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-74795
PREPRINT

Conf-730579--1

24, 992

LAWRENCE LIVERMORE LABORATORY
University of California/Livermore, California

THE INFLUENCE OF MACHINE ORGANIZATION ON ALGORITHMS

Jerry L. Owens

May 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.

MASTER

Symposium on Complexity of Sequential
and Parallel Numerical Algorithms May 16-18, 1973

INFORMATION OF THIS DOCUMENT IS UNLIMITED

174

THE INFLUENCE OF MACHINE ORGANIZATION ON ALGORITHMS*

Jerry L. Owens
University of California
Lawrence Livermore Laboratory
Livermore, California

INTRODUCTION

There are a great many algorithms for accomplishing any particular computational task. The goal of programming should be to choose the algorithm that is best in some sense. When making this choice, it is often necessary to take into consideration the nature of the machine upon which the algorithm is to be executed.

There is now appearing upon the market a class of machines that have an organization which forces upon the programmer a new set of considerations that have an influence upon his decision as to which algorithms are best. Some examples of this new class of machines are 1) The CDC STAR-100; 2) The ILLIAC IV and 3) The Texas Instruments ASC. Some of the new considerations that the programmer of these machines must take into consideration are 1) How to arrange the data for his algorithm so the machine can process that data efficiently; 2) are there ways of performing this operation upon more data, and 3) are there algorithms built into the machine upon which this algorithm will run that will accomplish this or that small piece of this algorithm in a more efficient manner. None of these considerations are entirely new to today's experienced programmer. What is new is the importance that these considerations have upon the execution rate of algorithms on these new machines.

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

This paper will give some of the observations of a group of programmers with two assigned tasks. First, we were asked to do a comparative analysis of the CDC STAR-100 and the ILLIAC IV for a given code. Second, we were asked to program a large two-dimensional lagrangian hydrodynamic model for the STAR-100 computer. This paper will consist of three parts. First, it will present some thoretical considerations concerning execution rates. Next, it will give some examples of where these theoretical considerations were important considerations in the design of our algorithms. Finally, it will be concluded by several observations and opinions, many of which may be controversial but which should be considered by the programming world at large.

THEORETICAL CONSIDERATIONS

In this section, we will talk only about machines like the CDC STAR-100 which perform operations upon vectors. For our purposes, we will define a vector to be a set of data stored in-to contiguous storage locations. In section 3 of this paper, we will show how the results of this section may be applied to a larger class of machines than the CDC STAR or the Texas Instruments ASC.

One of the first things that a person learns when he approaches a vector (or pipeline) machine is that the time that it takes to execute a vector instruction is composed of two times. First, there is the start-up time. This is the time it takes for the machine to decide what to do and to set up the data flow necessary to do it. This time is usually considered to be at an end when the first result of the operation becomes available. This time is independent of the length of the vectors involved. Second, there is the operation time. This is the time the instruction takes between the first result and the last result. This time depends upon the length of the vectors that take part in the operation. In the vector machines that are being considered, the start-up time is usually much longer than the time to generate a result once the data is flowing. Typically, the start-up time is equivalent to the time it takes to generate 30 to 60 results once the data is flowing. Since this is the case, it is clear that one thing that the programmer of these machines should be concerned about is the length of his vector operations. In order to get a better feel for how long our vectors should be, let us consider two ways of doing the same operation on a set of data. First, let's say that we have N items of data to operate upon and that we obtain our N results with one vector operation. Then the time it takes to complete the operation is $S + N \cdot OP$ where S is the start-up time and OP is the time that it takes

to deliver each result once the data flow has been set up. Next, let's consider obtaining the same results with K vector operations, each of which are L long (i.e. $N = K \cdot L$). The time to get the results this way is $K \cdot (S + L \cdot OP) = K \cdot S + N \cdot OP$. Now if we wish to know how much time was spent doing extra work in the second method, we can subtract the two times. If we do this, we get:

$$\Delta T = (K \cdot S + N \cdot OP) - (S + N \cdot OP) = (K - 1) \cdot S.$$

In other words the extra time was simply the number of additional start-ups times the time it takes for one start-up. One interesting way to view this number is to divide it by the time that it takes to do the operation the second way. If we do this, we get

$$\text{Ratio} = (K - 1) \cdot S / (K \cdot S + K \cdot L \cdot OP)$$

$$\text{Ratio} = \frac{(K - 1)}{K} \cdot (S/OP) / ((S/OP) + L)$$

If we now multiply this by 100 and let $R = S/OP$, we get

$$(1) \quad \text{Percent of time lost} = 100 \cdot \frac{(K - 1)}{K} \cdot R / (R + L)$$

The value of R depends upon the particular machine instruction that we are executing. We will take a value for R of 40. Now with R fixed, let's let $N = 100$ and $L = 1$. Then $K = 100$ since $N = K \cdot L$. Then we are saying by this choice of N and L is that we wish to find the percent of time lost in start-up, if we compare doing an operation of length one with doing a vector operation of length 100. The result of using these values is

$$\begin{aligned} \text{Percent of time lost} &= 100 \cdot \frac{99}{100} \cdot \frac{40}{40 + 1} \\ &= 99 \cdot \frac{40}{41} \approx 97 \text{ percent} \end{aligned}$$

This means that to get the same results with vectors of length one as with a vector of length one hundred we spend 97 percent of the time in vector start-up. Clearly, we would prefer to work with vectors of length 100 rather than vectors of length one.

But there is nothing magical about vectors of length 100. Now much better could we do with vectors of say 10,000 as compared to vectors of length 100. In this case, $N = 10,000$ and $L = 100$. Therefore, $K = 100$.

Putting these numbers into equation 1.1 with N still equal to 40, we get

$$\begin{aligned}\text{Percent of time lost} &= 100 \cdot \frac{99}{100} \cdot \frac{40}{40 + 100} \\ &= 99 \cdot \frac{40}{140} = 28 \text{ percent}\end{aligned}$$

Clearly in moving from vectors of length 100 to vectors of length 10,000, we have gained more than one fourth of our machine back but we must remember that if we use vectors of length 10,000, then we must have 10,000 memory locations for each temporary result in our program instead of 100 memory locations.

Let's consider one more case. Let's let $N = 100,000$ and $L = 1,000$. Then $K = 100$ again, and we get

$$\begin{aligned}\text{Percent of time lost} &= 100 \cdot \frac{99}{100} \cdot \frac{40}{40 + 1000} \\ &= 99 \cdot \frac{40}{1040} = 4 \text{ percent.}\end{aligned}$$

Now clearly we are not advocating using vectors of length 100,000. For one thing the CDC-STAR only allows vectors of lengths 64,000 maximum and for another, on a machine with one-half a million words this would allow only 5 vectors to be in core at any one time. But the calculation above does indicate that we do not lose very much of our machine to start-up time with vectors of length 1000. In fact, our group decided to try to use vectors of length of about 1000 whenever possible. See Figure 1 for a graphical presentation of the percent of time lost in start-up as a function of the length of the vector operation.

From the above analysis, it is clear that we would like to do operations that are as long as practical. Now let's suppose that we have 100 numbers that we wish to add to another 100 numbers but that these numbers are not organized into vectors. Is it worth the effort to collect these numbers into a vector so that we can do our vector operation? In order to decide this, let's assume that our 100 numbers are scattered through 10,000 memory locations.

First, let's calculate the time it would take to do our one hundred operation each on a vector length of one.

$$(1.2) \quad \text{TIME} = 100 \cdot (ST + 1 \cdot CT)$$

We also must add to this time the time to build the descriptions of each of our 100 vectors. Let's say that this takes $50 \cdot ST$. Then

Above is a graph that shows the percent of time used by vector start-up as a function of the length of the vectors being operated upon.

FIGURE 1

$$(1.3) \quad \text{TIME} = 150 \cdot \text{ST} + 100 \cdot \text{OP}.$$

On the CDC-STAR there are several ways to collect our 100 numbers into a vector. We will consider two of these ways. First, let's consider compressing out of the 10,000 locations the 100 numbers that are needed.

We will assume that we have a 10,000 bit long bit string with ones in the positions corresponding to the data items that we need and zeros elsewhere. If we do not have such a bit string, then we must generate it and that will mean added cost. First, let's calculate the time to collect one of the addends.

$$\text{TIME} = \text{ST} + 10,000 \cdot \text{OP}$$

Since we are assuming that $\text{ST} = 40 \cdot \text{OP}$ for our discussion, the time to do all the operations for equation 1.3 is

$$(1.4) \quad \text{TIME} = 150 \cdot 40 \cdot \text{OP} + 100 \cdot \text{OP} = 6100 \cdot \text{OP}.$$

While the cost to collect only one addend by compression is

$$\text{TIME} = 40 \cdot \text{OP} + 10,000 \cdot \text{OP} = 10,040 \cdot \text{OP}.$$

Therefore, you certainly do not wish to use compression to collect your data. It is in fact true in general that you should not choose compression if the data you want is less than about 5% dense in your data base. Note that this method is strongly dependent on the size of the data base. If the 100 locations had been scattered among 1000 locations, then the compression method would still be worth considering.

The next method of collecting this data that we will investigate is the vector indexing method that we will call indexing by a vector. In this method, you supply the machine a vector of indexes and a starting location. The machine calculates the address of the data by adding each index to the starting location. It then fetches the data from that location and stores the data fetched into contiguous core locations to form the resultant vector. Note this type of instruction is not itself efficient on a vector machine since it is doing the equivalent of a random fetch from core, however some speed-up can be gained because the output is a vector. To do this operation the OP time is on the order of 1/6 of the start-up time rather than 1/40 as we have previously been assuming. We will assume that you already have a vector of indexes. If this is not the case, then this vector will have to be built at added expense in time. Note that this method is not dependent on the size of the data base from which we are selecting.

The time to collect one addend by this method is

$$\text{TIME} = 5T + 100 \cdot (10 \cdot \text{OP}),$$

Here we use $10 \cdot \text{OP}$ because this instruction's rate of execution is about that much slower than the other instructions. Using this we find that the time to collect one addend is

$$\begin{aligned}\text{TIME} &= 40 \cdot \text{OP} + 1000 \cdot \text{OP} \\ &= 1040 \cdot \text{OP}.\end{aligned}$$

Since we must collect two addends and then store away the final result after the desired operation is performed, we conclude that the collection and distribution costs are

$$\text{TIME} = 1120 \cdot \text{OP}$$

To this we must add the time to do the operation itself which is

$$\text{TIME for OP} = 5T + 100 \cdot \text{OP} = 140 \cdot \text{OP}.$$

Adding these two together we get

$$\text{Total TIME} = 1260 \cdot \text{OP}.$$

Since equation 1.4 indicates that the original method requires a time of $6100 \cdot \text{OP}$, we conclude that in this case it is worth while to collect the data into a vector by using vector indexing. See Figure 2 for a graphical presentation that compares four ways of processing data that is not in contiguous memory locations. When we consider that we usually are going to collect data that will take part in many operations, it becomes even more important to consider collecting the data into vectors even if that appears like a great deal of effort.

The above analysis has raised another question. That question is "What method is the best to use to select a subset of data from a larger set?" In order to answer this question, we will calculate the time to do a selection by compression and also by indexing a vector. We will let T_I be the time to select by vector indexing, and T_C be the time to select by vector compression. Let N be the size of the data set that we must select over and let n be the number of items that need to be selected. Also, we will still assume the $5T = 40 \cdot \text{OP}$ and that the operation time for a vector index operation is about 10 times the normal operation time. Then we find that

$$T_I = 5T + n \cdot 10 \cdot \text{OP} = (40 + n \cdot 10) \cdot \text{OP}$$

and

$$T_C = 5T + n \cdot \text{OP} = (40 + n) \cdot \text{OP}.$$

Above is a graph that shows the time used to do a single vector operation as a function of the density of the data selected. Note this curve is for one operation only Figure 3 shows a more realistic case.

FIGURE 2

We might conclude from this that we will compress if $N < 10 \cdot M$ and will select by vector indexing if $N > 10 \cdot M$, but in practice, things are more complicated. First, we must consider that the rate of execution of the vector indexing instruction may be greater than that indicated if there are many memory conflicts caused by the distribution of the indexes. There are other considerations that affect our choice which will be described below.

The above considerations all lead to the conclusion that on a vector machine, you wish to have your vectors as long as possible. This conclusion is true in general only if all the results of the vector operation are in fact needed by the calculation.

There are circumstances in which one might wish to do vector operations on vectors that are shorter than the vectors with which the program is currently dealing. Consider the following example.

Assume that we have a vector V whose length is 1000. Also, let's suppose that we would like to modify certain elements of the vector with an algorithm which performs 20 vector operations. Now let's suppose that only 10 elements of the vector V need to be processed by the algorithm. We will determine the time required for this computation using two different procedures. First, let's assume that we perform this algorithm by using control bit vectors. This means that the algorithm will be performed on the entire vector V , but we will carry for each result a bit which indicates whether or not the result is to be stored.

We calculate our execution time to be

$$TIME = 20 \cdot (5T + 1000 \cdot OP)$$

or

$$TIME = 20 \cdot 5T + 20,000 \cdot OP$$

$$= 20 \cdot (40 OP) + 20,000 \cdot OP = 20,800 \cdot OP.$$

As a second method of doing this calculation, let us consider selecting out the 10 numbers upon which we wish to perform the algorithm. To do this, the time to do the selection must be added to the calculation time in order to get a total calculation time. Further, let's assume we do our selection with a compress even though we know that for only ten numbers out of a thousand it would be faster to do an indexed vector operation. We choose the compress instruction here because we have already assumed the needed bit string's existence in the first method of solving this problem. Using the compress we get

$$TIME \text{ for compress} = 5T + 1000 \cdot OP$$

and

$$\text{TIME for algorithm} = 20 (ST + 10 \cdot OP)$$

Taking into consideration that the output data must be put back into it's correct place we calculate

$$\begin{aligned}\text{TIME} &= 20 (ST + 10 \cdot OP) + 2 \cdot (ST + 1000 \cdot OP) \\ &= 20 (40 \cdot OP) + 200 \cdot OP + 2 \cdot 40 \cdot OP + 2000 \cdot OP \\ &= 800 \cdot OP + 200 \cdot OP + 80 \cdot OP + 2000 \cdot OP \\ &= 3080 \cdot OP\end{aligned}$$

This is considerably less than the $20,800 \cdot OP$ that we calculated for the control method.

In actual practice in a case like this you usually want to select out of several vectors and you do a bit compress on a set of integers to get an index set for your selections. You then select out of each of your vectors with that index set. If we use this index method on the above problem, we get

$$\begin{aligned}\text{TIME to generate index set} &= ST + 1000 \cdot OP \\ \text{TIME to select using index set} &= ST + 10 \cdot (10 \cdot OP) \\ \text{TIME to do algorithm} &= 20 \cdot ST + 20 \cdot 10 \cdot OP \\ \text{TIME to put data back in place} &= ST + 10 \cdot (10 \cdot OP) \\ \text{Total time} &= 23 \cdot ST + 1400 \cdot OP \\ &= 23 \cdot 40 \cdot OP + 1400 \cdot OP \\ &= (920 + 1400) \cdot OP \\ &= 2320 \cdot OP\end{aligned}$$

Note that this is even less than the $3080 \cdot OP$ that we obtained by the compression method. When we are performing the algorithm on vectors that are only 10 long we are spending almost all of our time in vector start-up. We are willing to do this because we can minimize the total calculational time. If the data that we wish to operate on represented about half of the original vector length, then the bit compression method would have taken less time because the index operation goes about 10 times slower per result.

It should be clear that whenever we arrive at a place in a program where certain parts of the data must go through one set of equations and the rest of the data must go through a different set of equations, we can calculate the time it will take to do each of the sets of equations and how long it will take to select out the correct data for each equation. Therefore, we can decide whether or not it is more efficient to select out the data needed for a particular equation and the

best method for selecting this data.

To illustrate this, let us consider a vector of data V . Further, let us suppose that LV is the length of the vector V and that $LV1$ elements of V are to be processed by an equation that requires $KOP1$ operations. Also suppose that $LV2$ elements of V are to be processed by an equation that requires $KOP2$ operations. For simplicity, let $LV = LV1 + LV2$. Under these circumstances, if we run the vector V through both equations using bit controlled operations, we find that

$$(1.5) \text{ TIME with no selection} = 2 \cdot (KOP1 + KOP2) \cdot ST + (LV1 + LV2) \cdot OP.$$

If we decide to do select with compression operations, then the time to select is

$$\text{Compression select TIME} = ST + (LV1 + LV2) \cdot CP.$$

Since we must also put the data back at the end and select for both equations, we have

$$\text{Total compression select TIME} = 4 \cdot (ST + (LV1 + LV2) \cdot OP).$$

Now on each side of our IF test we use only the data needed by the equation on that side. Therefore,

$$(1.6) \quad \text{TIME on Side 1} = KOP1 \cdot (ST + LV1 \cdot CP)$$

and

$$(1.7) \quad \text{TIME on Side 2} = KOP2 \cdot (ST + LV2 \cdot CP).$$

So our total time to do the calculation by this method is

$$(1.8) \quad \begin{aligned} \text{Total TIME with compression} = & (4 + KOP1 + KOP2) \\ & \cdot ST + (KOP1 + 4) \cdot LV1 + (KOP2 + 4) \cdot LV2 \\ & \cdot CP. \end{aligned}$$

If we decide to select with an index vector then the time to generate the index set (assuming we have a bit string) is

$$\text{Index generation TIME} = ST + (LV1 + LV2) \cdot OP.$$

and the time to do the selection on each side is

$$\text{Side 1 selection} = ST + LV1 \cdot (10 \cdot OP)$$

$$\text{Side 2 selection} = ST + LV2 \cdot (10 \cdot OP).$$

Since we must put the data back, we must multiply each of these by two. So we get as total selection time for vector indexing.

$$\text{Total indexed selection TIME} = 5 \cdot ST + (21 \cdot LV1 + 21 \cdot LV2) \cdot OP$$

Note that in this case the vector index method is slower than the compress method. This is because $LV1 + LV2 = LV$ so that either $LV1$ or $LV2$ or both are fairly dense in V and we know that vector indexing on a fairly dense vector is slower than vector compression. Figure 3 gives a graphical presentation that shows what the execution time is for one side of an IF test if each of the methods above are used as a function of the density of the selected data.

Since we are operating with the same vectors as before, we see that equation (1.6) and (1.7) hold for this case also. Therefore, we conclude that

$$(1.9) \quad \text{Total TIME with vector indexing} = (5 + NOP1 + NOP2) \cdot ST + ((21 + NOP1) \cdot LV1 + (21 + NOP2) \cdot LV2) \cdot OP$$

It should be noted that in the example above we could have used the vector merge instruction to put our vectors back together and that may have reduced the final timings. We did not do this because we wished the discussion to be as clear as possible. Introducing the merge instruction would only have complicated matters.

One thing to note about the above analysis is that the length of the various vectors going down each side of an IF test is not known until the code has the particular data vectors and bit strings with which it is to run. This means that this analysis can be done only at run time. To use dynamic selection of methods requires the coding of the various selection methods into the program, but in general this does not require multiple coding of the equations themselves.

The conclusion to be drawn from this section is that vectors should be as long as possible to reduce the percentage of time spent in vector start-up compared to the total vector execution time, but we do not wish to operate on a large number of data items that do not yield useful results.

The above is a graph that shows the time used to do 20 vector operations when the data is selected from three vectors as a function of the density of the data selected.

FIGURE 3

THE INFLUENCE OF VECTORS ON THE DESIGN OF ALGORITHMS

There are times when programming for a vector machine causes tremendous changes in the way you do things. As an example, consider the following program written in FORTRAN.

```
DIMENSION      A (100)

DO 5 I = 1,100

  IF (A(I) .GT. 10)  A(I) = A(I) / 10

5 CONTINUE
```

This algorithm might be programmed for a vector machine as follows:

```
VECTOR  A(99), B(99)

BIT B

B = A .GT. 10

A = B .CTEL. A/10
```

Note that in this second algorithm we pass over the entire vector A twice. The first time we pass over the vector A a bit string B is set to one in those places where A is greater than ten. The second time we pass over A each element of A is divided by ten. But the result of that division is stored only when the bit vector B has a one in that position.

It seems that no one in his right mind would program this algorithm in the second way on a present day serial machine but analysis shows that this second approach is the best way (in the sense of least execution time) to do this process on a vector machine. At first glance, it appears that the second algorithm is wasteful since you must pass over the entire vector A twice. But let us look a little closer at the first algorithm. First, we note that there is a variable I in that algorithm that must be incremented each time through the loop. Second, notice that there is a loop and not only that, but there is a test within the loop and the machine must wait for the results of this test before instruction processing can continue! It seems to us that the first algorithm has some very bad things in it, when all we wished to do was divide those elements of A that are greater than ten by ten. From this point of view, the second algorithm simply says first identify those elements of A that are greater than ten. Next, divide each of the elements identified by ten.

From a programmer's point of view, it is very nice to see such a clean algorithm, and the fact that the second algorithm will execute faster than the first confirms our suspicion that clean algorithms can be the best. The fact that almost all programmers, if given the English version of this algorithm, would come up with the first algorithm may indicate that our thought patterns have been greatly influenced by the organization of the computers and languages we have had to use in the past. The second algorithm could, in fact, be presented to a large bank of computers and all the compares done in parallel. Then all the divides could be done in parallel. We have noticed that when an algorithm is phrased in vector language, it is also set up so the many computers could process the instruction all at once. This is of course the ILLIAC IV concept. (Many computers all doing the same operation.) As we shall see in section 3 the ILLIAC IV makes a great "vector" machine. What we are in fact dealing with when we talk about a vector machine is a general parallel processor in which only the primitive operations proceed in parallel. This type of parallelism is very easy to control. We shall make an argument later that this may be the only type of parallelism needed if we wish to make optimum use of the computers resources.

Before we go into some of the questions raised above, let us give a few more examples for which vector processing changes the nature but not the results of our algorithms.

In the first part of the last section, we learned that we should keep our vectors long and that it is wasteful of machine time if we must collect our vectors out of a larger data set. These facts lead one to conclude that we should design programs so that the data is laid out in vectors. The fact that we are putting our data into vectors may mean great changes in our algorithms. For example, if we must access the columns of a matrix we may decide that it is best to transpose that matrix so that we can access them with vectors. This rearranging of the data may be such a traumatic change on an already existing program that it is best to start over and reprogram rather than try to change the data organization. As another example of how a change in the data organization may destroy the currently programmed code, consider the case where several different variables are being kept in the rows of a matrix so that the equations now say something like $A(J,1) = A(J,3) - A(J,5)$. When this program is vectorized it may be best to let each column of the original matrix be a vector with its own name. The above equation might then read $HASS = VOLIPE - DENSITY$. Notice also that in the original coding J may have been set and many pages of coding performed before J was incremented and the branch was made back to the beginning. We think, from

these examples, it can be perceived that, although the equations behind a code may not change, the details of the coding can be radically changed by attempting to restructure the data into vectors. Yet, for a vector machine you must restructure the data into vectors or you will pay a very high price to collect the data into vectors and then redistribute it back into its proper place during execution.

We also learned in the last section that sometimes it is best to pay the price of collecting the data into vectors rather than operate with short vectors. This principle can be applied in those cases where it is not possible to organize the data into vectors. For example, consider the case of a program that is modeling some physical process by finite size zones. It may happen that a zone can enter a state different from its neighbors and that a different set of equations will have to be applied to this zone. In this case, it may be best to collect all the zones that are in the same state and process them through the equations together and then put them back where they came from. Since the state of a zone can change with time, it is clear that this collection must be done during execution. The process described will probably differ from what is currently being done in a code of this kind. In such a code, it has been traditional to process zone by zone, branching to the equations that are to be applied to each zone. There is no reason why the current codes could not have been programmed in the vector manner described above, but since there appears to be much wasted effort in moving and selecting data, most of the current codes are not structured in this way. It is interesting that the "wasted effort" described is necessary if vector machines are to fulfill their promised increase in speed.

Another major change in programs that is necessary on a vector type machine is that for the first time, in many of the codes, the allocation of space for temporary results becomes a major problem. In a serial process, the temporary results tend to need one word of memory, but in a vector process each temporary result may be 1000 words long and this leads to a memory management problem that these programs have never had to face until now. This problem of memory management is probably the most important reason that the current programs are not coded in the vector manner. Why would anyone cut out a thousand words of memory to hold the results of an addition if he could take it with one word. The answer for the programmer on a vector machine is that if he does this then the code will execute many times faster. This may be just another form of that well known phenomena: the "space vs time trade-off".

In this section we have attempted to show how and why moving a code from a serial to a vector machine may require

radical changes to the structure of that code. There are codes that can make this change without major reprogramming but these codes are already vector codes in serial clothing. Considerable work will be required on most codes to move them to the vector machines in a form so that they will gain the speed increase promised by the vector machines. It is interesting to note that we have observed an extra dividend for this effort. We feel that casting a code into vector form yields a better code because most of the changes made in the algorithms necessary for vectorization also lead to better algorithms.

SOME OBSERVATIONS AND COMMENTS

In this section we will make some general comments on programming for vector machines and the direction of computer evolution in general. We will start with a short history of our effort in machines of this type.

The Lawrence Livermore Laboratory has had a STAR-100 computer on order for several years now. Our group has converted a large two-dimensional hydrodynamic Lagrangian program into a form that will run optically on the CDC STAR-100. During the process of converting this program into a form that would run well on the STAR we have learned a great deal. One of the things that we have learned is that many modern computers are constructed in a way that makes them run very well as vector machines. In 1969 we did an evaluation of the ILLIAC IV in which we concluded that the best way (in the sense of utilizing the hardware optically) to program the ILLIAC IV would be to write a set of assembly language subroutines to do vector operations and to call upon these to do the calculations. It appeared to us that with this approach we could pack all our data into consecutive memory locations on the ILLIAC IV and still run faster than if we used the traditional skewed ways of putting our data into the ILLIAC IV's memory. This procedure makes the maximum utilization of memory on this machine and memory is a very scarce resource on this machine. This approach will work because in the FE's you can have several operations going on simultaneously. The fact that many machines run well with assembly language simulation of vector instructions is very well illustrated by the fact that the same vector code mentioned above runs on the CDC-7600. This version runs about twice as fast on the 7600 as the current serial version of this code which is coded in FORTRAN. See Figure 4 for a graphical presentation of the results of a timing analysis comparing this program in FORTRAN with the vector version. The main reason why this works so well is that the 7600 computer was designed so the various functions that the machine must perform have been separated from

Above is a graph that shows a comparison of the number of points processed by the serial FORTRAN program and the vector program as a function of the length of the vectors used. Note that these are experimental curves derived by measuring the execution time of each of these programs on the CDC 7600.

FIGURE 4

one another in a manner that allows them to run in parallel with each other.

When we were designing this the vector version of this program there were several things that worried us about the vector approach. First the amount of time we spent moving data from here to there just so we could use vector operations looked very large. Now that we have a running code timings show that only about 15% of our execution time is spent in doing this "necessary wasted effort". This may sound large but we thought that it would be much larger. It is interesting to note that about this same percentage of our execution time is being spent in the serial code in fetching, storing and modifying indexes. So in some sense the serial code has about the same data manipulation overhead but it is just in a different and more hidden form. Another thing that we were worried about was how much time we would spend in vector start-up. As we said before we decided to try to run with vectors of length of about 1000 so that our start-up time would be about two to three percent. But as we have noted it is not always possible to keep your vectors that long. It now looks like we are spending about 5% of our time in vector start-up. Because we were worried about the time that we would spend in vector start-up we did our memory allocation statically so that we would not spend any extra time allocating space for each vector result. We now think that it would be better to use a simple dynamic temporary allocation scheme so that the programmer would not have to keep track of which temporary vectors are in use and which are available for use.

It appears to us that the STAR may be the first of a new generation of computers that have algorithms built into the hardware. These algorithms will work optimally for the machine hardware that is available. STAR tries to take from the programmer the job of making the hardware run at the highest rate possible and to place this responsibility upon the hardware designer. The difficulty that CDC is having building the first STAR may be an indication of how difficult the job of optimizing algorithms is. The programmer for the STAR becomes a "data manager" and "algorithm chooser". It may be that in the future more machines will be designed with instruction sets that are really high level languages. It is interesting to note that we have been able to convert the CDC-7600 into a vector machine by programming assembly language subroutines and it may be that in the future higher level machine instructions may be implemented by micro-code.

As we have noted before, if a program is cast in vector form then all the operations could be processed in parallel. It is not necessary to have several computers running in parallel doing the operation on the STAR because the pipeline on

the STAR is fast enough to keep up with the memory bandwidth. In fact, when doing a vector instruction on the STAR-100 three-fourths of the memory bandwidth of the machine is being used by the pipeline. In this circumstance having another processor running out of the STAR's memory would be a waste of computing power because the other processor could not access memory without interfering with the flow of data through the pipeline. It seems clear that any computer that is generating results at the memory bandwidth rate is using up the most important resource available to it. If all of these results are useful to the user, then he should feel confident that he is getting the most possible out of that machine.

One of the first problems that a programmer has when he decides to program for a vector machine is how to design and test his algorithms. We found that the interactive language called APL was most helpful when we were designing our algorithms for the STAR. We could program our algorithms and debug them rapidly using the interpreter. We also found APL helpful in overcoming our old programming habits. One of the most difficult problems with the new vector machines is going to be to get people to change their habits and make use of the machine instructions that they have been given, even if this means performing some of that "necessary wasted effort". These new machines may die simply because people are too lazy to use them properly.

CONCLUSION

By 1980 we should know what the judgement of the computer world was upon the vector machines. We may never know what the judgement should have been.

ACKNOWLEDGEMENT

I wish to express my thanks to Val Kransky, Carl Jensen, Dick Giroux, Gary Long and F. McMahon for making suggestions that improved this paper. I wish also to thank Dorothy Dean and Kay Ulibarri for their rapid typing of this paper.