever I need them in this synopsis.

INTRODUCTION

In the early days of the computer-age a few hundred arithmetic operations per second could be performed by the machines of that time. Nowadays, we have the so-called "supercomputers" whose speed exceeds 10^8 operations per second. This is the result of the never-ending demand for more speed in scientific and technical environments. This enhancement has been achieved by *hardware advances*, like the prodigious progress in micro-electronics and new hardware design as well as *software advances*, like faster algorithms, highly optimizing compilers but also by restructuring programs in order to take advantage of the new hardware design.

Large-scale problems as they frequently occur in disciplines like ecology, economics, aerodynamics, meteorology etc. require the solution of systems of equations with a tremendous number of unknowns in order to get a sufficiently detailed simulation for engineering purposes. Many algorithms for solving this kind of problems allow the concurrent calculation of several operations. This brings us to the supercomputers which are designed to perform calculations simultaneously. Basically, there are two different approaches: *parallel processing* and *pipelining*.

The idea behind *parallel processing* is that a program which uses N processors can run N times faster than the same program using only one processor. As an example of parallel processing, consider the following DO loop:

```
                   DO 10 I = 1,N
(1)                A(I) = B(I) + C(I)
                10 CONTINUE
```

Hence, having N identical processors, all additions can be executed simultaneously, each addition by one processor.

Pipeline computers are essentially analogous to assembly lines in which the product runs through a number of segments each performing simultaneously one stage of the manufacturing process.

To illustrate this concept we reconsider the DO loop (1); traditionally the following code must be executed for each pass through the loop:

Load B(I)/Load C(I)/Add/Store A(I)/Increase I and Branch

When executed in this order various parts of the processor are idle much of the time, e.g. while waiting for operands to be loaded, the adder is not in use. The principal idea of pipelining is to keep busy all parts of the processor continuously. Hence, the adder can act on the elements B(I) and C(I) while A(I-1) is stored, and B(I+1) and C(I+1) are fetched from memory.

Both types of machines are so-called *vector computers* built to speed up the execution of *vector operations*, which means operating on vectors (i.e. groups of values) with *one* instruction. The most famous vector computers being the CRAY-1 and the CYBER 205 are both pipelined. An example of a parallel processing machine is the ILLIAC IV.

In the forthcoming we will need some definitions: a vector's stride is the number of memory locations between consecutive vector elements. A vector with a stride of one is said to be contiguous. As we will see in section 2, there is a significant difference in the way the CRAY-1 and the CYBER 205 deal with non-contiguous vectors. Digital computers are equipped with a clock omitting pulses at a fixed interval called the clock cycle. At each tick of the clock the state of the machine is unambiguously determined and nothing happens in less than one clock cycle. Although computer manufacturors try to minimize the clock cycle, this cannot be shortened unrestrictedly. The speed of light limits how fast signals travel (about one foot per nanosecond), which limits how far components can be separated. However, because closely packed components generate much heat, the biggest problem in building low clock cycle machines is cooling them.

To express the speed of vector processors we use megaflops, which means millions of floating point operations per second. To give an impression, nowadays scalar computers operate with a megaflop rate of at most 5, while the CRAY-1 and CYBER 205 have a peak rate of about 100 megaflops or even larger.

Supercomputers, intended to perform large-scale computations possess large memories. In order not to slow down the speed of execution, the transfer rate from and to memory, known as the bandwidth should also be high. This problem has been treated differently on the CRAY-1 and the CYBER 205. One of the major differences in the architecture of both machines is the way vectors are streamed through the pipes. The CYBER 205 is a so-called memory-to-memory machine, which means that vectors are fetched from memory, pushed through the pipes where the actual operations are performed and stored back into memory again. The CRAY-1 on the other hand is a register-to-register machine; this means that vectors are put into vector registers before and after streaming through the pipe. Because the CRAY-1 is equipped with vector registers containing (maximally) 64 floating point numbers, the curve of megaflops as a function of the vector length shows peaks for the CRAY-1 (see figure 1).

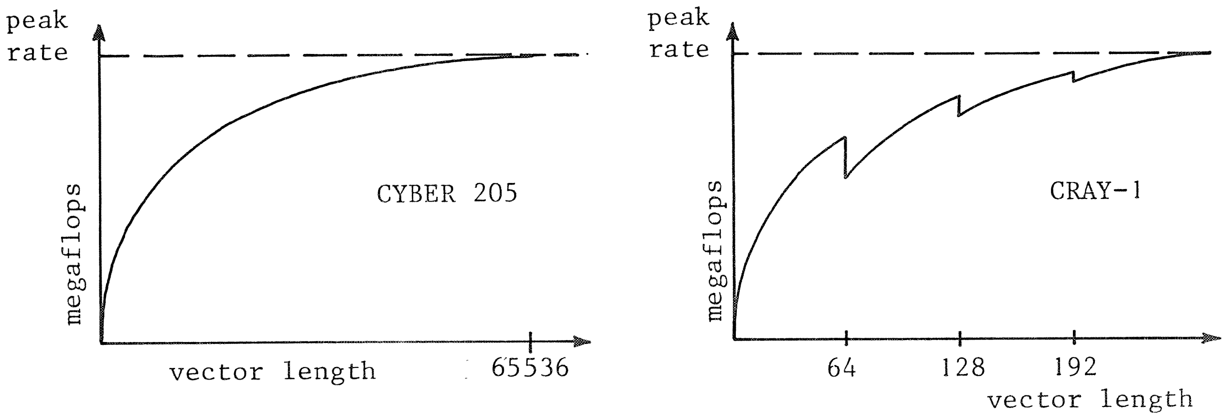


figure 1. Characteristic figures of megaflop rates for CYBER 205 and CRAY-1.

The peak rates mentioned by the manufacturors are seldomly reached because of several reasons: one reason is that a startup time is required in a vector operation, that is the time elapsed before the first result leaves the pipe (The startup time is independent of the length of the vectors involved). For the CRAY-1 this startup time is considerably lower than for the CYBER 205 but after having processed 64 vector elements, the CRAY-1 needs a new startup, while the CYBER 205 can handle vectors as long as 65536. Another facet is that for most vector operations the time per result is less for the CYBER. Hence, CYBER 205 outpaces CRAY-1 when vector length exceeds a certain value.

Another reason for not obtaining peak rate is that usually only part of the code can be vectorized. For example, it is readily verified that vectorizing half of the code into vector mode with an infinite speedup over scalar mode and leaving the other half in scalar mode only an overall speedup of 2 is obtained.

It is not always possible to vectorize a code, which is obvious from the following example

```
      DO 20 I = 1,N
      A(I+1) = A(I) + S
   20 CONTINUE
```

Because in vector processing the arguments must be determinable prior to the operation, this loop cannot be vectorized. This restriction shown here is known as recursion; it conflicts with the nature of vector processing.

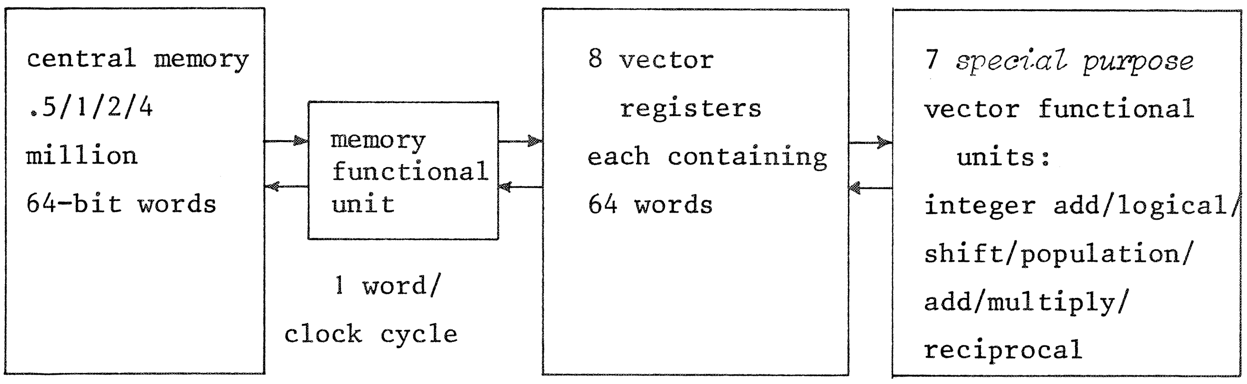
This overview is concluded with some statistics:

	CRAY-1	CYBER 205	CRAY X-MP
year	1976	1981	1982
clock cycle in nanoseconds	12.5	20	9.5
wordsize in bits	64	64	64
memory size in millions of words	.5/1/2/4	.5/1/2/4	2/4
memory bandwidth in gigabits	5.1	25.6	40.4
installed or on order	46	16	1

Before studying the several aspects of supercomputers in more detail, I like to quote a statement of Seymour Cray: "Nobody, and I mean nobody, knows how to program large, parallel machines". I emphasize that the judgement of this statement is left to the reader.

1. ARCHITECTURE

In this section I confine myself to the architecture of the CYBER 205 and the CRAY-1. Also some attention is paid to the CRAY X-MP, a very recent design of CRAY corporations. Let us start with the CRAY-1 architecture. Basically the CRAY-1 looks like:



Because only one word per clock cycle can pass the memory functional unit, the bandwidth at the CRAY-1 is $64/12.5_{10}{-9} = 5.1$ gigabits/second, which turns out to be small in relation to the megaflop rate which can be achieved when data is held in the registers (This bottleneck has been removed in the CRAY X-MP, having 6 memory functional units, hence 6 words can be passed every clock cycle (= 9.5 nanosecond for X-MP) resulting in a bandwidth of 40.4 gigabits/sec.).

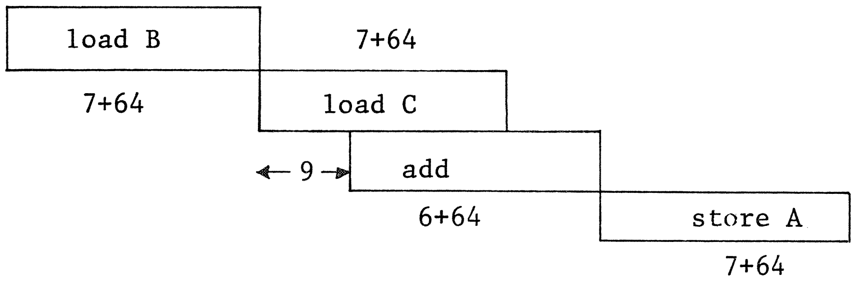
An important feature of the CRAY-1's vector hardware for compensating the low bandwidth is chaining, which means that vector functional units can be "chained together", i.e. output of one immediately becomes input of another. Let us consider the following DO loop:

```
      DO 30 I = 1,64
      A(I) = B(I) + C(I)
   30 CONTINUE
```

which actually results in

	(in clock cycles)	
	startup time	time per result
Load B	7	1
Load C	7	1
Add B and C	6	1
Store A	7	1

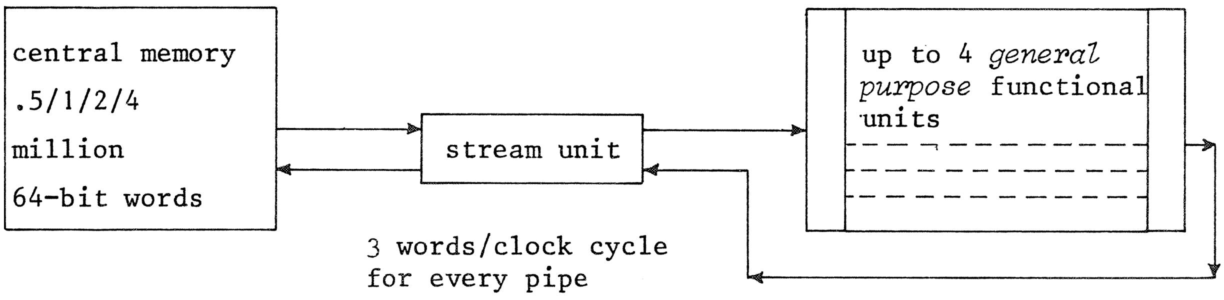
Because only one memory functional unit is available on the CRAY-1, the load, load and store cannot be performed concurrently. However, the addition of B and C can start when loading C. So, we obtain the following situation:



Hence, the total time for this example equals 29+3*64 clock cycles, which means that this DO loop can run at a 64/ (221*12.5 10^{-9}) = 23 megaflop rate. Because the CRAY X-MP can chain all three operations, the resulting megaflop rate is 70.9. A restriction of chaining on both CRAYs is that one particular vector functional unit (add, multiply etc.) can only appear once in the chain.

It should be noted that the disk transfer rate is very low (estimating conservatively, one disk read takes the same time as 100 arithmetic operations in vector mode). Hence, when dealing with very large data structures the strategy "re-compute old values rather than reading them in from disk" may become actual.

The CYBER 205 has, just like the CRAY-1, a scalar processor as well as a vector processor. The first one is of register-to-register type, similar to the CRAY-1 but the vector processor is of memory-to-memory type. The last concept can be pictorized as follows:



The bandwidth of the CYBER 205, which has a clock cycle of 20 nanoseconds, is for a two-pipe version (actually, at this moment no four-pipe machine has been installed yet) 19.2 gigabits/second. The floating point functional units, which are all identical, can add, multiply, devide and take square roots. However, the multi-pipe concept does not imply that more than one (vector) instruction can be performed simultaneously. For example, when executing a vector addition with two pipes (that is the usual situation) all vector elements with an odd index run through one pipe and the addition of all "even elements" is performed in the other pipe. But the *effect* of a two-pipe concept is that the megaflop rate is doubled.

I like to mention two nice features of the CYBER 205. The first one is the ability to deal with words of 32 bits. This decreases of course the accuracy but for most operations the 32-bit mode is twice as fast as the 64-bit mode.

Another feature is the possibility to work with linked triads, i.e. a triadic operation that involves one scalar operand and two vector operands (e.g. expressions like: A(I) = B(I) + C(I) * SCALAR). Again, the megaflop rate is approximately twice as large as in the case of a dyadic operation. As said before, for most operations the startup time is relatively large on the CYBER 205, but the time per result is fairly small. On the CYBER 205, vectors can be as large as 65536 ($= 2^{16}$).

Considering the same DO loop as used in the description of the CRAY-1, we will now calculate the megaflop rate of the CYBER 205, having a clock cycle of 20 nanoseconds:

startup time for vector addition: 51 clock cycles
time per result (using two pipes): $\frac{1}{2}$ clock cycle

Consequently, the total time for a vector addition of 64 elements is 83 clock cycles, yielding a megaflop rate of $64/(83*20\ 10^{-9}) = 38.6$. Hence, for this sample loop, a vector length of 64 appears to be sufficient for the CYBER 205 to outpace the CRAY-1.

We summarize the architecturial idiosyncrasies of both machines:

Similarities:

both have a 64-bit word (sign (1), exponent (15), mantissa (48))
both are pipelined
both have vector instructions

Differences:

CRAY-1	CYBER 205
- register-to-register	memory-to-memory
- same floating point functional units for vector and scalar instructions	different processors for vector and scalar instructions
- allows strides	only contiguous data
- fixed memory size	virtual memory
- vector chaining	linked triad capability
- special purpose functional units	general purpose functional units
- short vector startup time	long vector startup time

There are several other machines with quite different architectures but their discussion is postponed till the appendix.

2. LANGUAGES

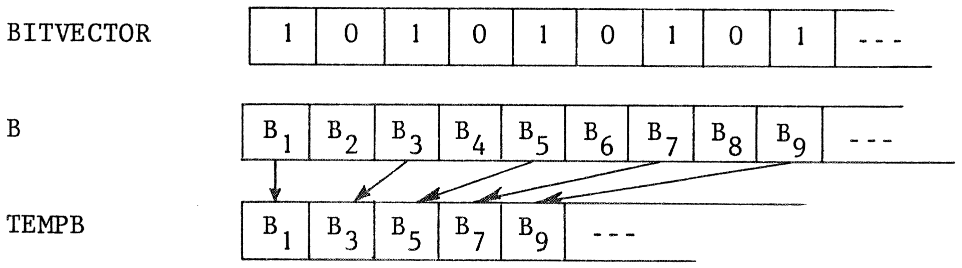
So far, vector computers are merely equipped with FORTRAN compilers, because FORTRAN is world's most widely used language in technical applications.

In order to get vector instructions out of the compiler we distinguish between: implicit vectorization, which means that we write DO loops as we use to do for scalar machines and rely on "automatic" vectorization by the compiler. Although FORTRAN-compilers are very smart nowadays, there frequently arise situations in which the compiler does not recognize the possibility to vectorize the loop and will generate scalar object code (see the sample loops in the appendix concerning the vectorization inhibitors for the various compilers). In such cases one should *restructure* the loop, that is rewrite the source in such a way that the inhibitors corresponding to the compiler one is going to use, are no longer applicable. This approach has the advantage that standard FORTRAN can be used which benefits the transportability of the program as well. However, when this procedure cannot be pursued (for whatever reason) one should perform the vectorization-process by hand which is called explicit vectorization. To that end, both the CRAY-1 and the CYBER 205 have a special vector syntax available. I will discuss some of the possibilities using the CYBER 205 syntax:

As mentioned in the introduction, CRAY-1 and CYBER 205 deal quite different with operations on vectors having a *stride* unequal to one. The CRAY-1 compiler can vectorize this situation. Moreover, the megaflop rate is independent of the value of the stride. On the contrary, the CYBER 205 can only deal with contiguous vectors; hence, the following DO loop cannot be vectorized:

```
      DO 10 I = 1,N,2
      A(I) = B(I) + C(I)
   10 CONTINUE
```

We first have to make the data contiguous. For that purpose there are some bit manipulating functions available. We can set a bit vector with ones in the positions corresponding to the items we need and zeros elsewhere. There is a *compress* function which can be pictorized as


BITVECTOR
1 0 1 0 1 0 1 0 1 ---
B
B_1 B_2 B_3 B_4 B_5 B_6 B_7 B_8 B_9 ---
TEMPB
B_1 B_3 B_5 B_7 B_9 ---

and which stores (under the control of the bitvector) the required elements of B contiguously into a temporary array TEMPB (all at vector speed). The vector C is treated similarly. Now, the actual addition can be performed and stored into a temporary array, say TEMPA. After completion of this DO loop, the vector TEMPA has to be *expanded* into the vector A. It will be clear that the use of strides favours the CRAY-1 above the CYBER 205. Because a compress and expand take 52 + z/2 and 58 + z/2 clock cycles, respectively (z is the number of elements compressed or expanded) we arrive for this example with N = 64 (see also section 1) on a megaflop rate of

$$(N/2)\ /\{((52 + N/4) * 2 + (51 + N/4) + (58 + N/4)).20_{10}{-9}\} = 5.8$$

For the CRAY-1 the megaflop rate for this example equals

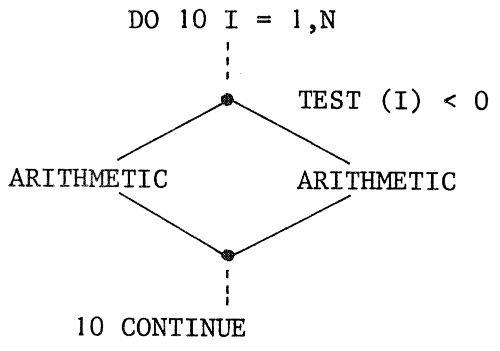
$$(N/2) / \{(29 + 3 * N/2) \cdot 12.5 \; 10^{-9}\} = 20.5$$

Another reason why explicit vectorization might be useful is the case of nested DO-loops. Both the CRAY-1 and the CYBER 205 compiler usually vectorize only the innermost loop.
Let us consider the example

```
   DO 20 J = 1,N
   DO 20 I = 1,M
   A(I,J) = B(I,J) + C(I,J)
20 CONTINUE
```

The innermost loop (running with I) acts on columns of matrices, i.e. on contiguous data. This is easily recognized by the CYBER 205 compiler as being vectorizable. However, because M may be less than the row dimension of the arrays, both loops cannot be collapsed into one long vector of length N * M and the compiler will omit collapsing. Explicit vectorization, using a bitvector, may be helpful in such cases.

Another frequently occuring situation is the presence of a conditional statement within the DO loop, which forces the compiler to abandon vectorization. A possible way for explicit vectorization of the loop



may be: translate (at vector speed) the IF test into a bit vector of length N, in which the i-th bit is set if test (i) < 0 and not set otherwise. Now, compress, under the control of the bitvector, the active data used in both branches into temporary vectors, do full efficiency arithmetic, and conclude with a merge back into the original data structure.

A second possibility to solve the difficulty of an IF statement is the use of the WHERE/OTHERWISE/END WHERE-construction.

Because in standard FORTRAN storage mode a vector is stored columnwise, a nice feature in the CYBER 205 syntax is the use of the ROWWISE declaration, which informs the compiler that arrays are to be stored with the right-most subscript varying fastest. Hence, ROWWISE allows rows of matrices to be accessed contiguously.

3. OPTIMIZATION

Before describing some indications to restructure FORTRAN programs in order to speedup the execution, I like to make some general remarks on optimization: It is inherent to the nature of modern high-speed computers that achieving fast-result rates requires rather much effort from the user. Another draw-back in vectorizing a program is that we frequently introduce a lot of overhead (e.g. setting bitvectors, introducing temporary arrays etc.) and that the surveyability of the program diminishes considerably in most cases.

A first step in optimization is the determination of the major time-consuming routines (for this, CRAY-1 offers a handy tool named "flow trace") and to consider the important parameters in it (array dimensions, DO loop length etc.).

I will now summarize some proposals which might help the compiler to recognize the code as being vectorizable, a few of them being specifically applicable to scalar mode. However, because a vector computer is a scalar computer as well, it is also worth to consider scalar optimization in order to speedup the overall execution. For some of these suggestions (marked with an asterisk) the appendix to this section contains an example in which the original source, the restructured version as well as a comparison of the timings is given.

A. DO LOOPS

1. change IF loop to DO loop
2. unroll loops
3. jam loops

* 4. use first dimension as innermost loop

* 5. switch loops

* 6. avoid testing on the DO loop index value

B. <u>SUBROUTINES AND FUNCTIONS</u>

* 1. pull into loops

* 2. use statement functions instead of external routines

3. avoid long parameter lists (pass arguments in COMMON if possible)

C. <u>ARITHMETIC EXPRESSIONS</u>

1. factor expressions to reduce number of operations
2. minimize number of divides

* 3. avoid double precision, or use sparingly

4. use data statements in place of initializing assignment statements
5. avoid mixed mode expressions

D. <u>IF STATEMENTS</u>

1. remove loop-independent IFs from DO loop
2. use IF statements instead of computed GOTOs for less than four branches
3. change loop-dependent conditional forward transfer to vector mode

E. <u>INPUT/OUTPUT</u>

$\ast\left\{\begin{array}{l}\text{1. avoid small I/O requests; use buffers to move large chunks}\\ \text{2. use small I/O lists}\\ \text{3. use unformatted I/O where possible}\\ \text{4. overlap processing with I/O by using buffer in/buffer out}\end{array}\right.$

5. avoid I/O inside a loop where possible

F. <u>MISCELLANEOUS</u>

* 1. split out recursiveness

* 2. statement reordering

CONCLUSIONS

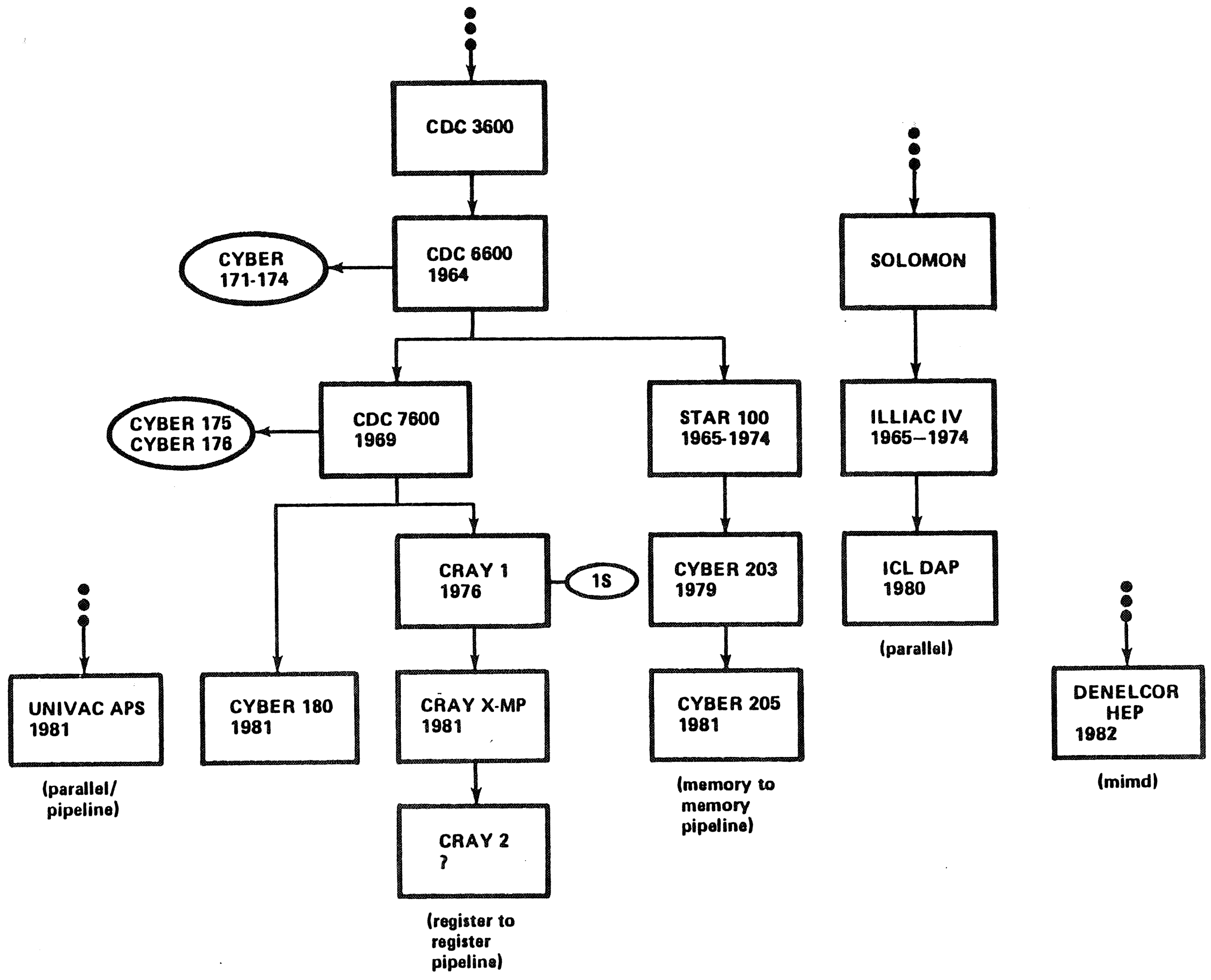
Although peak rates are rarely obtained, a considerable speedup can be achieved using a vector type machine. However, to utilize the advantages of these computers, usually a lot of effort is asked from the user. This is a consequence of the fact that the current compilers could not keep pace with the hardware enhancements. Moreover, in properly using a vector computer, programmers have to change their programming-habits and should not be afraid to introduce some overhead (memory as well as executions). Another aspect of explicit vectorization (i.e. using the additional vector syntax) is that the resulting source code often no longer looks like FORTRAN. Hence, one may wonder wether FORTRAN is the most suitable language for vector machines (VECTRAN ? According to my instructors, APL may be a useful alternative).

At the same time it is worth to consider the idiosyncrasies of vector computers in developing (numerical) algorithms. Some aspects which deserve attention are (i) choose data structures which can be processed efficiently on vector machines (use contiguous data or, at worst, constant strides) and (ii) avoid algorithms giving rise to recursive loops.

Finally, after comparing so frequently the behaviour of the CRAY-1 and the CYBER 205, one may wonder which machine is the fastest one. It depends; it depends on so many things (e.g. vector length, data structures used, contiguous data or not, use of indirect addressing etc.) that it may be too dangerous to designate one of them as being superior to the other.

Appendix to section 1.

In this appendix we give more detailed information concerning the architecture of vector computers. Let us start with a historical overview:



In order to get an impression about the approximate timings of several vector operations on the CYBER 205, the CRAY-1 and the CRAY X-MP we have the following table:

Total time = startup + Time Per Result * vector length
(timings in nanoseconds)

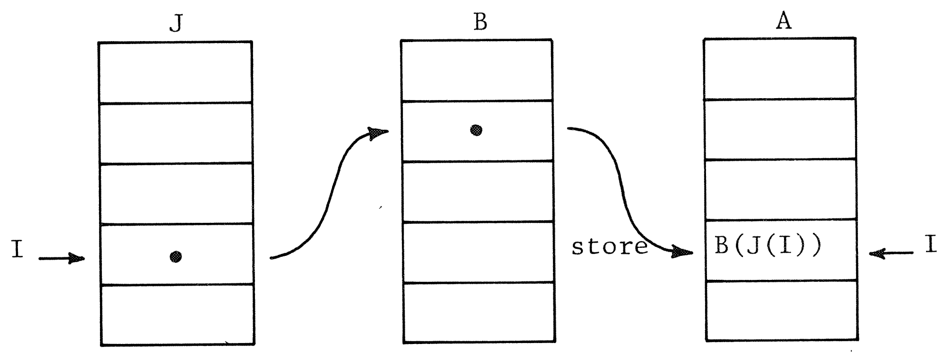
	CYBER 205 (2 pipes; 64-bit words)		CRAY-1		CRAY X-MP	
vector operation	startup	TPR	startup	TPR	startup	TPR
contiguous						
X(I) + Y(I)	1020	10	$\lceil N/64 \rceil *412.5$	37.5	$\lceil N/64 \rceil *266$	9.5
X(I) + S	1020	10	$\lceil N/64 \rceil *325+125$	25	$\lceil N/64 \rceil *266$	9.5
X(I) + Y(I)*Z(I)	2060	20	$\lceil N/64 \rceil *550$	50	$\lceil N/64 \rceil *370.5$	19
X(I) + Y(I)*S	2060	10	$\lceil N/64 \rceil *462.5+125$	37.5	$\lceil N/64 \rceil *418+95$	9.5
X(I) / Y(I)	1600	71	?	37.5	?	9.5
SQRT(X(I))	1580	71	?	?	?	?
gather	780	25	not needed		not needed	
periodi-cally						
scatter	1420	25	not needed		not needed	
gather	1380	25	2000	125	?	?
randomly						
scatter	1660	25	3500	100	?	?
compress	1040	10.f	not needed		not needed	
expand	1160	10.f	not needed		not needed	

we make some notes about this table:

- by $\lceil x \rceil$ we mean the smallest integer greater or equal to x
- a division on the CRAY machines is performed in two steps: first a reciprocal approximation is made, followed by a multiplication
- the number f in the compress/expand result is the fraction Z/N, where Z is the number of elements compressed or expanded and N is the vector length
- the gather-scatter-timings for the CRAY-1 are obtained using library routines
- gather-scatter instructions are used for indirect adressing. An example of the *gather* instruction is

```
      DO 10 I = 1,N
      A(I) = B(J(I))
   10 CONTINUE
```

where J is an integer array. The situation can be visualized as



Concomitant to the gather operation, there is a *scatter* operation, like B(J(I)) = A(I) ■

As promised in section 1, some alternative computer designs are discussed.

First, there are the FUJITSU's VP-100 and VP-200. They are, just like the CRAY-1, register-to-register machines however with multiple pipelines (up to 12) and flexible length of the vector registers (up to 1024 words). In vector mode, these machines can attain 250 and 500 megaflops, respectively, and 10-15 megaflops in scalar mode.

Next, we have the class of *attached array processors* (AP) which can be coupled to a scalar machine (termed the *host*). This AP is called by the host whenever a vector operation has to be performed. The most successful AP is the AP-120 from Floating Point Systems (over 2,000 units sold), having a word length of 38 bits. This AP-120 has recently been replaced by the AP-164, having a 64-bit entry. Another well-known AP is that of UNIVAC, which attaches to UNIVAC 1100/80 machines and shares memory with the host. It has very high speed processing (peak rate of 120 megaflops) and identical floating point representation as the host (36-bit). Concerning the performance of attached processors we remark that calling the AP takes fairly much time; hence, one should bundle up a significant amount of calculations and perform this in one invocation to the AP.

So far, all processors discussed were of pipeline-type. Now, we discuss some alternative approaches: Denelcor *HEP* (Heterogeneous Element Processor), originally conceived as a digital replacement for analog computers in solving systems of ordinary differential equations, has evolved to a "general purpose" MIMD (multiple-instruction-multiple-data) machine. MIMD computers have a set of interconnected processors, in which each processor can execute one instruction (affecting many different data) independently of the others. All pipelined machines discussed so far are of SIMD (single-instruction-multiple-data) type. This HEP, equipped with 64-bit words and a 100-nanosecond clock cycle, is expected in 1982.

Because most programs have a considerable amount of potentially exploitable parallelism, it is worth to consider the concept of *data flow:* "execute an instruction as soon as its operands are available". Hence, the sequence of the calculations, until now determined by the programmer, becomes unpredictable; in other words, there is no longer a program instruction counter as in the von Neumann sequential computer design. Because the compilers based on the current (serial) languages cannot detect all the parallelism, we need a new language to utilize the advantages of data flow machines.

Finally, we discuss a *scalar* processor based on the *Josephson junction* (discovered by Josephson in 1962), which is a superconducting (zero resistance) device consisting of tunnel barriers with a thickness of only 40 Angstroms (about 20 atomic layers).

These devices must operate at the very low temperature of 4^oK (liquid helium). However, this enormous cooling has some benefits too: low power dissipation (7 Watt) and a ninetyfold reduction of thermal noise, which increases the reliability. IBM is developing a "Josephson-computer" which fits into a cube with edges of 14 cm. It is going to have a 16-megabytes memory and a megaflop rate of 70.

Appendix to section 2

On the CYBER 205 some forms of recursiveness are recognized by the compiler and transformed into a STACKLIB call, that is very efficient *scalar* object code is generated. Examples are:

```
      DO 10 I = L,M                 (recursive add)
   10 X(I) = X(I-1) + Y(I)

      DO 20 I = L,M                 (summation)
   20 S = S+ X(I)

      DO 30 I = L,M                 (inner product)
   30 S = S+ X(I) * Y(I)
```

Next, I will summarize a number of reasons which inhibit implicit vectorization (i.e. automatic vectorization by the compiler of standard FORTRAN). To that end we need some definitions:
constant increment integer (cii): an integer variable that has a constant value added to it once, each pass through a DO loop
invariant: a scalar that is referenced but not defined in a DO loop
array constant: an array element whose subscripts are invariants or constants
vector array reference: an array element in which one subscript contains a cii and all others, if any, are invariants or constants
scalar temporary: a scalar variable defined by a vectorizable expression each pass through a DO loop
example: C and D are scalar tempories in:

```
      DO 40 K = 25,50
      C = A(K)
      D = B(K) * C
   40 X(K) = C**D
```

array constant temporary: an array constant used as a scalar temporary
reduction function scalar: a scalar variable or array constant used to hold cumulative information about a whole vector (example see 20-loop above); on CYBER only a scalar variable is allowed
indirectly addressed array: an array whose subscript is an integer vector expression, like A(IA(I))
order dependency: a value is destroyed before it can be used; example

```
      DO 50 I = 1,2
      A(I) = C(I)
   50 B(I) = A(I+1)
```

scalar mode: A(1) = C(1); $\underline{\text{B(1) = A(2)}}$; $\underline{\text{A(2) = C(2)}}$; B(2) = A(3)
vector mode: A(1) = C(1); $\underline{\text{A(2) = C(2)}}$; $\underline{\text{B(1) = A(2)}}$; B(2) = A(3)
recursion: values calculated during previous loop passes feed back into the current pass
ambiguous subscripts: invariants in subscript expression whose value could be positive or negative which might lead to recursion;
example: J is ambiguous in

```
      DO 60 I = 5,50
   60 B(I) = B(I+J)
```

In table 2.1 the inhibitors for vectorization are listed, both for the CRAY-1 compiler and for the CYBER 205 compiler.

INHIBITOR		CRAY-1	CYBER 205
A	CALL	x	x
B	IF	x	x
C	GOTO	x	x
D	I/O	x	x
E	order dependency	x	x
F	ambiguous subscript	x	x
G	non-linear array reference	x	x

		CRAY-1	CYBER 205
H	indirect addressing	x	x
I	recursion	x	STACKLIB
J	more than one subscript having a cii	x	only (I,I) allowed
K	array constant temporary	x	
L	cii other than loop index		x
M	cii used before being set		x
N	complicated cii expression	x	x
O	cii used in expression	x	x
P	illegal reduction function	x	x
Q	vector too long		x
R	use of equivalences		x
S	illegal loop index		x
V	loop vectorizes		

Table 2.1

We now give a collection of sample DO loops; for both compilers it is indicated wether the particular loop is (letter V) or is not vectorizable (the letters refer to the inhibitors of table 2.1)

	CRAY-1	CYBER 205

```
DO 1 I = 1,100
T = A(I) + B(I)
1 R(I) = T + C(I)/T
```
	V	V

```
J = 10
DO 2 I = 1,N
J = J + 1
2 R(I) = A(J) * B(I)
```
	V	L

	CRAY-1	CYBER 205

`DO 3 I = 3,M`
`3 Q(I) = A(I) + Q(I-1)`
	I	STACKLIB

`DO 4 J = N,M`
`A(K) = B(J) * C(J)`
`4 R(J) = (A(K) -.5)**2`
	K	V

`DO 5 I = 1,100`
`5 R(I) = A(I) * 2. + SQRT(B(I))`
	V	V

`DO 6 I = 1,N`
`J = I + 5`
`6 R(J) = 1.0`
	V	L

`DO 7 I = 1,N`
`7 R(I) = A(I) + I * 2.`
	O	O

`DO 8 I = 1,N`
`8 IF (ITEST.EQ.1) R(I) = A(I) * B(I)`
	B	B

`DO 9 I = 1,N`
`R(I) = A(I) + T`
`9 T = T + B(I)`
	P	P

`DO 10 I = 1,700000`
`10 R(I) = A(I)**2`
	V	Q

`DO 11 I = 1,2`
`11 R(I) = A(I)/B(I)`
	V	V

`DO 12 I = 1,2000`
`12 R(I) = A(I*2) + B(I+3)`
	V	V

```
                                                   CRAY-1          CYBER 205

      DO 13 I = 1,N
   13 R(I) = A(I,I) + B(I) * SCA                     J                V

      DO 14 J = 1,N
      DO 14 I = 1,N
      R(I,J) = 0.0
      DO 14 K = 1,N                                  V                P
   14 R(I,J) = R(I,J) + A(I,K) * B(K,J)

      EQUIVALENCE (A,B)
      DO 15 I = 1,N                                  V                V
   15 R(I) = A(I) + B(I)

      EQUIVALENCE (A,R)
      DO 16 I = 1,N                                  I                R
   16 R(I+1) = A(I) + B(I)

      DO 17 I = L,M
   17 R(I) = A(I+5)**3+B(I-N)*SCA + C(M)             V                V

      DO 18 I = 1,N
   18 R(I) = A(M-I)                                  V                S

      DO 19 I = 1,N
   19 A(I*2)= A(I**2) + B(I-2)                       G                G

      DO 20 J = 1,N
      I = I + 2                                      H                H
   20 R(J) = A (IB(I)) + C(J) *.5

      DO 21 K = 1,N
      I = I + 2                                      V                L
      R(J) = A(I)
   21 J = J + 6
```

```
                                           CRAY-1        CYBER 205

   DO 22 I = 1,100
   R(I) = A(I)                               E              E
22 A(I+1) = B(I)

   DO 23 I = 1,N
23 R(I) = SIN (COS(SQRT(A(I)) + 1.))         V              V

   DO 24 I = 1,N
   M = (I-1) * N + J                         N              L
24 A(I) = B(M)

   DO 25 I = 1,N
25 A(I) = B(I) / A(N+1)                      F              F

   DO 26 I = 1,N
   A(I) = SQRT (B(I))                        D              D
26 PRINT 800, A(I), B(I)

   DO 27 I = N,1,-1
   II = N - I + 1                            I              I
27 A(I) = A(II) + B(I)
```

Appendix to section 3

In this appendix we give some examples of restructuring source code. The letters and digits refer to the classification made in section 3. For each example we give the original code, the restructured code, the timings in microseconds (which are real timings most of the time), the speedup (defined as "old time/new time") and some comments. The timings were obtained by using CFT 1.09 on the CRAY-1 and FORTRAN 2.0 on the CYBER 205.

```
                                                CRAY-1         CYBER 205

A.4    original:  DO 10 I = 1,10
                  DO 10 J = 1,10
                  DO 10 K = 1,10                 176              256
               10 R(I,J,K) = A(I,J,K)

  restructured:   DO 10 K = 1,10
                  DO 10 J = 1,10
                  DO 10 I = 1,10                 148              11.2
               10 R(I,J,K) = A(I,J,K)

                             speedup             1.2              22.9
```

comment: because A and R are dimensioned as A(10,10,10), R(10,10,10) the CYBER 205 recognizes anything to be contiguous and collapses all loops to one loop, running from 1 to 1000.

```
                                                CRAY-1         CYBER 205

A.5    original:  DO 20 I = 1,100
                  DO 20 J = 2,100                4650             4530
               20 A(I,J) = A(I,J-1) * B(I,J)

  restructured:   DO 20 J = 2,100
                  DO 20 I = 1,100                 529              234
               20 A(I,J) = A(I,J-1) * B(I,J)

                             speedup              8.8              19.3
```

comment: because the innerloop is no longer recursive, the restructured form can be vectorized.

```
                                                   CRAY-1       CYBER 205
A.6   original:  DO 30 I = 1,100
                 A(I) = B(I) + C(I)
                 IF (I.LT.10) A(I) = A(I) * S1      159            172
                 IF (I.GT.90) A(I) = A(I) * S2
              30 IF (I.EQ.IREV) A(I) = -A(I)

  restructured:  DO 30 I = 1,100
              30 A(I) = B(I) + C(I)
                 DO 31 I = 1,9
              31 A(I) = A(I) * S1                   7.5            5.6
                 DO 32 I = 91,100
              32 A(I) = A(I) * S2
                 IF(IREV.LE.100) A (IREV) = -A(IREV)

                           speedup                 21.2           30.7
```

comment: the original code must run in scalar mode whereas the restructured code is vectorizable.

```
                                                   CRAY-1       CYBER 205
B.1   original:  DO 40 I = 1,100
                 CALL INIT (R(I))
              40 CALL  CALC(A(I), B(I), C(I), R(I))

                 SUBROUTINE CALC (W,Y,Z,X)
                 X = X + W*Y + Z ** 2
                 RETURN
                 END                                348            442

                 SUBROUTINE INIT (X)
                 X = 1.0
                 RETURN
                 END
```

```
                                                  CRAY-1       CYBER 205
                                                  ------       ---------
  restructured:  DO 40 I = 1,100
              40 R(I) = 1.0 + A(I)* B(I) + C(I)**2   8.4          8.3

                           speedup                41.4         53.3
```

comment: the restructured code can run in vector mode and eliminates subroutine overhead.

```
                                                  CRAY-1       CYBER 205
                                                  ------       ---------
B.2   original:  DO 50 I = 1,100
              50 R(I) = B(I) * FUNC(A(I)) + C(I)

                 FUNCTION FUNC (X)                 213          294
                 FUNC = X**2 + 2.0/X
                 RETURN
                 END

  restructured:  FUNC(X) = X**2 + 2.0/X
                 DO 50 I = 1,100                   13           13.5
              50 R(I) = B(I) * FUNC(A(I)) + C(I)

                           speedup                16.4         21.8
```

comment: the compilers insert in-line code at each reference to a statement function rather than executing a function call; this technique alse benifits the possibility to vectorize.

```
                                                  CRAY-1       CYBER 205
                                                  ------       ---------
C.3   original:  DOUBLE PRECISION X(100),Y(100),Z(100), DSUM

                 DO 60 I = 1,100
                 Z(I) = DSQRT (X(I)**2 + Y(I)**2)
                 DSUM = DSUM + Z(I)                2770         13000
              60 CONTINUE
```

		CRAY-1	CYBER 205
restructured:	DOUBLE PRECISION DSUM DO 60 I = 1,100 Z(I) = SQRT (X(I)**2 + Y(I)**2) DSUM = DSUM + DBLE (Z(I)) 60 CONTINUE	503	221
	speedup	5.5	58.8

comment: the double precision software on the CYBER 205 seems to be very complex.

E. That optimizing I/O may be useful is illustrated by the following example (only timings of the CRAY-1 are available):

		TIMINGS in *milli*seconds
	REAL X(10000)	
1.	WRITE (1,91)X 91 FORMAT (5E20.10)	328
2.	WRITE (1)X	63
3.	BUFFEROUT (1,1) (X(1), X(10000)) J = UNIT (1)	51
	REAL X (10240)	
4.	BUFFEROUT... J = UNIT	37
5.	BUFFEROUT... J = UNIT dataset assigned to buffer memory device	9.75

comment: selectively using I/0 instructions is very significant which may be clear from the fact that the DO loop

```
      DO 1 I = 1,1450000
    1 A(I) = B(I) + C(I)
```

takes as much time as the second write instruction (63 MS).

```
                                                          CRAY-1      CYBER 205

F.1   original:     DO 70 I = 2,100
                    A(I) = B(I) * SQRT(A(I)**2 + C(I)**2)*.0001
                 70 D(I) = D(I-1) * A(I) + C(I)             283          187

  restructured:     DO 71 I = 2,100
                 71 A(I) = B(I)* SQRT(A(I)**2 + C(I)**2)*.0001
                    DO 72 I = 2,100                          69           52
                 72 D(I) = D(I-1)*A(I) + C(I)

                               speedup                      4.1          3.6
```

comment: note that the 70-loop and the 72-loop must run in scalar mode and that the 71-loop can run in vector mode.

```
                                                          CRAY-1      CYBER 205
F.2   original:     DO 80 I = 1,100
                    A(I) = B(I) * 2.
                    B(I+1) = C(I)/D(I)                      73.6         120
                 80 CONTINUE

  restructured:     DO 80 I = 1,100
                    B(I+1) = C(I)/D(I)                      13.4         7.0
                    A(I) = B(I) * 2.
                 80 CONTINUE

                               speedup                      5.5          17.1
```

comment: it is easily verified that vectorizing the first loop would result in wrong values for the elements of A, hence this loop must run in scalar mode. However, by reordering the statements the loop can be vectorized.

36461