This is a preprint of a paper intended for publication in a journal or proceedings. Since changes may be made before publication, this preprint is made available with the understanding that it will not be cited or reproduced without the permission of the author.

UCRL - 74761

PREPRINT

CONF- 730706--1

22,085

LAWRENCE LIVERMORE LABORATORY

University of California/Livermore, California

FUTURE TRENDS IN COMPUTER HARDWARE

**E. A. Feustel
C. A. Jensen
F. H. McMahon**

May 22, 1973

NOTICE

This report was prepared as an account of work sponsored by the United States Government. Neither the United States nor the United States Atomic Energy Commission, nor any of their employees, nor any of their contractors, subcontractors, or their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness or usefulness of any information, apparatus, product or process disclosed, or represents that its use would not infringe privately owned rights.

This paper was prepared for presentation at the Computational Fluid Dynamics Conference sponsored by the American Institute of Aeronautics and Astronautics at Palm Springs, Calif., July 19-20, 1973.

MASTER

DISTRIBUTION OF THIS DOCUMENT IS UNLIMITED

109

FUTURE TRENDS IN COMPUTER HARDWARE*

E. A. Feustel, C. A. Jensen, and F. H. McMahon
Lawrence Livermore Laboratory
Livermore, California

Abstract

Vector processors, parallel processors, and sequential processors with much concurrency are considered. Some real codes can be shown to be almost completely suitable for parallel and vector processors. However, vector to scalar speed ratios can be so large for the vector and parallel machines that even a small residual scalar content is a serious problem. Monte Carlo approaches and implicit differencing schemes yield shorter vectors than explicit differencing schemes, causing problems due to long vector operation start up times.

Programming in a vector language may be a more global and easier to understand approach to programming. It is necessary to use such an approach in order to attain some reasonable fraction of the potential speed of a vector or parallel processor.

Introduction and Summary

The utility of various scientific processor architectures is weighed against a set of codes used at Lawrence Livermore Laboratory (LLL). On the basis of experience gained in the development of parallel programs, vector machines, parallel machines, and concurrent machines are considered.

The idea of utility is interpreted to relate to the question of how useful the maximum speeds of processors are when an entire program, rather than small parts, such as inner loops, is the weighting factor. Utility is also associated with the range of classes of numerical techniques which can be comfortably and easily mated to a given processor architecture.

It is desirable to consider ease of programming as an aspect of utility. It has not been possible to consider this as quantitatively as the other questions. But some comments are made on the subject. The future may bring better understanding of general programming approaches simultaneously suited to many classes of processor. For this paper, it is usually assumed that coding is done in an appropriate assembly language, or in a language especially well mated to a given architecture.

The point of view has been adopted that processing speed in computational physics is closely related to the average number of floating point operations done per unit of time. It is recognized that this approach is not entirely correct, but it is better than relating speed to instructions executed per unit time.

Hardware designers face common technological limits. Various design approaches have been the

result of the competition to achieve maximum utility within these limits. As a result, parallel processors, vector processors, and serial processors with instruction execution overlap (concurrency) have emerged as contenders for maximum scientific utility.

Code Parameters

Code parameters have been separated into two sets. One set relates to constraints on memory hierarchical characteristics. The other set involves the interaction of the central processor and memory. It is not claimed that these parameters are more than indicators of needs and performance.

The first such parameter set concerns memory requirements and affects memory sizes, hierarchies, transfer rates, and management. For the codes which consume most of the machine time at LLL, the parameters given in Table 1 are correct.

Table 1. Memory size and hierarchical characteristics.

Average number of floating point operations per variable per code cycle	20-80
Size of variable space in primary storage (in words)	100,000-350,000
Total size of variable space (in words) ^a	100,000-2,000,000
Average I/O rate ^b	5 kHz-100 kHz

^a 60-bit words.

^b This number is tape and printer oriented I/O. It does not include swapping or overlaying. The needs of the latter problem are suggested by the first item.

Clearly, the major codes at LLL are large. Looking ahead, there is only one major code whose designers do not think will require more than 500,000 words of primary storage in the future. The other major codes have already exceeded this in total size, or will, or their designers would like them to do so.

Tape and printer oriented I/O rates are low, but the first parameter in Table 1 indicates that primary to/from secondary storage rates must be very high, if a code overflows into secondary storage. For example, with an attainable vector processor, code could run at an average of 30 million floating point operations per second and could do 30 floating point operations per variable.

*This work performed under the auspices of the U.S. Atomic Energy Commission.

Then, as the information for a new cycle is brought in and that for an old one taken out, bi-directional primary-secondary storage transfer must proceed concurrent with computations at 1 million words per second. For the future, this kind of rate seems to be most appropriately obtainable through block secondary storage and an associated appropriate data bus structure, rather than through sole reliance on disc as secondary storage.

Table 2 gives some information that is primarily of interest for use with the vector and parallel machines. A "medium" length vector is about 500 words. Length of 200 words or less would imply a "short" vector. This table was derived from actual operation and vector counts. It is not intuitively obvious that the codes could be vectorized as completely as the table indicates.

Hardware Descriptions

In a recent paper, M. J. Flynn⁽¹⁾ has given four broad hardware classifications. While these are idealized extremes and no one machine fits entirely in a given category, they are useful for discussion purposes. They are:

1. The single instruction stream-single data stream (SISD) organization. This characterizes most computing equipment today and is the archetype "serial" or strictly sequential machine.
2. The single instruction stream-multiple data stream (SIMD) organization. This is

the archetype parallel machine, and ILLIAC IV with its single instruction decoder driving 64 processing elements is the prototype realization. However, Control Data's STAR and Texas Instrument's (TI) ASC also mostly fall into this class.*

3. The multiple instruction stream-single data stream (MISD) organization. Only a few special purpose machines fall in this category, and most are the "dual" in instruction-data space of the SISD organization.
4. The multiple instruction stream-multiple data stream (MIMD) organization. Any multiple processor system which permits processor-processor communication and synchronization falls into this category, although it is not limited to such arrangements. Among the very large scientific machines only the experimental quad-processor CDC 8600 fits in this organization. However, the multiple pipe TI ASC's have some aspects of this organization, as do the peripheral processor in the TI ASC and the set of them in the CDC 7600.

*Reference to a company or product name does not imply approval or recommendation of the product by the University of California or the U.S. Atomic Energy Commission to the exclusion of others that may be suitable.

Table 2. Properties of coding intended for vector or parallel processors.

Code	Description	Size of variable space (in words)	Long and medium vectors (>200 words) %	Short vectors (≤200 words) %	Scalar operations %	32 bit operations %	Logic operations %
No. 1	Elastic-plastic flow with slip planes using explicit differencing	up to 1,000,000	94	6	15	0	20
No. 2	Hydrodynamic using both explicit and implicit (10% of ops) differencing	up to 350,000	90	10	?	70	10
No. 3	Monte Carlo simulation of particle transport + hydrodynamic	up to 2,000,000	~100	~0	?	85	~15
Codes being written	More Monte Carlo + more reliance on implicit differencing schemes	>1,000,000 for the Monte Carlo codes	Implicit differencing and complex Monte Carlo reduce percentage and length of vectors			?	?

Concurrent machines, such as the CDC 7600, serve as a useful reference point. This is because some experience has been built up on them, and because, for the 7600, there is an interesting "Dr. Jekyll - Mr. Hyde" character which allows it to seem pure SISD or pure SIMD.

Our experience is that for the 7600, a FORTRAN compiler produces code yielding from about one floating point operand per microsecond to as much as 5.5 floating point operations per microsecond.⁽²⁾ However, the average for an entire code is not usually likely to exceed three floating point operations per microsecond.

At LLL, a subset of the arithmetic instruction set of CDC's STAR has been emulated on the 7600 by Frank McMahon.⁽³⁾ Lansing Sloan has also emulated some triadic operations such as $A + B * C + D$, where A, B, C, and D are vectors.⁽⁴⁾ This STAR emulation uses the 7600 instruction stack, overlapped instruction issue capability, and in-stack looping to achieve around eight floating point operations per microsecond for the dyadic operations (e.g., the STAR instruction set) and around eleven floating point operations per microsecond for the triads. These rates are essentially identical to what a good handcoder will get for most problems - irrespective of whether or not he adopted this "vector" viewpoint.

When operated using these emulations (labeled "in-stack loops," or simply "stack loops") almost exclusively, we call the 7600 a "vector 7600." These emulations increase the result rates by factors of from 1 to 5, depending on whether or not the original coding was in good hand code, or on how well the programmer, the problem, and the compiler interacted.

This discussion of the 7600 calls attention to a feature shared by all present architectures, i.e., the speed ratio between code that makes the machine work well and code that doesn't is very large. For the 7600 the ratio is about 5 to 1, and a good FORTRAN can get 30% or more of the 7600's potential. This ratio is likely to be an order of magnitude larger for pure vector or parallel approaches.

Vector machines are based on the idea that, if the same operation is to be repeated many times on different operands, then one can arrange the machine in an assembly line fashion. In this arrangement, a part of a processing step is done at each stage of the assembly line during a machine cycle. As for a conventional assembly line, the idea is to keep all the parts as busy as possible to maximally utilize the resources of the system.

Such processors may be divided into four segments: 1) the instruction decoder, 2) the buffers to hold operands coming to and from memory, 3) the arithmetic "pipe" segment, where the actual arithmetic is done and 4) (recognizing that the approach may not always be appropriate) a set of internal hardware registers for holding scalars.

Memory bandwidth is most economically attained by arranging the memory in banks (interleaving) and with word sizes which are large multiples of the CPU word size. Memory banks can then be started up one machine cycle apart,

even if the memory cycle time is longer than the machine cycle time. The number of CPU words in a memory word is closely matched to the CPU result time so that the necessary two CPU word fetches and one CPU word store can occur per CPU operand result. The memory buffers are the devices which function as the "elastic bands" to keep all these input and output operand synchronized between CPU and memory. Thus a certain ladder-like arrangement begins to appear in which operands are put in their place on a buffer queue, awaiting their turn for the next processing step.

The same ladder or queue arrangement shows up in the instruction decoder itself. This decoder may not do all the decoding of an instruction in one cycle. Because the processor is terribly complex, instruction conflict resolution is very difficult and the best that can be done is conflict detection. If this is the case, the instruction decoder halts until a conflict is resolved. Also, instructions must keep their place in the queue. That is, even if a scalar result is needed quickly and its generation involves no conflicts, the generating instruction is forced to remain in queue in the instruction decoder.

The arithmetic pipe is constructed to give high assembly line (vector) speed for all operations with minimum hardware cost. This entails an arrangement which takes much longer to produce a given arithmetic result than an arrangement with independent arithmetic functional units.

In contrast to the interleaved high bandwidth memory arrangement, economy may dictate that the internal CPU registers share a single bus. In this case, only one operand can be stored or fetched from these registers per CPU cycle.

Thus, the result of all the emphasis on producing a number crunching assembly line is that each step taken to speed up the vector process rates slow down scalar processing. Vector start up times may also be increased, so that the minimum length vector which can be efficiently processed is lengthened.

The effect is to make such a processor suitable only for nearly pure vector problems with substantial vector lengths. For example, we have defined a "medium" length vector as one which requires approximately 10% of the total processing time for start up. This definition, combined with all the queues which have to be followed in start up, makes medium length vectors on the order of hundreds of elements long for any foreseeable architecture.

For scalar work, the limit on rates is dictated by the nature of the scalar instruction set (e.g., register-to-register, register-to-memory, etc.), the bandwidth of associated buses (e.g., CPU register, memory buffer), and the number of machine cycles per instruction issue. This limit may be a sizable fraction of the vector rate. However, to achieve this limit, one must be dealing with highly concurrent coding, so that one is "streaming" the scalar instruction, and their operands through the various queues, and there must be an appropriate number of registers readily available to contain operands. While the details will be heavily dependent on the processor

architectural details, the probable result is that for most real world instances nothing like the maximum scalar operand result rate is likely to be maintained for useful periods of time.

An associated problem with scalars, sufficiently important to receive special attention, is the scalar fetch time from memory. For the reasons of hardware economy discussed above and on account of the ladder-like or assembly line architecture, a scalar and its associated instruction decode must pass through many steps on the way from memory to a CPU destination, taking one machine cycle per step. In consequence, memory fetch time may have little relationship to the actual time it takes to get a scalar to a useful working point. In fact, for known realizations of the vector approach, one could set the memory fetch time to zero and it would not substantially affect the time it takes to get a scalar from memory to the CPU. (The real utility of a short memory cycle is in a reduction of the size and complexity of the buffer connecting memory and the CPU, which then does, as a side effect, reduce effective fetch time.) Hence, there are real problems with scalar code if it is not concurrent but has substantial serial character. Further, as the fraction of scalars is reduced and one approaches a pure vector computation, the remaining small fraction of scalars will be less and less concurrent.

The point of all this is that vector processor computational power may be cannibalized by a large fraction of scalar coding, but even a small fraction will have a deleterious effect all out of proportion to its size.

This same discussion should make it clear that, since the vector processors amount to exceptionally strung out concurrent processors when dealing with scalars, experience with higher level languages on concurrent machines should apply here as well.

It is known that, for the vector processor architecture, compilers which do not have vector instructions in the source language will have difficulty generating them in any but the simplest cases. And the compiler is faced with a formidable task for the case of generating efficient scalar code. There does not seem to be any substantial evidence as yet that compilers can overcome these difficulties in any useful way.

Another problem, due to the interleaved memory, wide bus, and long memory cycle time (if present), is the slow speed of any operation whose operands are not (almost) consecutive in storage. For example, matrix transposes would be done slowly.

The question of mapping parts of vectors into parts of other vectors is a serious one. In the classical scalar machine this is done by indexing. However, it is not clear that indexing buys very much in the vector approach. Indexing is likely to involve throwing away much memory bandwidth because of the large memory word size compared to the CPU word size. In addition, a vector of indices may be generated which will consume a substantial amount of storage space. For reasons discussed below, the bit control vector approach seems to be more useful to mapping.

All present vector and parallel machines use bit control vectors for logic operations. That is, if arithmetic operands in a vector (or set of vectors) are to be split into two branches, then a vectorized arithmetic logic operation is done to the appropriate vector (or vectors), generating a bit string of ones and zeros indicating the result of the test for each operand or pair of operands. The bit string is then used to generate new shortened vectors by pulling out appropriate operands (under control of ones or zeros) for each branch from the original vectors. The arithmetic for each branch is done separately and the results merged together again under the control of the bit vectors. No actual program branch occurs, because the shortened vectors for each logical branch are given unique names and operated upon unconditionally. A similar procedure occurs for the simple problem of indexed operations: Bit vectors are generated; these bit vectors then control the generation of arithmetic vectors, which are operated upon.

For this kind of operation a rich instruction set is very useful, including instructions which operate on bit vectors themselves by other bit control vectors. In general, our conclusion is that bit control vectors are the most important non-arithmetic capability required in computational physics by a vector or parallel processor. It is worth pointing out that the bits should be stored in memory at maximum density or the size of memory devoted to bit strings can become excessive.

For most cases, bit control vectors (or strings) replace indexes in vector operations.⁽⁵⁾ But the utility and power of the idea may not be immediately clear. For example, the bit string which is the Boolean result of an arithmetic comparison between two vectors may be described simply in an appropriate language and generated by one instruction in hardware. If subsequent operations are to be based on this comparison, then the bit string represents a well defined and compact mapping. For the general case, indexes may not be compact, and it may be difficult to prove an indexing scheme is well defined. Furthermore, the bit string does not care how irregular a mapping it describes, whereas indexes lose utility once a fair degree of irregularity is present. For one code which deals with many fluids interacting in a turbulent viscous system, the minimum required bit vector storage space is 20,000 60-bit words (with 40,000 words being typical). It would probably not be possible to deal with this problem using indexes.

The parallel processor concept is older than the vector processor concept and its best realization — ILLIAC IV — is well described in the literature.⁽⁶⁾ In many ways the parallel processor can be compared directly to the vector processor. The ratio of vector to scalar speeds is somewhat more clearly defined, because it will be approximately equal to the number of parallel processing elements. Hence, one cannot hope to attain high scalar speeds due to concurrency. However, in return, the simpler memory to processor interface substantially reduces operand fetch times compared to vector processors (although at a higher memory cost). To our thinking these aspects may be one of the more important real differences between the vector and parallel approach. We see two other major differences. One is the modulo-number-of-processing elements efficiency factor of the parallel

machine, which generates discontinuities in the overall efficiency as a function of data set size. The other is processing element-to-processing element operand routing problems and complexity. The segmented memory of ILLIAC IV is unfortunate, but isn't necessarily an essential feature of the parallel approach.

Some Programming Considerations

For someone brought up with sequential programming, the vector machines seem hard to deal with. However, our experience is that, assuming the problem is appropriate for these machines, it can be programmed in a straightforward manner with a more global (vector) point of view. But one must abandon the scalar approach. Thus, even for the 7600, a program has to be re-thought and re-written to take advantage of the stack loops. The utility of the stack loop approach for the 7600 might not have been recognized had people not been thinking about STAR.

We especially want to make the point that we have seen no convincing evidence that any of these architectures will run efficiently on code generated by recompiling existing programs.

Architecture

The serial machines get fast scalar capability by way of memory word sizes which are the same as CPU word sizes, by multiple memory interleaving, and by simple interfaces between CPU and memory. They lose vector speed on account of the small number of CPU registers, the single bus between CPU and memory, and the inability of the instruction processor to issue instructions at high rates.

The vector or pipeline machines get fast vector capability through memory word sizes which are large multiples of CPU word sizes, multiple memory interleaving, and staged CPU processing pipes, which can each be kept busy all the time doing a given part of an operation. The price paid is a ladder-like architecture convenient for vectors, but poor for scalars, because the scalars have to keep their place on the ladder, even though they may be needed quickly and may require little processing.

The Solomon or parallel machine can avoid the ladder problem. But there will always be a vector-to-scalar speed ratio approximately equal to the number of processing elements.

The most serious oversight we can see in the present stream machines is the explicit use of CPU registers to contain vector parameters. It seems to us the vector parameters should have been in the form of a vector descriptor stored in memory, not necessarily at the head of the vector.⁽¹⁾ This would give a "tagged" vector architecture which would allow easier memory management and the possibility of doing something on the order of an APL-like interpreter indistinguishable from an APL-like compiler. Memory management with the vector machines would sensibly be based on managing the vectors themselves, rather than pages or sections of memory which bear no relationship to the vectors. Further worthwhile improvements might result from treating code segments as vectors. An

already extant example of managing both vectors and code segments in this way is the Burroughs Corporation approach.

In the course of this study we have considered other architectures. Two are listed below in approximate order of increasing difficulty.

- A multiple processor approach with the processors sharing one or two vector functional units. The processors would be concurrent and have bit control vector capability.
- A 7600-like machine with more registers, greater memory bandwidth, and the capability of issuing more than one instruction per cycle (with finer ability to distinguish between real and apparent conflicts). Bit vector capability should be a part of this machine as well. This is not unlike the now-abandoned IBM ACS machine.

There are several reasons for looking closely at the second structure. As a result of the work on stack loops at ILL, it has become clear that the use of stack loops on a machine that is basically serial, but has high concurrency, can allow the emulation of vector instructions. This approach is more general, in terms of the kinds of vectors which can be readily handled, than the approach used in vector or parallel machines.

The limitations to the speed of a stack loop are:

- a) the machine cycle time,
- b) the number of instructions which can be issued per cycle (in particular, multiple load and store commands would be useful),
- c) the number of CPU registers, as this helps determine the complexity of the loop and the degree of anticipation of needed operands,
- d) the length of the instruction stack,
- e) the number of functional units, the rate at which each unit can accept operands, and the number of cycles required to deliver an operand, and
- f) loop control overhead.

The chief resource which one cannot fully use in this approach is the bandwidth of the functional units because there must generally be some idle units in order for the approach to work. However, because the CPU register and instruction issue capability would become relatively more complex in this approach, and because there is already unused functional unit bandwidth in the successful 7600, this should not be a problem. A very expensive aspect of this approach would be the interleaved, multiple bus, single CPU word access memory interface.

What one is after is the ability to generalize the concept of a vector so that any sequence of operations which could be performed with high speed and efficiency would be regarded as an element of a "vector." As this sequence was iterated, the elements of the "vector" would be processed. Then, for example, the inversion of a small

matrix, sometimes important in implicit differencing schemes, could be a single element in a generalized vector. The user would identify the "elements" of the "vectors" of value to him. These would be hand coded as stack loops. Then a macroprocessor could be used to link the identification of these elements in a high level language with the hand coded processing loops.

Another way of thinking about this is to recognize that the stack loop on the 7600 is an exact analogue for computational physics to microcoded instructions on the minicomputer. From this point of view, one might want to think about actually letting the hardware treat the stack loop as a microcode. That is, some instructions could be used as pointers to blocks of "microcode," or stack loops, such pointer-instructions and "microcode" having been previously defined by the system, or user, or both.

The remainder of the goal of such an approach is fast scalar processing power. Examples given above have shown the havoc which a high vector to scalar speed ratio can create. It has been known for some time that a machine with general purpose CPU registers, an instruction stack, specialized functional units, simple CPU to memory interface, and memory and CPU words of the same size can give high serial computational speed. What has not been clear is that, in such a machine, the vector processing rate could be very high and the kinds of vectors readily manageable could be very general. If one does those things that make this structure faster for vectors, and also pays attention to keeping the scalar rate high, then one is tending toward a balanced machine that is fast for all problems.

The final generalization of this point of view is to note that we are speaking against the highly structured machines in favor of unstructured machines. By "structured machine" we mean a machine that demands limiting kinds of structure in its problems if it is to be efficient. Despite its overly structured I/O scheme and memory hierarchy, the 7600 is very unstructured at the CPU level. Instructions are very basic and are almost "microcode." Functional units and registers may be interconnected and assigned in a nearly random way. We are arguing that this lack of structure should be extended still further with the CPU and into the memory. We are also arguing that certain tagged architecture aids, such as vector descriptors, and code and "microcode" descriptors, would mate very well with this kind of unstructured approach.

Matching Problems With Processors

Our experience has made it clear that all of a code must be considered in the evaluation of the suitability of running a given problem on a processor of a given architectural class. The reason for this is the extreme speed ratio which can occur between pure vector or parallel code and pure scalar code. Additional special constraints, such as slow matrix transpose operations, imposed by particular processor realizations, may make certain vector operations relatively slow.

Consider two process rates, r_1 and r_2 , and assume that $r_1 > r_2$. Assume that the fraction of the total process that runs at the slow rate r_2 is f . Then the ratio of the speed of the process with the

two rates to the speed which the process would have, if the faster rate held entirely, is

$$M = \frac{r_2}{(1-f)r_2 + fr_1}$$

If r_1 is associated with the average vector speed of a problem and r_2 with the average scalar speed, then the effects of slow scalar operations can be computed. For example, if r_2 is one tenth of r_1 and f is 10%, then $M \approx 0.5$. That is, 10% of scalar code running at 10% of r_1 will retard a processor to one half its vector speed. The same effect is seen for 20% scalar code that is 5 times slower than vector code, or for 1% scalar code that is 100 times slower than vector code.

Areas in which this phenomenon is important which have been noted so far at LLL are:

- a) Transfer, without rework, of pre-processing and post-processing packages (graphics, zone set ups, data base managers, intercode linkers, etc.) if these packages are used extensively.
- b) Failure to almost entirely eliminate scalar code from a FORTRAN program rewritten with vector instructions.
- c) Monte Carlo coding in which the number of logical categories (number of vectors) become so large that the sizes of categories had to be reduced to the point where vector operation start up time was significant.
- d) No good solution to a full implicit difference scheme has been discovered for the vector processor. The alternating direction implicit scheme can be done, if the grid dimensions are large so that vector start up times are not too significant. This is because the back substitution can then be done one row at a time. The parallel machine does well in this case if the number of its processing elements is exactly a divisor of the grid dimension.
- e) Transferring a code with substantial mapping requirements, such as a multi-material elastic-plastic flow problem, onto a machine with poor bit control vector capability.

Conclusions

1. The parallel and vector machines are very special purpose architectures. None of the vector or parallel machines can be generally relied upon to do well on implicit differencing schemes of Monte Carlo approaches, which approaches, in our opinion, may tend to overshadow explicit schemes in the future for the kind of substantial computational problem discussed here.
2. Extreme speed differences between pure vector or parallel coding and pure scalar coding on the special purpose machines may easily pull down overall performance on a problem containing substantial scalar code. Note that much less than 50% scalar code can reduce

performance to well below that of a good concurrent processor.

3. Vectors must be several hundred elements long to keep start up time penalties below 10% of total operation time on vector machines today. The parallel approach has a similar, less easily stated problem. This problem is not resolved by confining vector lengths to multiples of the number of processing elements if substantial numbers of set up operations are required.
4. The speed of the 7600 can be increased by the use of instack-loops. That is, vector instructions can be emulated on the 7600 by taking advantage of its CPU structure. The concurrent processor coded this way seems to represent a very balanced architecture.

The vector concept is generally a good one primarily in the sense of stack-loops on the concurrent processor, where new vector instructions, possibly of a complex nature, can be readily emulated.

5. From the users point of view one major difference between the parallel and vector architectures relates start up time penalties on the vector processor versus discontinuous parallel processor utilization, depending on the number of active processing elements. Another difference is that the parallel machine intrinsically has a high vector to scalar speed ratio, which is compensated (to an unknown degree) by the faster scalar fetch times compared to the vector processor.
6. One cannot seriously consider putting an unmodified FORTRAN problem on any of the vector or parallel machines. Even if the problem is "merely" recoded for FORTRAN with vector extensions, it must be completely re-thought and re-done, if it is to use the available computing power. The principle advantage of vectorized FORTRAN, as opposed to an entirely different language, is that certain peripheral coding may be left intact.

7. Primary storage to/from secondary storage transfer rates are crucial to effective CPU utilization. If a code overflows primary storage, it must be possible, in a straightforward manner, to maintain bidirectional transfer rates that are 1/20th to 1/80th the CPU floating point result rate. If the disc doesn't measure up, then one must buy more primary storage, or block storage, such as large core memory (LCM) on the 7600.

References

1. M. J. A. J., "Some Computer Organizations and Their Effectiveness," IEEE Transactions on Computers C-21, 948 (1972).
2. F. H. McMahon, "FORTRAN Benchmark Results Using Several Optimizing Compilers on CDC 7600," Unpublished Memorandum (Nov. 24, 1972).
3. F. H. McMahon and L. J. Sloan, "STACKLIB- A Vector Function Library of Optimum Stack-Loops for the CDC-7600," (To be published.)
4. L. J. Sloan, An Introduction To The Triad Subroutines, R and D Associates, Santa Monica, Calif. #RDA-TR-035-DNA (1971).
5. J. L. Owens, "The Influence of Machine Organization on Algorithms," Symposium on The Complexity of Sequential and Parallel Numerical Algorithms, May 16-18, 1973, Carnegie-Mellon University, Computer Science Department, Pittsburgh, Pa.
6. G. H. Barnes, et al., "The ILLIAC IV Computer," IEEE Transactions On Computing C-17, 746 (1968).
7. E. A. Feustel, "On The Advantages of Tagged Architecture," IEEE Transactions on Computers C-22 (1973).