Adapting "Single Algorithm Multiple Data" Programs for Multiprocessing Vector Computers

R. W. Doran

Auckland Computer Science Report No. 45

July 1990

Adapting
"Single Algorithm Multiple Data" Programs
for
Multiprocessing Vector Computers

R. W. Doran

Abstract

This paper explores the way that simple but inherently parallel programs may be adapted to a parallel vector computer. Starting with an example algorithm that is applied independently to many sets of data, we proceed to convert it systematically into a highly parallel form by using multiprocessing and vector processing. The various versions of the program are written in Fortran for the Cray X-MP. Performance data are collected to show the extent of the performance improvement that can be expected.

1. SAMD programs

There is a common type of computer program where the same algorithm is to be applied to all members of a set of data and where the operations on each data item are quite independent of the operations on others. Examples are many graphics programs where the same operation is applied to each pixel and in monte-carlo simulation where each trial is independent. There seems to be no good word yet coined to describe such a program or program section - the epithet "event processing style" is applied in experimental physics as it is the basic nature of the programs in this area. Lets invent a term, say *SAMD* for “Single Algorithm Multiple Data”, using Flynn's enduring terminology, where we imply that the course of execution of the program on one data item is completely independent of the execution course of the program on other data and that the course of execution may well be different for different data.

What we will do here is describe in a tutorial fashion the techniques involved in adapting a SAMD program to work on a vector processing computer that has multiple processors. Although there have been many examples given of adapted programs (see for example Iliffe[1]) the process by which they are derived has not been well described. We will illustrate the techniques by applying them to a single example program on a specific computer. The computer that we shall use is the Cray X-MP. This has four CPUs sharing a common memory. Each CPU is a highly parallel vector processor that operates on vectors of length up to 64. All the example programs are written in Fortran CFT77.

2. The serial program

The example we will use is the heapsort applied to many arrays of different lengths. The arrays are presented to the program packed contiguously into a large array with the sections to be sorted identified by a pair of other arrays that specify bases and lengths, as follows:

```
      SUBROUTINE SORTSET(M,N,BASE,X)
         REAL X(*)
         INTEGER M,          !number of arrays to be sorted
                 N(1:M),     !length of each array
                 BASE(1:M)   !base of each array
C        Sort the M arrays: X(BASE(W)+1:BASE(W)+N(W)), W = 1..M
             INTEGER W
             DO 100 W = 1, M
  100            CALL SORT(N(W),X(BASE(W)+1))
         END
```

The heapsort algorithm is essentially serial so may not be the first choice for use on a parallel computer. However, it is a nice example for this exercise as it has a lot of structure yet is very compact. The sort is done in two phases, the first places the array in "tree order", i.e. such that `X(I)` is greater than both `X(2*I)` and `X(2*I+1)`. The second phase orders the array, making use of the fact that the top of the tree is the largest element in the array:

```
      SUBROUTINE SORT(N,X)
         INTEGER   N
         REAL      X(1:N)
C        Bring X(1:N) into ascending order using heap sort
             INTEGER   K, L
             REAL      XT
C            Ensure that N is > 1
                 IF( N .LT. 2 ) RETURN     ! Exit
C            Place X into descending tree order
                 DO 30 L = N/2, 1, -1
C                    Insert X(L) in subtree L+1:N
   30                    CALL INSERT(N,X,X(L),L,N)
C            Bring X into full order
                 DO 300 K = N, 2, -1
C                    As tree 1..K is in tree order, X(1) is
C                            largest, so place at top of array,
C                            saving old top in XT
                         XT = X(K)
                         X(K) = X(1)
C                    Restore tree order to tree 1:(K-1) by
C                            inserting XT in subtree 2:K-1
  300                    CALL INSERT(N,X,XT,1,K-1)
         END
```

Both phases make use of a subroutine `INSERT` that inserts an element into an array in tree order so that tree order is maintained:

```
      SUBROUTINE INSERT(N,X,XS,B,T)
         INTEGER N, B, T
         REAL X(1:N), XS
C        Insert XS in subtree X(B+1:T), starting at X(B)
C                which is the initial insert slot, making use
C                of the fact that X(B+1:T) is in tree order
             INTEGER I, J
C            Select initial subtree I
                 I = B
C            Recursively insert in subtree I
   20            CONTINUE
C                Set J to first branch, but exit if I terminal
                     J = I + I ! possible first branch
                     IF (J .GT. T) GOTO 30
C                If there are two branches choose the larger
                     IF( J .LT. T ) THEN
                         IF( X(J) .LT. X(J+1) ) J = J+1
                         ENDIF
C                Exit if XS >= larger branch
                     IF( XS .GE. X(J) ) GOTO 30
C                Make insert slot in larger branch
                     X(I) = X(J)
```

```
C               Select larger branch subtree then continue
                    I = J
                    GOTO 20
C           Place XS into insert slot
   30           X(I) = XS
        END
```

This subroutine in fact comprises an algorithm that calls itself recursively, but this would be so inefficient that not even the purest purist would express it recursively.

Measuring the program's performance

Our goal is to improve the performance of the program; viewed in isolation the goal is to reduce the time that the program takes to execute. We need therefore to measure the speed of execution of the serial sort to use as a basis for comparison. Unfortunately, the execution time for this sort depends not only on the size of the array but also on the way that the data is arranged, so that we will have to be content with estimating the mean performance over a range of sample data. We will assume that the array lengths and the data themselves are uniformly distributed (generated by the built-in random number function). The performance will be tabulated for varying maximum array size and for varying batch size (number of arrays in a batch - `M` in the algorithm above). Reasonably representative figures can only be obtained by ensuring that a large number of arrays are sorted. This happens naturally if the batch size is large but for small batches many repetitions are required - in the figures below there are always at least 100 arrays sorted - this gives enough accuracy for our purpose but we have to be careful when interpreting small differences in performance. Because the X-MP is a multiprocessor, the performance of a processor on one CPU may be degraded by memory accesses made by the others executing other programs, a more-likely affect when the amount of data being processed is large. This is dealt with by making a number of runs and eliminating anomalously slow results.

For the serial algorithm we get performance figures as in Table 1 where the units are time in microseconds per element sorted, the columns are arranged in order of increasing batch size and the rows in increasing maximum array length.

Table 1: Structured serial sort

		size of batch				
		1	4	16	64	256
	1	4.6	2.3	1.7	1.6	1.5
max	4	4.5	3.4	3.2	3.1	3.4
array	16	4.9	4.5	4.4	4.4	4.5
length	64	6.2	6.2	6.2	6.0	6.2
	256	8.1	7.8	7.8	7.8	7.8

This behaviour is much as would be expected. Apart from when the array and batch sizes are very small, the size of the batch doesn't make much difference. The time per element increases appropriately with the log of the array size. (The reason for the increase when batch size is large is unknown - there are a number of such small anomalies in the data collected that we will just ignore).

Optimizing for a serial processor

Most Fortran programmers would be horrified by the above program because by their experience it contains constructs that perform poorly. The worst culprits here are the subroutine calls. There is a natural tendency to remove these by flattening the inner calls, replacing them by the subroutine bodies. In this program it is easy to make such a change for the inner loops where the most benefit will be obtained. Such an improvement has effect as in Table 2.

This has the same general behaviour as the structured version but is faster by a factor of about 1.37, which does show why Fortran programmers tend to be "flat earthers". We will work with both forms of program because we do want to see whether or not such optimization is still worthwhile when parallelism has been introduced.

Table 2: Flattened Serial Sort

		size of batch				
		1	4	16	64	256
max array length	1	4.4	2.2	1.6	1.5	1.5
	4	2.9	1.9	1.7	1.6	1.6
	16	2.9	2.6	2.5	2.5	2.5
	64	4.1	4.1	4.1	3.9	4.1
	256	5.7	5.8	5.7	5.7	5.7

3. Multiprocessing

There are two methods provided to take advantage of multiprocessing in Cray Fortran. One is conventional macro-tasking which requires the programmer to split programs into independent communicating processes. The other approach is called *micro-tasking*; it is very easy to apply to SAMD programs.

Microtasking - The Cray fray

Microtasking as introduced by Cray [2,3,4] is simple, elegant, and easy to use. The programmer is told not to bother thinking of the number of processors that are available but to concentrate on exhibiting the natural parallelism in programs. The basic concept that is introduced is that of a *fray* which is a procedure that may be executed by any number of processors. In our example it is natural for us to pick the procedure for sorting a batch as being a fray - this is done merely by inserting a pseudo-comment at the start of the procedure:

```
CMIC$ MICRO
      SUBROUTINE SORTSET(M,N,BASE,X)
```

The fray will have processors applied to its execution until it has been fully executed, each processor will start at the beginning and, unless directed otherwise, will execute the whole fray using its own copy of the local variables - global variables (parameters and common) will not be replicated for each processor. This will clearly be of little use if each processor does execute the entire procedure, hence sections of the fray must be defined as *control structures* that are to be executed only once but which are composed of sections that may be executed in parallel (for the most part independently) - global variables must only be accessed from code within control structures. Any section of a fray may be defined to be part of a control structure but the most usual approach is to specify that a DO-loop is to be a control structure with every iteration being an independent section. In our example:

```
CMIC$ MICRO
      SUBROUTINE SORTSET(M,N,BASE,X)
         REAL X(*)
         INTEGER M, N(1:M), BASE(1:M)
C        Sort the M arrays X(BASE(I),BASE(I)+N(I)-1)
CMIC$    DO GLOBAL FOR 4
             DO 100 I = 1, M
  100            CALL SORT(N(I),X(BASE(I)))
         END
```

In this case we have not only specified with `DO GLOBAL` that each iteration is an independent section of a control structure but that we also think that it would be a good idea to divide the work into four chunks (remembering that there are four processors). In more complicated programs there will be frays that have some parts that must be completed before others are started and sections that can only be executed one CPU at a time but, in our example, nothing more is required than an indication to the compiler of the maximum number of CPUs that we would like to have applied to our problem. These controls must come, in execution order, before and after the fray:

```
CMIC$    GETCPUS 4

CMIC$    RELCPUS
```

The program must now be processed by a precompiler before execution. The precompiler generates subroutine calls in the Fortran program and also produces a small assembly language program that must be bound into the executable module.

Microtasking performance

With microtasking the performance is even less predictable than before. All of the desired processors are not necessarily available so that, although results from run to run on the same data are mostly consistent, some will be way out of line. Additionally, when the control sections are rather large it seems to be more difficult to claim the attention of all of the CPUs all of the time. The figures in Table 3 are for four processors and are the minimum that was obtained in each case. Although minimum, they do represent the gains to be had when other users are not competing for the system.

Table 3: Structured sort - microtasking with 4 CPUs							
		size of batch					
		1	4	16	64	256	1024
max array length	1	9.6	3.5	1.5	1.0	0.8	0.8
	4	6.6	3.0	1.9	1.8	1.5	1.7
	16	5.7	2.6	2.1	1.5	1.3	1.2
	64	6.6	2.8	2.1	1.9	1.6	1.7
	256	8.1	3.3	2.8	2.3	2.2	2.1

One notable point is the low overhead of microtasking. Even with only one batch (and therefore only one processor) the overhead is negligible except for the trivial cases. The best improvement that is achieved over the serial version is by a factor of 3.7. This shows that memory access conflicts are not serious in this case. The improvement with two CPUs is about 1.9.

The improvements with the flattened optimized program are also listed in Table 4 for later comparison. Here the maximum performance improvement observed is a little less at 3.6, perhaps an indication of the more concentrated data accesses to memory made by the faster algorithm.

Table 4: Flattened Sort - Microtasking with 4 CPUs							
		size of batch					
		1	4	16	64	256	1024
max array length	1	9.6	3.3	1.6	1.0	0.8	0.8
	4	5.0	2.0	1.1	0.9	0.9	0.8
	16	3.5	1.4	0.9	0.7	0.7	1.3
	64	4.3	2.0	1.7	1.2	1.0	1.6
	256	5.7	2.5	2.0	1.7	1.6	1.6

4. Vectorization

There is no obvious use of vectors in our sort algorithm. However, there is a technique that can be applied quite systematically to vectorize such programs (see Lipps[5]). This is not really a task suited to humans as it is far too complex and error prone. It is much better left to compilers but, alas, compilers do not currently do the job. It is our intent to explore what can be done so we will vectorize our SAMD program trying to use rules that can be easily generalized and followed by others.

The first step, SAMD to STMD

The motivation behind what we are doing here is that a computer such as the Cray which is widely considered to be a *vector* computer, can equally be regarded as a classic SIMD (Single Instruction Multiple Data) machine because, with most vector instructions, elements of vectors are manipulated quite independently of each other. Thus, rather than regarding the Cray as a processor that deals with vectors of 64 elements, we can equally (for the most part) regard it as a machine containing 64 processors that are responsive to the same instruction and each of which operates on scalar operands. We can regard our task as one of converting our program to SIMD form.

Eventually we will get a program that will have the one course of execution for each set of data (which is what SIMD requires) but the actual course taken by our serial algorithm may well be dependent on the particular data being operated on. A first step is to alter our serial program so that, aside from loop exits, it will always take the same course of execution and so that it will not be affected by extra loop iterations. Lets call this form Single Thread Multiple Data form or STMD. (It is difficult to define this form in a single phrase - the idea is that for a given batch of data there will be one trace of execution that is a superset of all the others needed for that batch and that the superset could be applied to any in the batch and still give the correct results).

This is not a trivial thing to do. The mechanism is to precede every statement by a mask test that can "turn off" execution when it is not needed and to alter loops so that exits are also made on the basis of testing a mask. Lets see what happens to our example:

```
      SUBROUTINE SORT(N,X)
         INTEGER   N
         REAL      X(1:N)
C        Bring X(1:N) into ascending order using heap sort
             INTEGER   K, L
             REAL      XT
             LOGICAL M1
C            Ensure that N is > 1
                 IF( N .LT. 2 ) RETURN     ! Exit
C            Place X in descending tree order
                 DO 30 L = N/2, 1, -1
                     M1 = (N .GE. 2*L)
C                    Insert X(L) in subtree L+1:N
   30                    CALL INSERT(N,X,X(L),L,N,M1)
C            Bring X into full order
                 DO 300 K = N, 2, -1
                     M1 = (N .GE. K)
C                    As tree 1..K is in tree order, X(1) is
C                            largest, so place at top of array,
C                            saving old top in XT
                         IF (M1) THEN
                             XT = X(K)
                             X(K) = X(1)
                             ENDIF
C                    Restore tree order to tree 1:(K-1) by
C                            inserting XT in subtree 2:K-1
  300                    CALL INSERT(N,X,XT,1,K-1,M1)
               END
```

Here, in the `SORT` procedure, the two loops are made independent of the number of iterations by introducing a mask that if false indicates that the iteration of the loop is to have no effect i.e. the loops will work correctly if the upper bound on the index is replaced by a larger value. In the second loop the mask is applied to all the statements inside the loop. In both loops the mask is passed to the `INSERT` subroutine as a parameter to nullify its execution, if necessary:

```
      SUBROUTINE INSERT(N,X,XS,B,T,M1)
         INTEGER N, B, T
         REAL X(1:N), XS
C        Insert XS in subtree X(B+1:T), starting at X(B)
C                which is the initial insert slot, making use
C                of the fact that X(B+1:T) is in tree order
             INTEGER I, J
             LOGICAL M1, M2, M3
C            Select initial subtree I
                IF (M1) I = B
C            Recursively insert in subtree I
                M2 = M1
   20           CONTINUE
C               Set J to first branch, but exit if I terminal
                    IF (M2) J = I + I !possible first branch
                    !IF (J .GT. T) GOTO 30
                        IF (M2) M2 = (J .LE. T)
                        IF (.NOT. M2) GOTO 30
C               If there are two branches choose the larger
                    !IF( J .LT. T ) THEN
                        !IF( X(J) .LT. X(J+1) ) J = J+1
                        !ENDIF
                        M3 = M2
                        IF (M3) M3 = (J .LT. T)
                        IF (M3) M3 = (X(J) .LT. X(J+1))
                        IF (M3) J = J+1
C               Exit if XS >= larger branch
                    !IF( XS .GE. X(J) ) GOTO 30
                        IF (M2) M2 = (XS .LT. X(J))
                        IF (.NOT. M2) GOTO 30
C               Make insert slot in larger branch
                    !X(I) = X(J)
                        IF (M2) X(I) = X(J)
C               Select larger branch subtree then continue
                    IF (M2) I = J
                    GOTO 20
C            Place XS into insert slot
   30           IF (M1) X(I) = XS
         END
```

This has a lot more content of interest. Every statement in the procedure is masked by `M1` passed as a parameter, or by some restriction of `M1`. The loop is masked by mask `M2` which is initially set from `M1`. The exits from the loop are made on the basis of testing mask `M2`, with the mask being restricted to ensure that extra iterations of the loop can do no harm. In the depth of the loop there is highly conditional operation. This is handled by introducing a further temporary mask `M3` and restricting that.

What we now have is a serial program that uses standard Fortran but has our STMD character. It should still produce the same results as the SAMD form which is good check at this stage (although the performance will be severely degraded).
The general approach that is necessary should be clear, although there are many other program constructs hat are not in this particular example.

Step 2: STMD with unique data

The next step is tedious but mechanical. We must take our scalar variables and extend them so that there is a unique name for each variable/data combination. The tool we have for that in Fortran is to introduce an array to replace each scalar variable. For example, a mask such as `M1` used for the `M` batches of data must be replaced by an array `M1(1:M)` so that `M1(W)` is the mask used with batch `W`. Herbert Lipps[5] calls this process "colouring" the program.

The outermost loop calling our STMD procedure remains the same except that the iteration variable is passed to the procedure so that differentiation may take place. `SORTSET` becomes:

```
      SUBROUTINE SORTSET(M,N,BASE,X)
         REAL X(*)
         INTEGER M,           !number of arrays to be sorted
                N(1:M),       !length of each array
                BASE(1:M)     !base of each array
C        Sort the M arrays X(BASE(W)+1,BASE(W)+N), W = 1..M
             INTEGER W
             DO 100 W = 1, M
  100           CALL SORT(M,N,BASE,X,W)
         END
```

The remainder of the transformation is very straightforward. As an illustration of what is involved, here is the second loop of the SORT procedure:

```
   30                CALL INSERT(M,BASE,X,X(BASE(W)+L),L,N(W),M1,W)
C            Bring X into full order
                DO 300 K = N(W), 2, -1
                    M1(W) = (N(W) .GE. K)
C                   As tree 1..K is in tree order, X(1) is
C                           largest, so place at top of array,
C                           saving old top in XT
                    IF (M1(W)) THEN
                             XT(W) = X(BASE(W)+K)
                             X(BASE(W)+K) = X(BASE(W)+1)
                             ENDIF
C                   Restore tree order to tree 1:(K-1) by
C                           inserting XT in subtree 2:K-1
                        IF (M1(W)) KA(W) = K-1
  300                   CALL INSERT(M,BASE,X,XT(W),1,KA(W),M1,W)
  100    CONTINUE
                END
```

Note how variables that are different for each array, such as the array size `N` and the mask `M1`, are elevated to be arrays of size `(1:M)`. For the array `X`, into which the arrays being sorted are already packed, we have to distinguish elements by referring to the actual offset inside `X` e.g. `X(K)` is replaced by `X(BASE(W)+K)`. The only change that requires a little thought is to ensure that subroutine arguments are uniform across all calls. In this case the sixth parameter to `INSERT` is `N(W)` in the first call so we have to introduce an array `KA` so that the second call also has a `W`-dependent variable in that position.

The performance of this version should be much worse than before but, we hope, it will all be worth it after the next step.

Step 3: STMD to SIMD

The change to make here is to alter the program to a form that is suited to a SIMD machine. In essence, the same instructions must be executable for all data simultaneously, with masks used to stop individual processors executing where that is appropriate. The previous step gave unique names to the variables used to sort each array. The program is now such that it does not matter how the instructions to sort each batch are interleaved as long as the instructions within a batch remain in the same order. Thus we can take the loop over all batches in the outermost loop in procedure `SORTSET` and move the loop "in" as far as we like, so that we iterate over a small set of statements. Such small inner loops can be regarded as the parallel commands to an SIMD machine.

In this example, if the `W` loop in `SORTSET` is moved inwards then `SORTSET` no longer has any separate purpose and could well be combined with `SORT`:

```
      SUBROUTINE SORTSET(M,N,BASE,X)
         REAL X(*)
         INTEGER M,            !number of arrays to be sorted
                 N(1:M),       !length of each array
                 BASE(1:M)     !base of each array
C        Sort the M arrays X(BASE(W)+1,BASE(W)+N), W = 1..M
             CALL SORT(M,N,BASE,X)
         END
```

Inside `SORT`, things get messy because the `W` loops now cut right across our program's structure. They are made obvious in the following listing of `SORT` by a highlight of `****`s. (The complete program listing is in the appendix, only an outline is given here.)

```
      SUBROUTINE SORT(M,N,BASE,X)
      ............
                 NMAX = N(1)
                 DO 839 I = 2, M
  839            IF (N(I) .GT. NMAX) NMAX = N(I)
C            Ensure that NMAX is > 1
                 IF( NMAX .LT. 2 ) RETURN     ! EXIT
```

```
C           Place X in descending tree order
               DO 30 L = NMAX/2, 1, -1
      DO 991 W = 1, M !****************************************
                   M1(W) = (N(W) .GE. 2*L)
                   IF (M1(W)) XT(W) = X(BASE(W)+L)
  991 CONTINUE !*******************************************
   30                  CALL INSERT(M,BASE,X,XT,L,N,M1)
C           Bring X into full order
               DO 300 K = NMAX, 2, -1
      DO 992 W = 1, M !****************************************
                   M1(W) = (N(W) .GE. K)
                       IF (M1(W)) XT(W) = X(BASE(W)+K)
                       IF (M1(W)) X(BASE(W)+K) = X(BASE(W)+1)
                       IF (M1(W)) KA(W) = K-1
  992 CONTINUE !*******************************************
  300                  CALL INSERT(M,BASE,X,XT,1,KA,M1)
         END
```

An SIMD program has to have the same course of execution for all data so the variable-sized `DO`-loops have to be replaced by just one loop. Thus the introduction of the `NMAX` variable which must be calculated first. Notice that the number of calls on the `INSERT` subroutine has been reduced to only two.

More change is needed inside `INSERT`:

```
      SUBROUTINE INSERT(M,BASE,X,XS,B,T,M1)
                  ..........
      DO 321 W = 1, M !**********************************
               IF (M1(W)) I(W) = B
               M2(W) = M1(W)
  321 CONTINUE!****************************************
   20          CONTINUE
      DO 322 W = 1, M !**********************************
                   IF (M2(W)) J(W) = I(W) + I(W)
                       IF (M2(W)) M2(W) = (J(W) .LE. T(W))
  322 CONTINUE !***************************************
                       !IF (.NOT. M2(W)) GOTO 30
                           DO 323 I6 = 1, M
  323                          IF (M2(I6)) GOTO 324
                           GOTO 30
  324                      CONTINUE
      DO 325 W = 1, M !**********************************
C              If there are two branches choose the larger
                       M3(W) = M2(W) .AND. (J(W) .LT. T(W))
                       IF (M3(W)) M3(W) = (X(BASE(W)+J(W))
     1                                .LT. X(BASE(W)+J(W)+1))
                       IF (M3(W)) J(W) = J(W)+1
```

```
C               Exit if XS >= larger branch
                        IF (M2) M2 = (XS .LT. X(J))
                        IF (M2(W)) M2(W) =
     1                          (XS(W) .LT. X(BASE(W)+J(W)))
  325 CONTINUE !*******************************************
                        !IF (.NOT. M2(W)) GOTO 30
                            DO 326 I6 = 1, M
  326                           IF (M2(I6)) GOTO 327
                            GOTO 30
  327                       CONTINUE
     DO 328 W = 1, M !****************************************
C               Make insert slot in larger branch
                        IF (M2(W)) X(BASE(W)+I(W)) =
     1                                      X(BASE(W)+J(W))
C               Select larger branch subtree then continue
                    IF (M2(W)) I(W) = J(W)
  328 CONTINUE !********************************************
                    GOTO 20
C           Place XS into insert slot
   30           CONTINUE
     DO 329 W = 1, M !*****************************************
                IF (M1(W)) X(BASE(W)+I(W)) = XS(W)
  329 CONTINUE !********************************************
     END
```

Just as within `SORT`, the only changes that have to be made to individual statements are at control points where the total status of all processors has to be considered. In this case there have to be loops introduced to test all mask bits before exiting from any of the various loops.

This can now be compiled and tested. The CFT77 compiler has to be told to forget its concerns about vectorizing some loops (the `CDIR$ IVDEP` controls) but it can be cajoled into vectorizing all of the inner loops (except those, such as that for finding `NMAX`, that are essentially serial). The performance obtained is as in Table 5.

Table 5: Structured SIMD sort

		size of batch					
		1	4	16	64	256	1024
max array length	1	71	19	4.6	1.3	1.8	0.6
	4	91	36	9.7	3.2	1.7	1.6
	16	112	49	14	6.1	3.9	3.2
	64	121	55	18	8.9	6.6	5.7
	256	130	61	22	11	8.9	7.9

Step 4: Optimizing for the Cray

The above is not very good at all, in fact worse than when we started. The main problem is that the compiler guarantees correct code and is very cautious in the way that it handles conditional assignments. For example, with

```
IF (M3(W)) M3(W) = J(W) .LE. T(W)
```

the compiler will ensure that there are no errors resulting from the J and T comparison by generating code like:

```
set s1 to element of J selected by first 1 bit of M3
set temporary vector T1 to J but with s1 where M3 false
create T2 likewise from T
compare T1 with T2 to give T3
assign to M3 from M3 or T3 depending on M3
```

What we have to do now is to recast our program so that the compiler is less cautious. This is a highly machine and algorithm dependent operation. On the Cray, in particular, it is very dangerous to the integrity of the program because the vector mask does not apply to operations other than loads, stores, and merges. Any errors caused by garbage being selected are not masked off.

Here are examples of what we can do: In `SORT`:

Replace:
```
IF (M1(W)) I(W) = B
```
by:
```
I(W) = B
```

This is safe, setting extra values of `I` is OK because they will not be used and the value `B` is constant and valid.

There are many other examples like this:

Replace:
```
IF (M2(W)) J(W) = I(W) + I(W)
```
by:
```
J(W) = I(W) + I(W)
```

This does no harm because we have now assured that even unused values of `I` are initialized.

Replace:
```
IF (M2(W)) M2(W) = (J(W) .LE. T(W))
```
by:
```
M2(W) = (J(W) .LE. T(W))
```

Which is OK because `J` and `T` are fully initialized and J is increasing so that once the condition becomes false it remains false.

It may seem OK to take the masks off everything that doesn't store into our arrays `X`. Unfortunately, making the following change will cause trouble because accesses will be made to invalid data:

Change:

```
IF (M2(W)) M2(W) = (XS(W) .LT. X(BASE(W)+J(W))
```

To:

```
M2(W) = M2(W) .AND. (XS(W) .LT. X(BASE(W)+J(W))
```

This is bad because it could cause accesses to be made beyond the end of `X`. Care is needed here but the obvious trick in this case is to extend `X` a bit more:

Replace:

```
REAL X(1:M*NMAX)
```

by:

```
REAL X(1:M*NMAX+NMAX+1)
```

Then we can recast some of the stores that must remain masked into a form provided by Cray Fortran that will force invalid but safe references to be made. The `CVMGT` function merges its two first operands based on the mask in the 3rd operand:

Replace:

```
IF (M2(W)) I(W) = J(W)
```

By:

```
I(W) = CVMGT(J(W),I(W),M2(W))
```

Another improvement is to reduce the number of exits from loops to just one. This is possible because all statements are guarded by masks that ensure that some extra executions do no harm.

Now, the moment of truth, the performance figures for this final form are as in Table 6. The improvement is by a factor of 2.23 when the number of batches is large. This is good but we must remember that in the process of vectorizing our program we have reduced subroutine calls, effectively flattening it, so we must take into account the performance improvement due to flattening.

Table 6: Structured sort – optimized SIMD

		size of batch					
		4	16	64	256	1024	4096
max array length	1	19	4.5	1.3	1.9	0.7	0.4
	4	49	15	4.4	1.9	1.3	1.4
	16	64	19	5.9	3.3	2.3	2.0
	64	66	20	6.6	3.7	2.9	2.8
	256	71	22	8.0	4.7	3.8	3.5

The same process of vectorization an be applied to the flattened sort or, alternatively, the last program can be flattened. There are, in fact, a few more optimizations possible to the flattened program, leading to figures as in Table 7. So the flattened and unflattened versions have about the same performance when the array and batch sizes are large. This is disappointing on the one hand in that it means that the performance gain from vectorizing is only 1.63. On the other hand, it is heartening in that it means that we do not have to bother flattening our programs if we are vectorizing them using this approach.

Table 7: Flattened Sort Optimized SIMD

		size of batch					
		4	16	64	256	1024	4096
max array length	1	36	8.7	3.3	3.0	1.0	0.5
	4	49	5.4	2.1	1.4	1.1	1.1
	16	11	4.1	2.8	2.2	2.0	2.0
	64	11	4.8	2.9	2.8	2.8	2.8
	256	13	5.7	3.9	3.6	3.6	3.5

Introduction of vector statements

The messy version of the program that we just created can be much improved in readability by making the inner loops each apply to a single scalar statement then replacing each loop by a vector statement. This approach has to be used with caution or else the compiler will generate poor code because we have asked it to treat each statement as a complete operation on the whole `M` arrays. The overhead of breaking the vectors into 64 element chunks is replicated for every statement. The compiler is not able to make use of temporary values in vector registers. Note that this isn't an insoluble problem but requires that the compiler is smart enough to, essentially, unvectorize the program, and combine adjacent loops.

What is possible with vectorization?

The technique that we have been through certainly does not produce the best SIMD code for the program. For example, the value `NMAX` is calculated over all arrays whereas it could be restricted to being the maximum for the batch being processed (in the case of the Cray to the maximum for each group of 64).

Additionally, there are optimizations that apply to real machines like the Cray that are not SIMD in reality. For example, it is possible to select for action only those subarrays that are "active" by using compress instructions and then change the vector length so as to operate on the set of reduced length arrays. There is no way to do this in Fortran (it is explored fully in Herbert Lipps' work[5]).

Such compression has the advantage that it will only do actual vector operations that are necessary and safe. However, to counter the performance gained thereby, it has the drawback that extra operations will be required to actually select the data required, e.g in our example to compress the number of bases. Whether or not there would be an overall performance gain is unclear in this case.

In general, the benefits from vectorization do depend very much on the problem. If most iterations of loops involve unmasked values then the speed-up is much more marked. In our example, if all arrays being sorted are the same size of 256 then the time per element for length 256 arrays is improved from 8.5 to 2.4 microseconds, an improvement factor of 3.4 (2.7 due to the vectorization).

5. Vectorization and Multiprocessing Combined

The two techniques are quite independent of each other so we would expect them to compound. With 4 processors the structured vectorized sort performs as in Table 8. The improvement over the flattened sort (table 2) is by a factor of 6 at its best. We would have expected 3.6 from MP and 1.63 from vectorization for a total of about 5.9 so the two techniques have compounded with little loss. In the case of equisized length 256 arrays the delay per element gets as low as .75 microseconds for an overall improvement over flattening of 8.7 which is a most heartening improvement.

Table 8: Structured sort, vectorized and MPed

		size of batch					
		4	16	64	256	1024	4096
max array length	1	69	17	4.4	1.2	2.3	.60
	4	107	46	13	3.1	.84	.82
	16	118	46	13	4.0	1.0	.76
	64	122	51	14	3.9	1.2	.78
	256	128	53	15	4.0	1.3	.95

6. Conclusions

We have seen that it is mechanical but hard work to manually vectorize a SAMD program. The performance gain is good enough to be worth the effort if the program is to widely used. There is no doubt that the task is for compilers, not for people, but the number of special cases that require algorithm knowledge is such that compilers will have difficulty in getting the best results. The task would be much easier, for humans or compilers, on computers that are able to apply masks to all vector operations

It is nice to confirm that programs can be left in structured form. However, it is much easier to use multiprocessing via microtasking where the performance gain is very good and is much easier to obtain (though it does not improve system-wide throughput).

Acknowledgment: This paper was first prepared while the author was on sabbatical leave during 1988 visiting the data handling division of European Organisation for Nuclear Research (CERN) in Geneva. Thanks is due to The University of Auckland and CERN for support during this period, and particularly to Herbert Lipps and Mike Metcalf for their encouragement.

References

[1] Iliffe, J. K. : Advanced Computer Design. Prentice Hall International. 1982

[2] Cray Research Incorporated.: Cray X-MP Multitasking, Programmer's Reference Manual, Publication No. SR-0222, 1987

[3] Fatoohi, R. A.: "Multitasking on the Cray Y-MP: An Experiment with a 2-D Navier-Stokes Code. International Journal of High Speed Computing. Vol. 1, No. 3, Sept. 1989, p. 433.

[4] Doran R. W.: "Multiprocessing via Microtasking", Proceedings of the 11th New Zealand Computer Conference, Lindsay J. Groves Editor, Wellington 16-18th August 1989, p. 335.

[5] Lipps, H.: Unpublished report, CERN, 1988.

Appendix - Program Listings

Disclaimer - these examples were tested and run on a Cray X-MP. However, since then they have been reformatted and it is not guaranteed that some transcription errors may have been committed.

A. Structured serial sort

```
     SUBROUTINE SORTSET(M,N,BASE,X)
        REAL X(*)
        INTEGER M,          !number of arrays to be sorted
                N(1:M),     !length of each array
                BASE(1:M)   !base of each array
C       Sort the M arrays: X(BASE(W)+1:BASE(W)+N(W)), W = 1..M
            INTEGER W
            DO 100 W = 1, M
  100           CALL SORT(N(W),X(BASE(W)+1))
        END
C
     SUBROUTINE SORT(N,X)
        INTEGER   N
        REAL      X(1:N)
C       Bring X(1:N) into ascending order using heap sort
            INTEGER   K, L
            REAL      XT
C           Ensure that N is > 1
                IF( N .LT. 2 ) RETURN     ! Exit
C           Place X into descending tree order
                DO 30 L = N/2, 1, -1
C                   Insert X(L) in subtree L+1:N
   30                   CALL INSERT(N,X,X(L),L,N)
C           Bring X into full order
                DO 300 K = N, 2, -1
C                   As tree 1..K is in tree order, X(1) is
C                           largest, so place at top of array,
C                           saving old top in XT
                        XT = X(K)
                        X(K) = X(1)
C                   Restore tree order to tree 1:(K-1) by
C                           inserting XT in subtree 2:K-1
  300                   CALL INSERT(N,X,XT,1,K-1)
        END
C
```

```
      SUBROUTINE INSERT(N,X,XS,B,T)
         INTEGER N, B, T
         REAL X(1:N), XS
C        Insert XS in subtree X(B+1:T), starting at X(B)
C                which is the initial insert slot, making use
C                of the fact that X(B+1:T) is in tree order
             INTEGER I, J
C            Select initial subtree I
                 I = B
C            Recursively insert in subtree I
   20            CONTINUE
C                Set J to first branch, but exit if I terminal
                     J = I + I ! possible first branch
                     IF (J .GT. T) GOTO 30
C                If there are two branches choose the larger
                     IF( J .LT. T ) THEN
                         IF( X(J) .LT. X(J+1) ) J = J+1
                         ENDIF
C                Exit if XS >= larger branch
                     IF( XS .GE. X(J) ) GOTO 30
C                Make insert slot in larger branch
                     X(I) = X(J)
C                Select larger branch subtree then continue
                     I = J
                     GOTO 20
C            Place XS into insert slot
   30            X(I) = XS
         END
```

B. Single thread structured sort

```
      SUBROUTINE SORTSET(M,N,BASE,X)
         REAL X(*)
         INTEGER M,          !number of arrays to be sorted
                 N(1:M),     !length of each array
                 BASE(1:M)   !base of each array
C        Sort the M arrays: X(BASE(W)+1:BASE(W)+N(W)), W = 1..M
             INTEGER W
             DO 100 W = 1, M
  100            CALL SORT(N(W),X(BASE(W)+1))
         END
C
      SUBROUTINE SORT(N,X)
         INTEGER   N
         REAL      X(1:N)
C        Bring X(1:N) into ascending order using heap sort
             INTEGER   K, L
             REAL      XT
             LOGICAL M1
C            Ensure that N is > 1
                 IF( N .LT. 2 ) RETURN     ! Exit
C            Place X in descending tree order
                 DO 30 L = N/2, 1, -1
                     M1 = (N .GE. 2*L)
C                    Insert X(L) in subtree L+1:N
   30                    CALL INSERT(N,X,X(L),L,N,M1)
C            Bring X into full order
                 DO 300 K = N, 2, -1
                     M1 = (N .GE. K)
C                    As tree 1..K is in tree order, X(1) is
C                            largest, so place at top of array,
C                            saving old top in XT
                         IF (M1) THEN
                             XT = X(K)
                             X(K) = X(1)
                             ENDIF
C                    Restore tree order to tree 1:(K-1) by
C                            inserting XT in subtree 2:K-1
  300                    CALL INSERT(N,X,XT,1,K-1,M1)
                END
C
```

```
      SUBROUTINE INSERT(N,X,XS,B,T,M1)
         INTEGER N, B, T
         REAL X(1:N), XS
C        Insert XS in subtree X(B+1:T), starting at X(B)
C                which is the initial insert slot, making use
C                of the fact that X(B+1:T) is in tree order
             INTEGER I, J
             LOGICAL M1, M2, M3
C            Select initial subtree I
                IF (M1) I = B
C            Recursively insert in subtree I
                M2 = M1
   20           CONTINUE
C               Set J to first branch, but exit if I terminal
                    IF (M2) J = I + I !possible first branch
                    !IF (J .GT. T) GOTO 30
                        IF (M2) M2 = (J .LE. T)
                        IF (.NOT. M2) GOTO 30
C               If there are two branches choose the larger
                    !IF( J .LT. T ) THEN
                        !IF( X(J) .LT. X(J+1) ) J = J+1
                        !ENDIF
                        M3 = M2
                        IF (M3) M3 = (J .LT. T)
                        IF (M3) M3 = (X(J) .LT. X(J+1))
                        IF (M3) J = J+1
C               Exit if XS >= larger branch
                    !IF( XS .GE. X(J) ) GOTO 30
                        IF (M2) M2 = (XS .LT. X(J))
                        IF (.NOT. M2) GOTO 30
C               Make insert slot in larger branch
                    !X(I) = X(J)
                        IF (M2) X(I) = X(J)
C               Select larger branch subtree then continue
                    IF (M2) I = J
                    GOTO 20
C            Place XS into insert slot
   30           IF (M1) X(I) = XS
         END
```

C. Single thread with unique data

```
      SUBROUTINE SORTSET(M,N,BASE,X)
         REAL X(*)
         INTEGER M,          !number of arrays to be sorted
                 N(1:M),     !length of each array
                 BASE(1:M)   !base of each array
C        Sort the M arrays X(BASE(W)+1,BASE(W)+N), W = 1..M
             INTEGER W
             DO 100 W = 1, M
  100            CALL SORT(M,N,BASE,X,W)
         END
C
      SUBROUTINE SORT(M,N,BASE,X,W)
         INTEGER   M, N(1:M), BASE(1:M), W
         REAL      X(*)
C        Bring X(1:N) into ascending order using heap sort
             INTEGER K, L, KA(1:M)
             REAL    XT(1:M)
             LOGICAL M1(1:M)
C            Ensure that N is > 1
                 IF( N(W) .LT. 2 ) RETURN     ! EXIT
C            Place X in descending tree order
                 DO 30 L = N(W), 1, -1
                     M1(W) = (N(W)/2 .GE. 2*L)
C                    Insert X(L) in subtree L+1:N
                         IF (M1(W)) XT(W) = X(BASE(W)+L)
   30                    CALL INSERT(M,BASE,X,X(BASE(W)+L),L,N(W),M1,W)
C            Bring X into full order
                 DO 300 K = N(W), 2, -1
                     M1(W) = (N(W) .GE. K)
C                    As tree 1..K is in tree order, X(1) is
C                            largest, so place at top of array,
C                            saving old top in XT
                         IF (M1(W)) THEN
                              XT(W) = X(BASE(W)+K)
                              X(BASE(W)+K) = X(BASE(W)+1)
                              ENDIF
C                    Restore tree order to tree 1:(K-1) by
C                            inserting XT in subtree 2:K-1
                         IF (M1(W)) KA(W) = K-1
  300                    CALL INSERT(M,BASE,X,XT(W),1,KA(W),M1,W)
  100     CONTINUE
                 END
C
```

```
      SUBROUTINE INSERT(M,BASE,X,XS,B,T,M1,W)
         INTEGER M, BASE(1:M), B, T(1:M)
         REAL X(*), XS(1:M)
C        Insert XS in subtree X(B+1:T), starting at X(B)
C               which is the initial insert slot, making use
C               of the fact that X(B+1:T) is in tree order
            INTEGER I(1:M), J(1:M), W
            LOGICAL M1(1:M), M2(1:M), M3(1:M), M4(1:M)
C           Select initial subtree I
               IF (M1(W)) I(W) = B
C           Recursively insert from subtree I
               M2(W) = M1(W)
   20          CONTINUE
C              Set J to first branch, but exit if I terminal
                  IF (M2(W)) J(W) = I(W) + I(W) !possible
C                                                  1st branch
                  !IF (J .GT. T) GOTO 30
                     IF (M2(W)) M2(W) = (J(W) .LE. T(W))
                     IF (.NOT. M2(W)) GOTO 30
C              If there are two branches choose the larger
                  !IF( J .LT. T ) THEN
                     !IF( X(J) .LT. X(J+1) ) J = J+1
                     !ENDIF
                     M3(W) = M2(W)
                     IF (M3(W)) M3(W) = (J(W) .LT. T(W))
                     IF (M3(W)) M3(W) = (X(BASE(W)+J(W))
     1                           .LT. X(BASE(W)+J(W)+1))
                     IF (M3(W)) J(W) = J(W)+1
C              Exit if XS >= larger branch
                  !IF( XS .GE. X(J) ) GOTO 30
                     IF (M2(W)) M2(W) =
     1                       (XS(W) .LT. X(BASE(W)+J(W)))
                     IF (.NOT. M2(W)) GOTO 30
C              Make insert slot in larger branch
                  !X(I) = X(J)
                     IF (M2(W)) X(BASE(W)+I(W)) =
     1                                   X(BASE(W)+J(W))
C              Select larger branch subtree then continue
                  IF (M2(W)) I(W) = J(W)
                  GOTO 20
C           Place XS into insert slot
   30          IF (M1(W)) X(BASE(W)+I(W)) = XS(W)
         END
```

D. SIMD structured sort

```
      SUBROUTINE SORTSET(M,N,BASE,X)
         REAL X(*)
         INTEGER M,          !number of arrays to be sorted
     1           N(1:M),     !length of each array
     2           BASE(1:M)   !base of each array
C        Sort the M arrays X(BASE(W)+1,BASE(W)+N), W = 1..M
             CALL SORT(M,N,BASE,X)
         END
C
      SUBROUTINE SORT(M,N,BASE,X)
         INTEGER   M, N(1:M), BASE(1:M), NMAX
         REAL      X(*)
C        Bring X(1:N) into ascending order using heap sort
             INTEGER   K, L, W, KA(1:M)
             REAL      XT(1:M)
             LOGICAL M1(1:M)
C            Find maximum N for batch
                 NMAX = N(1)
                 DO 839 I = 2, M
  839            IF (N(I) .GT. NMAX) NMAX = N(I)
C            Ensure that NMAX is > 1
                 IF( NMAX .LT. 2 ) RETURN     ! EXIT
C            Place X in descending tree order
                 DO 30 L = NMAX/2, 1, -1
      DO 991 W = 1, M !**************************************
                     M1(W) = (N(W) .GE. 2*L)
C                Insert X(L) in subtree L+1:N
                     IF (M1(W)) XT(W) = X(BASE(W)+L)
  991 CONTINUE !*********************************************
   30            CALL INSERT(M,BASE,X,XT,L,N,M1)
C            Bring X into full order
                 DO 300 K = NMAX, 2, -1
CDIR$ IVDEP
      DO 992 W = 1, M !**************************************
                     M1(W) = (N(W) .GE. K)
C                    As tree 1..K is in tree order, X(1) is
C                            largest, so place at top of array,
C                            saving old top in XT
                         IF (M1(W)) XT(W) = X(BASE(W)+K)
                         IF (M1(W)) X(BASE(W)+K) = X(BASE(W)+1)
C                    Restore tree order to tree 1:(K-1) by
C                            inserting XT in subtree 2:K-1
                         IF (M1(W)) KA(W) = K-1
  992 CONTINUE !*********************************************
  300                    CALL INSERT(M,BASE,X,XT,1,KA,M1)
         END
C
```

```
      SUBROUTINE INSERT(M,BASE,X,XS,B,T,M1)
         INTEGER M, BASE(1:M), B, T(1:M)
         REAL X(*), XS(1:M)
C        Insert XS in subtree X(B+1:T), starting at X(B)
C                which is the initial insert slot, making use
C                of the fact that X(B+1:T) is in tree order
             INTEGER I(1:M), J(1:M), W
             LOGICAL M1(1:M), M2(1:M), M3(1:M), M4(1:M)
CDIR$ IVDEP
      DO 321 W = 1, M !****************************************
C            Select initial subtree I
                IF (M1(W)) I(W) = B
C            Recursively insert from subtree I
                M2(W) = M1(W)
  321 CONTINUE!*******************************************
   20           CONTINUE
CDIR$  IVDEP
      DO 322 W = 1, M !****************************************
C               Set J to first branch, but exit if I terminal
                    IF (M2(W)) J(W) = I(W) + I(W) !possible
C                                                1st branch
                    !IF (J .GT. T) GOTO 30
                        IF (M2(W)) M2(W) = (J(W) .LE. T(W))
  322 CONTINUE !******************************************
                        !IF (.NOT. M2(W)) GOTO 30
                            DO 323 I6 = 1, M
  323                           IF (M2(I6)) GOTO 324
                            GOTO 30
  324                       CONTINUE
CDIR$ IVDEP
      DO 325 W = 1, M !****************************************
C               If there are two branches choose the larger
                    !IF( J .LT. T ) THEN
                        !IF( X(J) .LT. X(J+1) ) J = J+1
                        !ENDIF
                        M3(W) = M2(W) .AND. (J(W) .LT. T(W))
                        IF (M3(W)) M3(W) = (X(BASE(W)+J(W))
     1                              .LT. X(BASE(W)+J(W)+1))
                        IF (M3(W)) J(W) = J(W)+1
C               Exit if XS >= larger branch
                    !IF( XS .GE. X(J) ) GOTO 30
                        IF (M2(W)) M2(W) =
     1                          (XS(W) .LT. X(BASE(W)+J(W)))
                        IF (.NOT. M2(W)) GOTO 30
  325 CONTINUE !******************************************
                        !IF (.NOT. M2(W)) GOTO 30
                            DO 326 I6 = 1, M
  326                           IF (M2(I6)) GOTO 327
                            GOTO 30
  327                       CONTINUE
CDIR$ IVDEP
```

```
      DO 328 W = 1, M !****************************************
C                Make insert slot in larger branch
                     !X(I) = X(J)
                          IF (M2(W)) X(BASE(W)+I(W)) =
     1                                           X(BASE(W)+J(W))
C                Select larger branch subtree then continue
                     IF (M2(W)) I(W) = J(W)
   328 CONTINUE !********************************************
                     GOTO 20
C            Place XS into insert slot
    30           CONTINUE
CDIR$ IVDEP
      DO 329 W = 1, M !****************************************
                 IF (M1(W)) X(BASE(W)+I(W)) = XS(W)
   329 CONTINUE !********************************************
      END
```

E. Sort optimized for the Cray X-MP

```
S     SUBROUTINE SORTSET(M,N,BASE,X)
         REAL X(*)
         INTEGER M,          !number of arrays to be sorted
     1           N(1:M),     !length of each array
     2           BASE(1:M)   !base of each array
C        Sort the M arrays X(BASE(W)+1,BASE(W)+N), W = 1..M
         INTEGER   M, N(1:M), BASE(1:M), NMAX
         REAL      X(*)
C        Bring X(1:N) into ascending order using heap sort
             INTEGER   K, L, W, KA(1:M)
             REAL      XT(1:M)
             LOGICAL M1(1:M)
C            Find maximum N for batch
                 NMAX = N(1)
                 DO 839 I = 2, M
  839            IF (N(I) .GT. NMAX) NMAX = N(I)
C            Ensure that NMAX is > 1
                 IF( NMAX .LT. 2 ) RETURN     ! EXIT
C            Place X in descending tree order
                 DO 30 L = NMAX/2, 1, -1
      DO 991 W = 1, M !**************************************
                     M1(W) = (N(W) .GE. 2*L)
C                Insert X(L) in subtree L+1:N
                     XT(W) = X(BASE(W)+L)
  991 CONTINUE !******************************************
   30            CALL INSERT(M,BASE,X,XT,L,N,M1)
C        Bring X into full order
                 DO 300 K = NMAX, 2, -1
CDIR$ IVDEP
      DO 992 W = 1, M !**************************************
                     M1(W) = (N(W) .GE. K)
C                    As tree 1..K is in tree order, X(1) is
C                            largest, so place at top of array,
C                            saving old top in XT
                         XT(W) = X(BASE(W)+K)
                         X(BASE(W)+K) =
      1                  CVMGT(X(BASE(W)+1),X(BASE(W)+K),M1(W))
C                    Restore tree order to tree 1:(K-1) by
C                            inserting XT in subtree 2:K-1
                         KA(W) = K-1
  992 CONTINUE !********************************************
  300                    CALL INSERT(M,BASE,X,XT,1,KA,M1)
         END
C
```

```
      SUBROUTINE INSERT(M,BASE,X,XS,B,T,M1)
         INTEGER M, BASE(1:M), B, T(1:M)
         REAL X(*), XS(1:M)
C        Insert XS in subtree X(B+1:T), starting at X(B)
C                which is the initial insert slot, making use
C                of the fact that X(B+1:T) is in tree order
             INTEGER I(1:M), J(1:M), W
             LOGICAL M1(1:M), M2(1:M), M3(1:M), M4(1:M)
CDIR$ IVDEP
      DO 321 W = 1, M !**************************************
C            Select initial subtree I
                IF (M1(W)) I(W) = B
C            Recursively insert from subtree I
                M2(W) = M1(W)
  321 CONTINUE!*******************************************
   20           CONTINUE
C               Single uniform exit point for loop
                       !IF (.NOT. M2) goto 30
                           DO 323 I6 = 1, M
  323                          IF (M2(I6)) GOTO 324
                           GOTO 30
  324                      CONTINUE
CDIR$  IVDEP
      DO 328 W = 1, M !**************************************
C               Set J to first branch, but exit if I terminal
                    J(W) = I(W) + I(W) !possible first branch
                    !IF (J .GT. T) GOTO 30
                        M2(W) = M2(W) .AND. (J(W) .LE. T(W))
C               If there are two branches choose the larger
                    !IF( J .LT. T ) THEN
                        !IF( X(J) .LT. X(J+1) ) J = J+1
                        !ENDIF
                        M3(W) = M2(W) .AND. (J(W) .LT. T(W))
                        M3(W)=M3(W).AND.
    1                   (X(BASE(W)+J(W)).LT.X(BASE(W)+J(W)+1))
                        J(W) = CVMGT(J(W)+1,J(W),M3(W))
C               Exit if XS >= larger branch
                    M2(W)=M2(W).AND.(XS(W).LT.X(BASE(W)+J(W)))
C               Make insert slot in larger branch
                    !X(I) = X(J)
                        X(BASE(W)+I(W))=CVMGT(X(BASE(W)+J(W)),
    1                       X(BASE(W)+I(W)),M2(W))
C               Select larger branch subtree then continue
                    I(W) = CVMGT(J(W),I(W),M2(W))
  328 CONTINUE !*****************************************
                    GOTO 20
C            Place XS into insert slot
  30            CONTINUE
CDIR$ IVDEP
      DO 329 W = 1, M !**************************************
                X(BASE(W)+I(W)) =  CVMGT(XS(W),X(BASE(W)+I(W))
```

```
     1                                          ,M1(W))
 329 CONTINUE !*******************************************
     END
```