AIPS MEMO NO. 33

# Gridding Synthesis Data on Vector Machines

Donald C. Wells and William D. Cotton

National Radio Astronomy Observatory [1]
Edgemont Road, Charlottesville, VA 22901
(804)296-0211, FTS=938-1271

30 January 1985

## Abstract

An algorithm which is intended to grid aperture synthesis visibility measurements efficiently on vector computers, especially on 'long vector' machines, is described. The algorithm is parameterized to enable it to adapt to the properties of various CPUs and APs. The chief technical problem which is discussed is 'vector dependency'; both the statistics of occurrence of dependency in real synthesis data and the technical options for coping with it in real vector machines are treated in detail.

## Table of Contents



[1] The National Radio Astronomy Observatory is operated by Associated Universities, Inc., under contract with the U. S. National Science Foundation.

## 1.0 INTRODUCTION

Present implementations of NRAO's Astronomical Image Processing System (hereafter AIPS) depend critically on Floating Point Systems' AP-120B array processors (now repackaged and renamed as their 5105 and 5205 models). Many of the FPS library functions are used by AIPS (e.g., vector add, vector max/min, FFT, etc.). In addition, two specialized functions which NRAO has microcoded for the AP-120B, "gridding" and "cleaning", are profoundly important in radio synthesis data processing. This paper addresses the problem of programming the gridding operation on vector machines (coding of the CLEAN algorithm for such machines will be the subject of a future paper). Gridding is a time-consuming part of two of the "workhorse" programs in AIPS, UVMAP and MX (MX actually grids and re-grids data in an iterative loop). The gridding operation must be coded carefully for any new vector processor if NRAO is to produce an effective implementation of AIPS on that processor. We begin our discussion of vector gridding by giving some historical background.

The AIPS programmers began trying to find new array processors for AIPS during 1983. There were three motivations for this: (1) we wanted to find cheaper APs, (2) we wanted APs for host CPUs which FPS declined to support, and (3) we wanted to obtain higher performance if possible. Three APs were studied extensively (Analogic AP-500, Numerix MARS-432, Masscomp AP-501). Our conclusion was that the best approach to the new APs would be to implement new versions of the FPS library routines which would have the same names, arguments, and functionality, but which would be optimized for the peculiarities of each AP (see Cotton and Wells 1983).

In January 1984 the AIPS programmers submitted a formal proposal inside NRAO for funds and authorization to obtain a Masscomp AP-501 array processor and other image processing peripherals to be installed on a Masscomp MC-500 computer which was to be purchased for another project. This proposal led to an elaborate evaluation (January-March 1984) of ALL aspects of the Masscomp MC-500 and the AP-501 in an effort to estimate the probable performance and price-performance ratio advantage of this new technology (supermicros plus APs). It soon became obvious that the AP-501 has an architecture which is quite different from that of the FPS AP-120B. This made it very difficult to predict the probable system performance by analogy to the 120B on a VAX. We were forced to assume that, at best, the system would only work about half as well as a 120B. Because the system price was about half that of a VAX-750/FPS-5105 combination, and because the project goal was to obtain a price-performance advantage of TWO, it followed that the system would need two AP-501s. Unfortunately, in practice, the MC-500 could not be configured with two AP-501s. Reluctantly, the proposal was withdrwawn in March 1984.

The uncertainties about the AP-501 performance did not involve the standard vector operators such as addition, finding maxima, and FFTs. We were, and still are, convinced that the MC-500/AP-501 combination is a formidable competitor for a VAX-750/FPS-5105 combination in such applications. Rather, the problem lay in the unique, critical

algorithms of aperture synthesis: gridding and cleaning. We chose gridding as our architectural test problem (we think cleaning is probably not really as difficult as it appears to be on first examination), and we defined the goal to be: "make subroutine QGRD4 [2] run well in the vector machine". It was obvious in March 1984 that the conventional QGRD4 algorithm was poorly suited to the AP-501, and there was no assurance at that time that there was any better approach to the gridding problem. This was the chief technical reason why the Masscomp supermicro-plus-AP proposal was abandoned. During the subsequent months we studied how to modify QGRD4 in order to improve vectorization (our knowledge of the AP-501 architecture, gained on non-disclosure terms, was very important in this process).

Meanwhile, in April 1984 we began to study the CDC Cyber 205 supercomputer and the Star ST-100 "super" array processor as potential hosts for AIPS, and we encountered similar problems with QGRD4! More recently, the Pennsylvania State PSAIPS group has begun implementing AIPS for the Sky "Warrior" AP, which is architecturally similar to the AP-501. All of these machines can be categorized as "long vector" machines. By this we mean that they operate most efficiently on vectors of length 100 or more. The 120B runs efficiently on vectors of length less than 5. The Cray and Convex machines are an intermediate case: they are efficient on vectors of length 10-20 (but their efficiency is significantly improved for longer vectors). So, the challenge became: "design a generic version of QGRD4 for long vector architectures". During the period April-October 1984 we evolved an approach to this problem, and we present it in this paper.

We conjecture that this new "vectorized" QGRD4 algorithm may turn out to be the best approach for all vector machines which need vectors longer than about 7 to be efficient, from APs (Masscomp [3] and Sky) thru super-APs (Star) and mini-supercomputers (Convex) to supercomputers (CDC and Cray). During our evaluations of this whole range of vector hardware, we have become convinced that QGRD4 is a very effective test case: it tends to expose and accentuate differences between the architectures. This is because all vector systems perform operations like the FFT efficiently, but performance on real total applications tends to be determined by the fraction of each application which cannot be vectorized on the various systems. QGRD4 is a rather good general indicator of the capabilities of an architecture for

---

[2] Fortran emulations of the AIPS AP-120B microcode are available in the AIPS "pseudo-AP" library. These Fortran subroutines are much easier to analyze and port than the 120B microcode. During 1984 the gridding subroutine we analyzed was called APGRD4. In the 15JAN85 release of AIPS, it has been renamed as QGRD4. The other AP-emulation subroutines have also been given names that start with "Q"; the purpose is to formally establish a virtual vector hardware interface in AIPS. As a part of this change, the integer subscripts being passed as arguments are now 32-bit integers rather than 16-bit.

[3] Obviously, if we had known in March 1984 what we now know, the Masscomp project decision might have been different.

handling the "difficult" fraction of all problems, and it is an especially good indicator for our synthesis mapping application.

## 2.0 "SCALAR" GRIDDING (SUBROUTINE QGRD4)

In this section we present and then discuss the machine-independent Fortran subroutine QGRD4.FOR from the "Q-subroutine" library of AIPS (15JAN85 version). The data declaration INCLUDEs have been expanded in the listing shown below. The include file ZVND.INC before the inner DO-loop is contains the "no-dependency" compiler directive for vectorizing compilers ("CDIR$ IVDEP" for the CFT compiler on Crays, "C$DIR NO_RECURRENCE" on Convex). Without such a declaration these compilers will almost certainly refuse to vectorize this DO-loop. Array APCORE() is dimensioned elsewhere (in subroutine QINIT.FOR) to be 65536 R*4 cells. This dimension could be changed for machines with large memories.

```
      SUBROUTINE QGRD4 (UV, VIS, WT, GRID, CONX, CONY, NO2, M, LROW,
     *   INC, NVIS)
C-----------------------------------------------------------------------
C   Pseudo-AP version
C   Convolves visibility data onto a grid.
C   A single channel is gridded at a time.
C   It assumes that NO points lie within one half the
C   convolving function support size of the outside edge.
C    Inputs:
C     UV     I*4   Location of (u,v) values in cells.
C     VIS    I*4   Location of (complex) visibilities.
C     WT     I*4   Weight for data. Assumes any tapering
C                  has already been done.
C     GRID   I*4   Base address of gridded data.
C                  Order assumed to be the following
C                  for each of the M rows:
C                   1) 2 * LROW visibilities
C     CONX   I*4   Base address of X convolving fn.
C     CONY   I*4   Base address of Y convolving fn.
C     NO2    I*4   INT( (# cells used on a row) / 2 )
C     M      I*4   Number of rows kept in the AP.
C     LROW   I*4   Length of a row ( max. X).
C     INC    I*4   Increment for UV, VIS and WT
C     NVIS   I*4   Number of visibilities to grid.
C      In the above, X refers to rows and y to columns
C      in the gridded data, NOT on the sky.  The total
C      numbers of rows and cells used on a row should
C      be odd.
C     All AP memory I/O values are assumed floating.
C     It is assumed that all values of v correspond to row M/2.
C-----------------------------------------------------------------------
```

```
      INTEGER*4 UV, VIS, WT, GRID, CONX,  CONY, NO2, M, LROW, INC,
     *   NVIS,  N, INCR, HAF, IX, IY
      INTEGER*4 JUV, JVIS, JWT, JGRID, JCONX, JCONY, JCX, JCY,
     *   JG, JJCX, JJLOOP, IFIX, IRND
      REAL*4    AIM, RE, RRE, AAIM, X, XX, XWT, Y, RHALF, SIGN
      INCLUDE 'INCS:DAPC.INC'
C                                                          Include DAPC
      REAL*4 APCORE(1), RWORK(4096)
      INTEGER*4 APCORI(1), IWORK(4096), SPAD(16)
      COMPLEX CWORK(2048)
C                                                          End DAPC
      INCLUDE 'INCS:CAPC.INC'
C                                                          Include CAPC
      COMMON /APFAKE/RWORK, APCORE
      COMMON /SPF/ SPAD
C                                                          End CAPC
      INCLUDE 'INCS:EAPC.INC'
C                                                          Include EAPC
      EQUIVALENCE (APCORE, APCORI), (RWORK, IWORK, CWORK)
C                                                          End EAPC
      DATA RHALF /0.5/
C-----------------------------------------------------------------------
      IRND(XX) = IFIX (XX + SIGN (RHALF, XX))
C                                        Convert addresses to 1 rel.
      JUV = UV + 1
      JVIS = VIS + 1
      JWT = WT + 1
      JGRID = GRID + 1
      JCONX = CONX + 1
      JCONY = CONY + 1
      N = NO2 * 2 + 1
      HAF = LROW / 2 - NO2
      INCR = 2 * LROW - 2 * N
C                                        Loop over visibilities.
      DO 300 JJLOOP = 1,NVIS
C                                        Check weight.
         XWT = APCORE(JWT)
         IF (XWT.LE.0.0) GO TO 300
C                                        Determine location.
         X = APCORE(JUV+1)
         Y = APCORE(JUV)
C                                        Deter. conv. fn loc.
         JCX = JCONX + IRND (100. * (IRND (X) - X - 0.5)) + 100
         JCY = JCONY + IRND (100. * (IRND (Y) - Y - 0.5)) + 100
C                                        Determine grid loc.
         JG = JGRID + 2 * (IRND (X) + HAF)
C                                        Save JCX.
         JJCX = JCX
C                                        Get visibility.
         RE = APCORE(JVIS) * XWT
         AIM = APCORE(JVIS+1) * XWT
```

```
C                                                  Gridding loop.
         DO 200 IY = 1,M
            JCX = JJCX
            RRE = RE * APCORE(JCY)
            AAIM = AIM * APCORE(JCY)
      INCLUDE 'INCS:ZVND.INC'
            DO 100 IX = 1,N
C                                                  Sum to grid.
               APCORE(JG) = APCORE(JG) + APCORE(JCX) * RRE
               APCORE(JG+1) = APCORE(JG+1) + APCORE(JCX) * AAIM
C                                                  Update pointers.
               JCX = JCX + 100
               JG = JG + 2
 100           CONTINUE
C                                                  Update pointers.
            JCY = JCY + 100
            JG = JG + INCR
 200        CONTINUE
C                                                 Update for next vis.
         JUV = JUV + INC
         JVIS = JVIS + INC
         JWT = JWT + INC
 300     CONTINUE
C
 999  RETURN
      END
```

## 2.1 Discussion Of Subroutine QGRD4

QGRD4 convolves complex visibility samples onto a regular grid. The grid is required in order to allow the use of the Fast Fourier Transform (FFT) to compute the map. There are three main problems in vectorizing QGRD4 in various types of vector machines: data dependent addressing, short vectors, and vector dependencies.

1. Data Dependent Addressing

   The integer subscripts JCX, JCY, and JG are computed in the outer DO 300 loop from the U-V coordinates of the visibility measurements. These subscripts are then used inside the DO 200 and DO 100 loops to access arrays in APCORE(). Some APs have trouble passing addresses computed in floating point back to their addressing generators. This is generally not a problem in supercomputers.

2. Short Vectors

   The values of variables M and N, which are the loop limits of the DO 200 loop and the DO 100 loop, are generally of order 7 in AIPS applications (i.e., a 7x7 convolution kernel is used). This code executes efficiently in an AP-120B because the pipelines are short (only 2-3 clock cycles) and memory access can be overlapped well

with pipeline operations. Because the Numerix MARS-432 is very similar architecturally, we can confidently predict that it will be at least as effective as the AP-120B. But many other vector machines are not very efficient when processing vectors of length 7. The Cray machines are a notable exception; they will probably perform with at least 30% efficiency on this code on the first try because their vector pipelines have a low startup overhead [4]. But the Cyber 205, a long vector machine, has a vector half length of about 50; its efficiency will be less than 15% on this code. Thus, the goal is now established: we must increase the vector lengths in the innermost loop if QGRD4 is to run efficiently in long vector machines.

3. Vector Dependencies

A pipeline processor can only overlap the beginning of processing one vector with the end of processing the previous one if the two vectors do not overlap in memory. If this rule is violated, improper computations may result. In QGRD4, dependency arises because we are co-adding visibility data to the grid and because successive visibilities may need to co-add to the same cells. In practice, the code as presented above has little trouble with dependencies; we must merely assure that the last store operation on the last cycle of the outer loop is completed BEFORE the beginning of the first load operation of the first cycle of the outer loop for the next visibility.

The real problem comes when we try to increase the vector length by permuting the order of the DO-loops. In particular, the outer loop which processes visibilities can be arbitrarily long, but the dependency is in this loop, and we will move the dependency into our innermost loop (the worst place!) if we bring this loop to the inside. It is precisely this problem which motivates this entire paper; we will see that vector instructions can be used to detect the presence of dependencies in order to avoid producing erroneous results.

## 2.2 Related "Q-Subroutines"

Subroutine QGRD4 is called by routine Q1GRD; these are the gridding routines for AIPS task MX. Other gridding tasks have slightly different requirements, and hence they have different Q-routines. Routines QGRD1, QGRD2, and QGRD3 each process only one visibility at a

---

[4] Actually, the "half length" of the Cray pipes is about 7; the inner loop efficiency should be 50%. We choose to apply a systems analyst's rule-of-thumb of 60%, and get 30% as the estimated performance. The "half length" is the number of clock ticks it takes to start the pipe, i.e., the vector length for which the pipe runs at 50% efficiency (Hockney & Jesshope 1981).

time; therefore, they have only two DO-loops inside them. The third DO-loop is in another Q-routine which calls them. In particular, routine QGRID calls QGRD1 and QGRD2, and QGRIDA calls QGRD3 and QGRD2. Inspection of the code shows that they are all very similar, and that a solution to the "QGRD4 problem" is effectively a solution for the other routines as well (presumably QGRID and QGRIDA would have to incorporate the three subordinate routines so that the triple DO-loops could be permuted).

## 3.0 VECTORIZED GRIDDING

Two ideas immediately come to mind: "unroll" the inner loop so that the length becomes 49 rather than 7, or permute the DO-loops so that the outer one over visibilities comes to the inside. Unrolling has the problems that the length is still only 49 (not really enough for a Cyber 205), and that the convolution support size is hard-coded. In addition, we still have the problems that the cells of the kernel are not a constant-stride vector (i.e., we need a gather-scatter operation), and that we cannot overlap the load of one vector with the store of another. We conclude that we would like to permute the DO-loops and eliminate the dependency problem in the innermost loop.

The dependency problem is that several visibility points may contribute to a given cell and, with the probable use of gather/scatter operations, all but the last contribution will be lost. It should be noted that if the loop over the visibility is the inner loop, then this conflict only occurs if there are several visibilities centered on the same grid cell, and that this conflict will then occur over the whole convolving function support size.

With this latter point in mind, one solution is to partition the input visibility vector into sections which have no dependencies. Then as much work as possible is done on the full vector but updating the stored grid is done by partition. This will clearly work best in the case of completely unsorted data, i.e., the full grid can be kept in memory, but will likely give acceptable performance with the current practice (a "scrolling" buffer). The following sections outline this approach for the full gridding case. It should work very well in cases where there are no dependencies and at least do the right thing in the other extreme where all data are in the same cell.

Please note that the discussion in this paper applies only to uni-processor pipelined vector machines. It is likely that other strategies will be more appropriate for multi-CPU machines like the Denelcor HEP. We think that the case of a dual or quad CPU (e.g., the Cray X-MP) is much easier, but we have not considered it in detail.

## 3.1 Overview Of The New Algorithm

Note: most of the operations given below are vector operations over vectors of length equal to the number of visibilities.

I. The [complex] visibilities are multiplied by the weights.

II. Partitioning the problem.

1. Compute the central cell offset for each visibility. CENCEL = LX * U + V*2
2. Establish partitions (see following sections for alternate methods). Fill arrays ISTART and ISTOP, which are the first and last elements in each partition.
3. Initialize ACX and ACY, which are the addresses in the convolving function lookup tables for each visibility.

III. Loop over Y convolving function support.

1. Compute CY address (increment ACY by 100).
2. Gather CY (convolving function values).
3. Multiply CY times (visibility * weight) and save in temporary vector(s).
4. Loop over X convolving function support.
   a. Compute grid cell numbers (CENCEL + scalar).
   b. Compute CX address (increment ACX by 100).
   c. Gather CX (convolving function values).
   d. Multiply CX times (CY * visibility * weight).
   e. Loop over the partitions (a test could be done to check for short partitions and do these cases in scalar mode).
      i. Gather old [complex] grid values (CENCEL).
      ii. Sum new contributions into grid values.
      iii. Scatter new grid values.

## 3.2 The Vectorized QGRD4 In Fortran

The following example gives a hypothetical version of QGRD4 using calls to the vector subroutines in the AIPS "Q-routine" library. A description of all of these routines is given in section 4.0, entitled "Vector Operators". Implementation details for various vector systems are given in Section 5.0, entitled "Real Machines".

Note that GATHER/SCATTER routines have been used. Note also that the call to QGRD4 is assumed to pass the actual arrays (Fortran-style) rather than subscript pointers to them in AP memory (which is what the QGRD4 in section 2.0 does). The purpose of this example is to show one implementation of the procedure outlined above using vector primitive operators. Probably the code as given here will not execute efficiently on any real computer; in the course of our discussion we will outline how the algorithm can be adapted for use with several real machines, both APs and supercomputers.

The version given here grids unsorted visibilities onto a full grid (i.e., array GRID will usually have a dimension of 2*LROW*LROW). For sorted visibilities and a "scrolling" buffer, GRID will have a dimension of only 2*LROW*M (as in the present QGRD4). This makes only a minor change in the algorithm, as noted below. The default dimensions of CONX() and CONY() are (100,7) in the present implementation (i.e., the convolving functions are tabulated at intervals of 0.01 grid cell).

```
      SUBROUTINE QGRD4 (UV, VIS, WT, GRID, CONX, CONY,
     *   NO2, M, LROW, INC, NVIS)
C-----------------------------------------------------------------------
C     QGRD4 convolves visibility data onto a grid.
C     A single channel is gridded at a time.
C     It assumes that NO points lie within one half the
C     convolving function support size of the outside edge.
C     This example assumes that the full grid is available in memory.
C   Inputs:
C         UV        array of (u,v) values in cells.
C         VIS       array of (complex) visibilities.
C         WT        weights for data. Assumes any tapering
C                         has already been done.
C         GRID      data grid.
C         CONX      X convolving function lookup table.
C         CONY      Y convolving function lookup table.
C         NO2       INT( (# cells used on a row) / 2 )
C         M         convolution function support size in Y-direction
C                       In the above, X refers to rows and Y to columns
C                       in the gridded data, NOT on the sky. The total
C                       numbers of rows and cells used on a row should
C                       be odd.
C         LROW      length of a row ( max. X, 2*#-complex)
C         INC       increment for UV, VIS and WT.
C         NVIS      number of visibility points to grid.
C                       This example can handle up to 1000.
```

```
C-----------------------------------------------------------------------
      INTEGER NO2, M, LROW, INC, NVIS
      REAL    UV(INC,NVIS), VIS(INC,NVIS), WT(INC,NVIS), GRID(1),
     *   CONX(1), CONY(1)
C                                        Declare temporary variables:
C                                        NOTE: In a real implementation
C                                        Many of the arrays could be
C                                        EQUIVALENCEd.
      INTEGER I, J, K, N1, NPART, I1, NCHK, NX,
     *   ISTART(1000), ISTOP(1000), IX(1000), IY(1000), TIX(1000)
      REAL    TSCALR, XMAX, F2,
     *   VREAL(1000), VIMAG(1000), ACX(1000), ACY(1000), CENCEL(1000),
     *   CFN(1000), TGRID(1000), TVSRE(1000), TVSIM(1000), TEMP(1000),
     *   X(1000), Y(1000), VSRE(1000), VSIM(1000)
      DATA N1 /1/,   F2 /2.0/
C-----------------------------------------------------------------------
C                                        Multiply visibilities by
C                                        weights. Real part first.
      CALL QVMUL (VIS(1,1), INC, WT, INC, VREAL, N1, NVIS)
C                                        Imaginary part.
      CALL QVMUL (VIS(2,1), INC, WT, INC, VIMAG, N1, NVIS)
C                                        Compute central cell numbers.
C                                        NOTE: the details of computing
C                                        central cell numbers depends
C                                        on whether the full or partial
C                                        grid is kept in memory. Round
C                                        cell numbers in scalar mode.
      DO 10 I = 1,NVIS
        IX(I) = UV(2,I) + SIGN (0.5, UV(2,I))                          [5]
        IY(I) = UV(1,I) + SIGN (0.5, UV(1,I))
 10     CONTINUE
C                                        Float
      CALL QVFLT (IX, N1, X, N1, NVIS)
      CALL QVFLT (IY, N1, Y, N1, NVIS)
C                                        Convert to cell number.
      CALL QVSMUL (X, N1, F2, TEMP, N1, NVIS)
      CALL QVSMA (Y, N1, LROW, TEMP, N1, CENCEL, N1, NVIS)
C-----------------------------------------------------------------------
C   Insert code here which will establish partitions, fill ISTART,
C   ISTOP, and set NPART. See later sections for alternate methods.
C-----------------------------------------------------------------------
```

[5] This rounding operation can also be vectorized. The algorithm is: compute a Boolean truth vector on the relation [X().GE.0.0] (in a Cray or Convex this is done in the Vector Mask register, in a 205 it is a bit vector), set up temporary vectors of all +0.5 and all -0.5, merge them into another temporary conditioned on the Boolean vector, add X(), and fix it into IX(). This discussion also applies to the DO 100 loop below. Also note that the Masscomp pipes can round while fixing, all in one operation.

```
C                                          Offset CENCEL to first cell
C                                          to use.
      TSCALR = - (NO2 * LROW + (M/2) * 2)
      CALL QVSADD (CENCEL, N1, TSCALR, CENCEL, N1, NVIS)
C                                          Form initial convolving
C                                          function addresses.
      CALL QVSUB (X, N1, UV(2,1), INC, X, N1, NVIS)
      CALL QVSUB (Y, N1, UV(1,1), INC, Y, N1, NVIS)
      TSCALR = -0.5
      CALL QVSADD (X, N1, TSCALR, X, N1, NVIS)                    [6]
      CALL QVSADD (Y, N1, TSCALR, Y, N1, NVIS)
C                                          Multiply by 100.
      TSCALR = 100.0
      CALL QVSMUL (X, N1, TSCALR, X, N1, NVIS)
      CALL QVSMUL (Y, N1, TSCALR, Y, N1, NVIS)
C                                          Round X, Y.
      DO 100 I = 1,NVIS
         TIX(I) = X(I) + SIGN(0.5, X(I))
         IY(I) = Y(I) + SIGN(0.5, Y(I))
 100     CONTINUE
C                                          Loop over Y convolving
C                                          function support:
      DO 600 I = 1,M
C                                          Compute CY address (add 100)
C                                          first restore IX:
         CALL QVMOV (TIX, N1, IX, N1, NVIS)
         TSCALR = 100
         CALL QVSADD (IY, N1, TSCALR, IY, N1, NVIS)
C                                          Gather CY function values.
         CALL GATHER (CONY, IY, CFN, N1, NVIS)
C                                          Multiply CY times vis and save
         CALL QVMUL (VIS(1,1), INC, CFN, N1, TVSRE, N1, NVIS)
         CALL QVMUL (VIS(2,1), INC, CFN, N1, TVSIM, N1, NVIS)
C                                          Loop over X convolving support:
         NX = NO2 * 2 + 1
         DO  500 J = 1,NX
C                                          Compute CX address (add 100).
            TSCALR = 100
            CALL QVSADD (IX, N1, TSCALR, IX, N1, NVIS)
C                                          Gather CX function values.
            CALL GATHER (CONX, IX, CFN, N1, NVIS)
C                                          Multiply CX.
            CALL QVMUL (TVSRE, N1, CFN, N1, VSRE, N1, NVIS)
            CALL QVMUL (TVSIM, N1, CFN, N1, VSIM, N1, NVIS)
```

---

[6] On Cray and Convex computers several successive vector operations of the same length (NVIS in this case), operating on common vectors, can be performed in one loop which "strip mines" by the vector register length, thus "chaining" the pipes to achieve greater speed. The cache memories of the Star, Masscomp and Sky APs can be used in a similar fashion. The goal is to increase the ratio of pipe cycles to memory cycles.

```
C                                       Loop over partition.
            DO 400 K = 1,NPART
               I1 = ISTART(K)
               NCHK = ISTOP(K) - I1 + 1
C                                       NOTE: could trap short
C                                       partitions here and do in
C                                       scalar mode.
C                                       Gather old real part:
               CALL GATHER (GRID, CENCEL(I1), TGRID, N1, NCHK)
C                                       Sum reals.
               CALL QVADD (TGRID, N1, VSRE(I1), N1, TGRID, N1, NCHK)
C                                       Scatter new grid values back.
               CALL SCATTER (GRID, CENCEL(I1), TGRID, N1, NCHK)
C                                       Gather old imaginary part.
               CALL GATHER (GRID(2), CENCEL(I1), TGRID, N1, NCHK)
C                                       Sum imaginaries.
               CALL QVADD (TGRID, N1, VSIM(I1), N1, TGRID, N1, NCHK)
C                                       Scatter new grid values back.
               CALL SCATTER (GRID(2), CENCEL(I1), TGRID, N1, NCHK)
 400           CONTINUE
C                                       Update CENCEL to next cell.
            TSCALR = 2
            CALL QVSADD (CENCEL, N1, TSCALR, CENCEL, N1, NVIS)
 500        CONTINUE
C                                       Update CENCEL to start of next
C                                       row.
         TSCALR = LROW - M * 2 - 2
         CALL QVSADD (CENCEL, N1, TSCALR, CENCEL, N1, NVIS)
 600     CONTINUE
 999  RETURN
C-------------------------------------------------------------------------
      END
```

## 3.3 The "Natural" Partition Method

The simplest method of establishing partitions is to divide up the data, without rearrangement, into partitions with no dependencies. Since each element in a partition must be compared with every other member, the cost of this method is proportional to the square of the partition length. On some machines (such as a Cray) there are utility routines which search for the first occurrence in a vector of a value; if such is available then it should be used. The following gives an example of a vectorized method using functions available on an FPS array processor (and assumed to be universally available). The notation is that used in section 3.2. The REGSIZ parameter used below shows an example of how algorithms can be adapted to architectural features. In this case, REGSIZ should be 64 for Crays, 128 for the Convex C-1,and 65535 for the CDC Cyber 205. Lengths shorter than the nominal REGSIZ may be appropriate to balance the N-square search cost against the vector pipeline startup cost.

```
C                                          Partitions need not be longer
C                                          than vector register length:
      PARAMETER (REGSIZ = rrrr)
C-------------------------------------------------------------------------
C                                          Use CENCEL to construct
C                                          partition table.
C                                          Loop over elements:
      NPART = 1
      ISTART(1) = 1
      ISTOP(1) = MIN (ISTART + (REGSIZ - 1), NVIS)
      DO 20 I = 2, NVIS
C                                          Check for short (1) partitions.
         IF (CENCEL(I).EQ.CENCEL(I-1)) GO TO 10
C                                          Subtract CENCEL(I) from rest of
C                                          array.
         I1 = ISTART(NPART)
         NCHK = I - I1
         TSCALR = -CENCEL(I)
         CALL QVSADD (CENCEL(I1), N1, TSCALR, TEMP, N1, NCHK)
C                                          Look for minimum value.
         CALL QMINMG (TEMP, N1, XMAX, NCHK)
C                                          If max.abs.value(XMAX) > 0.5
C                                          then no prior use of cell.
         IF ((XMAX.GT.0.5) .AND. (NCHK.LE.REGSIZ)) GO TO 20
C                                          Prior use, start new partition.
 10         ISTOP(NPART) = I-1
            NPART = NPART + 1
            ISTART(NPART) = I
            ISTOP(NPART) = NVIS
 20      CONTINUE
```

### 3.3.1 Distribution Of "Natural" Partition Lengths

The viability of the "Natural" partitioning scheme depends, at some level, on the lengths of the partitions encountered in typical data sets. The issue is somewhat complicated by the fact that short partition lengths make determining the partition boundaries easier but updating the grid more difficult.

In order to determine the partition lengths to be expected with VLA observations two sets of data were analyzed. The first set contained 6000 visibilities which consisted of several scans; the second, and larger, set contained 121,446 visibilities. Two cases were considered for each data set: (1) the current case in which one row at a time is gridded and the data is completely sorted, and (2) the case in which the entire matrix is gridded at once and the data are in time-baseline order. In all cases the size of the grid is 512x512 with normal sampling.

The data were analyzed to determine the distribution of partition lengths which occurred. These results are summarized in Table 1 which gives the average partition length and the approximate maximum

partition length.

Table 1

Distribution of Partition Lengths from Sample Data

| Case | Average Length | Maximum Length |
|---|---|---|
| 6K vis, sorted | 2.2 | 9 |
| 121K vis, sorted | 7.2 | 35 |
| 6K vis, unsorted | 44 | 150 |
| 121K vis, unsorted | 63 | 150 |

A more detailed examination of the distribution of partition lengths shows that roughly 50% of the data is in partitions of equal or greater length than the average length partition. The conclusion that can be derived from this table is that the partition lengths expected for sorted data are sufficiently short that the updating of the grid will not make good use of the vector hardware. For unsorted data, the resulting partition sizes will make acceptable length vector operations on a short vector machine such as a Cray but not on long vector machines such as a Cyber 205.

The above suggests that this partitioning scheme is useful only in the case of gridding unsorted data (i.e., much more of the grid than the current row-at-a-time method). The case of sorted data may be best dealt with by the Scatter/Compress method described in the following sections. The typical partition length will increase with the size of the image being made.

## 3.4 The "Scatter/Compress" Method

An alternate scheme for partitioning the input data is to rearrange the data to maximize the lengths of the partitions. This is done by scattering the indices of the data points onto a work array the size of the portion of the grid being worked on, and then compressing this vector to obtain a list of indices. The reason this works is that when multiple visibilities use the same cell, only the indices of the last will be kept; thus, this scheme obtains the maximum length list of visibilities which has no dependencies. These visibilities are removed from the list and the process is iterated until the list of data remaining is exhausted. This method makes extensive use of GATHER/SCATTER and COMPRESS operations, and so we refer to it as the "Scatter/Compress" scheme for partitioning the data [7]. The algorithm creates a large number of temporary vector variables which are

[7] When we first thought of this idea in April 1984, we called it the "anti-sorting" algorithm, because sorting visibilities maximizes the occurence of vector dependency in gridding, whereas this technique minimizes it.

described below. Consult section 4.0 for a description of the vector operators which are used in this algorithm. Note that the algorithm as given here only makes sense when gridding sorted data, in which case it is working on only a few rows (usually one).

Vectors:

| | |
|---|---|
| RAMP | An array initialized to the sequence 1,2,3,... |
| INDICES | An array containing the original indices of the input visibilities before they are rearranged. |
| INDEX | An array which will contain the indices of the input visibilities rearranged in sorted partitions. |
| CENCEL | An array containing the center cell grid addresses of the visibilities. |
| WROW | A work array the length of working section of grid |
| WROW2 | Another work array the length of working section |
| MASK | A mask array indicating members to be compressed from the vector. |
| GADR | A work vector of length equal to the number of data points to contain the grid addresses. |

Arrays:

| | |
|---|---|
| ISTART | Array of pointers to the first members of partitions. |
| ISTOP | Array of pointers to the last members of partitions. |

Scalars:

| | |
|---|---|
| N | The number of visibilities in the input vector. |
| LROW | The length of a row. |
| NPART | Number of partitions |
| VLEN | Length of list of cells left |
| VLENP | Length of list of cells left (temporary value) |

```
C-----------------------------------------------------------------------
C                                          Set up partitions using the
C                                          "Scatter/Compress" method.
      NPART = 0
      ISTART(1) = 1
      CALL QVMOV (CENCEL, N1, GADR, N1, N)
      VLEN = N
C                                          Clear row work vector.
      CALL QVFILL (0, WROW, LROW)
C                                          Following loop vectorizes on
C                                          many machines:
      DO 15 I = 1, N
 15      RAMP(I) = I
      CALL QVMOV (RAMP, N1, INDICES, N1, N)
```

```
C                                       Loop until list exhausted:
 20   CONTINUE
C                                       Scatter the ramp
C                                       to row work vector:
        CALL SCATTER (RAMP, GADR, WROW, VLEN)
C                                       WROW contains indices of the
C                                       last hits in each cell.
C                                       Make mask of active cells:
        CALL MAKMASK (WROW, MASK, LROW)
C                                       Compress them to next partition:
        NPART = NPART + 1
        IF (NPART.GT.1) ISTART(NPART) = ISTOP(NPART-1) + 1
C                                       first compress cell indices:
        CALL COMPRESS (WROW, MASK, WROW2, LROW)
C                                       Get length of partition:
        LEN = POPCNT (MASK, LROW)
C                                       and gather the partition:
        CALL GATHER (INDICES, WROW2, INDEX(ISTART(NPART)), LEN)
        ISTOP(NPART) = ISTART(NPART) + LEN - 1
        VLENP = VLEN
        VLEN = VLEN - LEN
C                                       Finished?
        IF (VLEN.LE.0) GO TO 25
C                                       Clear work vector again:
        CALL QVFILL (0, WROW, LROW)
C                                       Now remove processed indices.
C                                       Zero indices in partition:
        CALL SCATTER (WROW, WROW2, INDICES, LEN)
C                                       Mask indices not in partition:
        CALL MAKMASK (INDICES, MASK, VLENP)
C                                       Compress indices in partition:
        CALL COMPRESS (INDICES, MASK, INDICES, VLENP)
C                                       Compress cell addresses:
        CALL COMPRESS (GADR, MASK, GADR, VLENP)
        GO TO 20
C                                       End of loop:
 25   CONTINUE
C                                       INDEX now contains indices of
C                                       the input data in the new
C                                       partition order. Rearrange
C                                       the addresses and data:
       CALL GATHER (CENCEL, INDEX, GADR, N)
       CALL QVMOV (GADR, N1, CENCEL, N1, N)
C                                       Real part.
       CALL GATHER (VREAL, INDEX, TVSRE, N)
       CALL QVMOV (TVSRE, N1, VREAL, N1, N)
C                                       Imaginary part.
       CALL GATHER (VIMAG, INDEX, TVSIM, N)
       CALL QVMOV (TVSIM, N1, VIMAG, N1, N)
```

Note: this scheme will sort data by partition which should enhance the performance of the entire gridding process on machines with virtual and/or cache memory.

### 3.4.1 Cell Hit Statistics

The amount of work the Scatter/Compress method requires depends on the frequency of multiple hits on a single cell and, in particular, on the maximum number of hits per cell in a given row (assuming row-at-a-time gridding). The statistics shown in Table 2 below have been derived for the same data sets and in a manner similar to that used to obtain the distribution of "Natural" partition lengths described above. The table gives the average number of hits per cell, an approximate value of the maximum number of hits per row which is exceeded 50% of the time and the maximum hits in any cell.

Table 2

Cell Hit Statistics

| Case | Avg. hit per cell | Prob. row max. | Max hit/cell |
|---|---|---|---|
| 6K vis sorted | 5.7 | 9 | 30 |
| 121K vis sorted | 15.6 | 20 | 350 |

The values in Table 2 indicate that the typical number of partitions per row is of the order of a few tens. This number will decrease for map sizes larger than 512x512 because the cells in uv space become smaller. If the number of visibilities being processed at a time (or the number of visibilities on a typical row) is of the order of a few thousand, then the typical partition length should be on the order of a hundred. The typical partition length is proportional to the size of the grid.

It should be noted that this scheme involves operations on a work vector the length of the portion of the grid currently being accumulated. This is relatively efficient for sorted data being gridded onto a single row at a time because the length of the work vector is fairly short and will have a higher density of data points. In the other extreme, the full grid accumulation of unsorted data, the work vector will be exceedingly long and sparsely populated, and this method of establishing partitions may become unattractive.

## 4.0 VECTOR OPERATORS

This section documents all of the vector operators which were used in the previous sections. In the first group, we describe the functions which have the same names, functionality, and (nearly) the same call arguments as those in the AIPS Q-routine library. (NOTE: In any actual implementation of the vectorized gridding algorithm, ALL details should be checked against Chap. 11 of "Going AIPS".) The operators are grouped into classes and a formula is given for the last one in each class. Note that we present the operators as accepting the vectors themselves as arguments (rather than their addresses), and as using zero-based subscripting in order to simplify the notation. Another simplification is that the vector operators are regarded as being capable of handling either integer or floating point data (for example, operator QVSADD is used to increment integer index vectors in a number of places in Sections 3.2, 3.3, and 3.4).

Unary operators:

```
      QVMOV (A, IA, B, IB, N)        [vector copy operation]
      QVFIX (A, IA, BI, IBI, N)
      QVFLT (AI, IAI, B, IB, N)
            B(m*IB) = (FLOAT (AI(m*IAI))) for m = 0 to N-1
```

Vector-Vector operators:

```
      QVADD (A, IA, B, IB, C, IC, N)
      QVSUB (A, IA, B, IB, C, IC, N)
      QVMUL (A, IA, B, IB, C, IC, N)
            C(m*IC) = (A(m*IA) * B(m*IB)) for m = 0 to N-1
```

Vector-Scalar operators:

```
      QVSADD (A, IA, S, B, IB, N)
      QVSMUL (A, IA, S, B, IB, N)
            B(m*IB) = (A(m*IA) * S) for m = 0 to N-1
```

"Linked-Triad" operators:

```
      QVSMA  (A, IA, B, C, IC, D, ID, N)
            D(m*ID) = ((A(m*IA) * B) + C(m*IC)) for m = 0 to N-1
```

Special unary operators:

```
      QVFILL (A, B, IB, N)
            B(m*IB) = (A) for m = 0 to N-1
      QMINMG (A, IA, B, N)
            B = (MIN (ABS (A(m*IA)), B)) for m = 0 to N-1
```

Boolean "Mask" operators:

```
     MAKMASK (A, MASK, LEN)  Create a mask vector.
            MASK(m) = (A(m) .NE. 0.0) for m = 0 to LEN-1

     POPCNT (MASK, LEN) Counts number of TRUEs in MASK.
            POPCNT = 0
            for m = 0 to LEN-1
               If (MASK(m) = TRUE) then POPCNT = POPCNT + 1
```

```
COMPRESS (A, MASK, B, N) Vector compress.        [8]
       n = 0
       for m = 0 to N-1
          If MASK(m) = TRUE then B(n) = A(m); n = n + 1
```

The Gather and Scatter Operators (Indirect Addressing):

```
SCATTER (A, B, C, N) Vector scatter.
       C(B(m)) = (A(m)) for m = 0 to N-1

GATHER (A, B, C, N) Vector gather.
       C(m) = (A(B(m))) for m = 0 to N-1
```

Note: a complete vector hardware system must also implement the EXPAND operator (inverse of COMPRESS), and will also need to have several forms of Vector Merge operations.

---

[8] The compress definition presented here is the one used by the Cyber 205 and the Convex C-1. Three of the real machines discussed in the next section (Cray, Masscomp, and Sky) use a different approach. The comparison operator generates an index vector rather than a bit vector, and then the compressed vector can be produced by a gather operation. In addition, the comparison returns the length of the index vector (see notes [11], [13], and [14] in Table 3). Remund and Taggart (pp. 402-4 in Kuck, et.al. (1977)) have argued that this technique is more efficient for machines that lack bit-vector hardware (a CDC 7600 in their case). With this approach, the eight statements following statement 20 in the Scatter/Compress method would be transformed from:

```
      CALL SCATTER (RAMP, GADR, WROW, VLEN)
      CALL MAKMASK (WROW, MASK, LROW)
      NPART = NPART + 1
      IF (NPART.GT.1) ISTART(NPART) = ISTOP(NPART-1) + 1
      CALL COMPRESS (WROW, MASK, WROW2, LROW)
      LEN = POPCNT (MASK, LROW)
      CALL GATHER (INDICES, WROW2, INDEX(ISTART(NPART)), LEN)
      ISTOP(NPART) = ISTART(NPART) + LEN - 1
```

to

```
      CALL SCATTER (VLEN, WROW, GADR, RAMP)
      CALL WHENNE (LROW, WROW, N1, 0, MASK, LEN)
      NPART = NPART + 1
      IF (NPART.GT.1) ISTART(NPART) = ISTOP(NPART-1) + 1
      CALL GATHER (LEN, WROW2, WROW, MASK)
      CALL GATHER (LEN, INDEX(ISTART(NPART)), INDICES, WROW2)
      ISTOP(NPART) = ISTART(NPART) + LEN - 1
```

for a Cray. Note that the order of the arguments is different for the Cray library and that MASK is a vector of indices in this case.

## 5.0 REAL MACHINES

> "Any attempt at a generalized comparison between the CYBER 205 and the Cray-1 is largely irrelevant since the performance of each is critically dependent on the problem being solved and the way it is mapped onto the hardware." [Ibbett (1982), p.163]

In this section we discuss the special features and idiosyncrasies of a number of different systems. In Table 3 we give the names of the library routines which perform the vector operators on the machines. The information in this section was culled from a variety of sources, which are listed in the Bibliography.

### FPS AP-120B (now the model 5105 or 5205)

The 120B is the classic horizontally microcoded array processor. Two pipes clock at 6 MHz for a peak rate of 12 MFlop (but AIPS gets an effective rate of only about 1 MFlop). COMPRESS and SCATTER are not available in the library. There is no question of feasibility, but obviously there is little incentive because this machine has no need for our vectorized QGRD4 algorithm. Price about $55K. Floating Point Systems, Inc., Beaverton, OR 97005.

### Numerix MARS-432

The MARS-432 is a new horizontally microcoded AP. It was deliberately designed to be quite similar to the 120B, but to improve upon it in both performance and in ease of programming. Three pipes clock at 10 MHz for a peak rate of 30 MFlops. In general, anything you can do with a 120B can also be done with a 432, and in about the same way. (This means that the 432 doesn't need our new vectorized QGRD4 algorithm.) The names, arguments, and functionality of the subroutine library are essentially IDENTICAL to FPS (therefore, we do not present them in Table 3). Numerix has an optimizing Fortran compiler. Price about $125K. Numerix Corporation, 320 Needham Street, Newton, MA 02161, (617) 964-2500.

### Cray-1 and Cray X-MP

The Crays are the classic vector register machines. They are able to overlap scalar operations with vector pipe operations. The pipes are able to "chain" together for increased speed. Cray-1s are often able to hit 120 or more MFLops in extended bursts, and can sustain 50+ MFlops (even more in the X-MP). The vector register load supports constant stride. Existing Cray machines have one notable weakness: gather/scatter and related operators are not supported in the vector hardware. Future Cray machines (e.g., Cray-2 and advanced X-MP models) are expected to have such hardware. The Cray optimizing, vectorizing Fortran compiler has a good reputation; extensive libraries are available. Price about $10**7. Cray Research, Inc., 1440 Northland

Drive, Mendota Heights, MN 55120, (612) 452-6650.

Table 3

Names of the Vector Operators in the Various Systems

| This paper[1] | FPS AP-120B | Cray-1 & X-MP | CDC Cyber205 | Convex C-1 | Star ST-100 | Masscomp AP-501 | Sky Warrior |
|---|---|---|---|---|---|---|---|
| QVMOV | VMOV | [2] | [3] | [2] | [4] | CPFV | VMOV |
| QVFIX | VFIX | " | VINT | " | AVINT | INTIFV | VINT |
| QVFLT | VFLT | " | VFLOAT | " | [5] | CVTIFV | VI2SP[6] |
| QVADD | VADD | " | [3] | " | AVADD | ADDFVV | VADD |
| QVSUB | VSUB | " | " | " | AVSUB | SBFVV | VSUB |
| QVMUL | VMUL | " | " | " | AVMUL | MULFVV | VMUL |
| QVSADD | VSADD | " | " | " | AVSADD | ADDFVS | VSADD |
| QVSMUL | VSMUL | " | " | " | AVSMUL | MULFSV | VSMUL |
| QVSMA | VSMA | " | " | " | AVSMA | MAFSVV | VPIV |
| QVFILL | VFILL | " | " | " | SVFILL[7] | CPFV[8] | VSET |
| QMINMG | MINMGV | ISAMIN | [9] | [10] | [5] | ABMINF | VMNMV |
| MAKMASK | LVEQ | [11] | [12] | " | ALVEQ | CXFSNE | VSCMP |
| POPCNT | SVE | " | Q8SCNT | " | ASVE | [13] | [14] |
| COMPRESS | [15] | " | Q8VCMPRS | " | [5] | " | " |
| SCATTER | " | SCATTER | Q8VSCATR | " | " | MOX[16] | [17] |
| GATHER | VINDEX | GATHER | Q8VGATHR | " | " | EXVF[16] | VINDX |

Notes to Table

[1] The operators listed are only those used in this paper. A complete set would be larger.
[2] Automatic vectorization of Fortran DO-loops handles this.
[3] Automatic vectorization of Fortran DO-loops handles this (translates to routines 'Q8fbrm' and 'Q8fsbrm'); alternatively, use the explicit vector extensions to Fortran.
[4] Synthesize from SMM2C and SMC2M.
[5] Missing! The ST-100 needs this! No question of feasibility.
[6] Use VD2SP if I*4.
[7] Use AVFILL in ACP.
[8] Note use of copy with zero stride on source vector!
[9] Synthesize from VABS followed by Q8SMIN.
[10] No library subroutines yet; easy in assembly language.
[11] Synthesize from WHENNE (POPCNT='nval' output) and GATHER.
[12] Use: MASK(1;LEN)=INPUT(1;LEN).NE.0.0 (it's vector Fortran!).
[13] Synthesize from CXAFSEQ (POPCNT='L1' output) and EXVF.
[14] Synthesize from VMAXG (POPCNT='scalar2' output) and VINDX.
[15] Not available in library; no question of feasibility.
[16] Only in 16 KW AP memory; extension to main memory feasible?
[17] Missing! The Warrior needs this! Probably feasible.

## CDC Cyber-205

The 205 inherits all of the tradition of the STAR-100, the original "long-vector" supercomputer (the "100" was for 100 MFlops). The pipes in this architecture always work memory-to-memory (i.e., no vector registers). A four pipe two million word 205 (one of the largest configurations) can hit a peak rate of 800 MFlops on very long vectors of 32-bit data, but sustained rates on most 205 configurations are generally more like 50 MFlops. Only unit stride vectors are allowed, but gather/scatter is well supported in the hardware, including the gather/scatter of constant stride vectors. Cyber 205s gather or scatter at an average rate of 40 Mwords/sec for random indices. The 205 has an unusually rich and elegant instruction set. Note that the Cyber 205 Fortran compiler supports vector extensions to Fortran (see chap. 11 of the manual). Regarding footnote [5] in section 3.2: the VANINT library subroutine delivers the "nearest whole number" of the elements of a vector. Price about $10**7. Control Data Corporation, P.O. Box 0, Minneapolis, MN 55440. NOTE: CDC has created a subsidiary corporation called ETA Systems, Inc., also located in Minneapolis, which is charged with the mission of building a new computer to be called the "GF-10", which is expected to implement an advanced version of the STAR-100/Cyber 205 architecture. A recent report indicates that CDC will market the machine as the Cyber "250".

## Convex C-1

This is a "mini-supercomputer" which was announced in October 1984. Its architecture is a blend of the best features of the Cyber 205 and the Cray-1. It has vector registers, constant stride loading, gather/scatter and friends, and can do scalar operations while vector operations are in progress. Supported data types are byte, 16 and 32-bit integers, and 32 and 64-bit FP (VAX F&G formats). The architecture is non-byte-swapped (opposite to the VAX). It can chain and/or overlap its pipes (at up to 60 MFlop peak rates). The operating system is 4.2bsd Unix and the optimizing, vectorizing Fortran compiler accepts VMS-style Fortran, including most VMS extensions! The CPU-memory bandwidth is 80 MB/sec and the I/O bandwidth is also 80 MB/sec. Currently the gather/scatter and other "exotic" operators are not supported in a subroutine library, but it should be easy to do anything that is needed for gridding in assembly language. We conjecture that the C-1 can do any vector operation that the Cyber 205 can perform, although it will execute several instructions for one on the 205 to synthesize the operations. Note that the AIPS "sed" script used for installation on Unix systems could easily be augmented so as to expand all calls to the QVxxxx operators into inline DO-loops so that the vectorizing compiler could optimize them. This would eliminate subroutine CALL overhead. Base price about $500K. CONVEX Computer Corporation, 1819 Firman, Suite 151, Richardson, TX 75081, (214) 669-3700.

## Star ST-100

This "super-AP" is popular with the seismic industry. It was recently selected by General Electric for a medical image processing application. Its arithmetic unit (the "ACP") is horizontally microcoded with a 25 MHz clock, and has four pipelines, resulting in a peak performance of 100 MFlops (hence the name). It also has a separate horizontally microcoded "storage move processor" (the SMP) which also clocks at 25 MHz. This is a very powerful unit for moving and converting data between main memory (with a very large address space) and the cache memory (48 KW) which the ACP accesses. A 68000 off to the side sequences the operations of the ACP and SMP, and performs scalar operations for loop control (its role is much like the "vector function chainer" of the 120B). On the whole, the ST-100 looks like the Cray (cache memory equals vector registers), but it has the powerful memory processing abilities of the Cyber 205 (the SMP can do anything a 205 can do). Probably the weaknesses of the macro library just reflect the interests of the first customers of the ST-100, namely the seismic industry. Base price about $250K. Star Technologies, Inc., 1200 Benjamin Franklin Plaza, One S. W. Columbia, Portland, OR 97258, (503) 227-2052.

## MassComp AP-501

Masscomp is one of the very few computer companies which have designed, fabricated, and marketed an integrated AP to attach to their computers. Their AP-501 was announced in the spring of 1984; first deliveries were late in the summer of 1984. The AP-501 is probably just about the minimum AP which is really interesting for use with AIPS. It can overlap DMA with pipeline operations, and it can hide most setup overhead behind the pipes. There are two pipes, an adder and a multiplier, clocking at 5 MHz, for a peak rate of 10 MFlops. It has a peak bandwidth to the host MC-500 of about 5 MB/sec, which is very good. Note that Masscomp offers an FPS compatibility mode for the AP-501 in which the library routines have names VMOV, VFIX, etc., but this only works in the 16 KW AP memory. Price about $8K as an add-on device. MASSCOMP, One Technology Park, Westford, MA 01886, (617) 692-6200.

## Sky WARRIOR

The WARRIOR is quite similar to the Masscomp AP-501, but is intended to be host independent. It was announced in September 1984; first deliveries are expected to occur early in 1985. Its first implementation is for the VME buss (and Versabus), but implementations for other busses are expected soon. It has three pipes, two adders and a multiplier, clocking at 5 MHz, for a peak rate of 15 MFlops. (It uses the same Weitek pipeline chips as the AP-501). The WARRIOR has more chips than the AP-501 and therefore has a somewhat more flexible and powerful architecture, although it is unclear whether the difference is really important for AIPS. The PSAIPS project at Pennsylvania State University expects to implement AIPS on the WARRIOR attached to their CRDS 68000-based system in the spring of 1985. This implementation of the Q-routines should also be useful for other hosts

in the future (VAXen maybe?). Price about $15K. Sky Computers, Inc., Foot of John Street, Lowell, MA 01852, (617) 454-6200.

> "Almost any attempt to predict future developments in the world of computers is doomed to failure, since some technological developments which appear promising fail to materialise, while others perform beyond all reasonable expectations. What is clear is that in the past architectural techniques pioneered on large machines have eventually found their way into small machines, and there is no sign that this trend is abating as microprocessors become more complex and more sophisticated. The manufacturers of high performance systems constantly strive to produce better products, and for students of computer architecture the design of the latest supercomputer will always be important." [Ibbett (1982), p. 163-4]

## 6.0 FUTURE WORK

First, no algorithm designed theoretically, as this one is, can be fully trusted until at least one implementation is functional (several independent implementations on different architectures would be even better). A Cray-1 implementation will be a particularly interesting test case: can this algorithm be configured and tuned in such a way as to beat the performance of the "scalar" QGRD4 of Section 2.0 on a Cray-1? This question is interesting because the Cray-1 (and existing X-MP models) have a short vector half length (i.e., they are close to being scalar machines), and because they lack gather/scatter hardware and must perform these operations with library subroutines. We expect that NRAO programmers will try this experiment on a Cray X-MP within the next few months.

Footnote [6] in Section 3.2 points out that "strip mining" allows improved chaining in vector register machines. To fully implement this idea, the variable REGSIZ (see section 3.3) should be used as the third argument of an outer DO-loop in a number of places in the code and the order of the operations should be permuted a bit. In the neighborhood of footnote [6] this would look something like:

```
      DO ssss IFIRST = 1, NVIS, REGSIZ
         ILAST = MIN (IFIRST + (REGSIZ - 1), NVIS)
         ICOUNT = ILAST - IFIRST + 1
         TSCLR1 = -0.5
         TSCLR2 = -100.0
         CALL QVSADD (X(IFIRST), TSCLR1, X(IFIRST), N1, ICOUNT)
         CALL QVSMUL (X(IFIRST), TSCLR2, X(IFIRST), N1, ICOUNT)
            and similarly for Y...
```

This rearrangement of the code implies that the adder and multiplier pipes can CHAIN in a Cray or Convex machine, which reduces memory

traffic (Cray-1s really don't have enough memory bandwidth; chaining can help a lot). A detailed examination will show that the memory traffic can be reduced even further by making use of more vector registers to hold temporary vectors. In a real implementation on a vector register machine it would probably be necessary to express this concept in assembly language in order to get maximum performance.

Another area for future research is to devise variations on the theme of section 3.4, the Scatter/Compress method. For example, full grid operation on unsorted data might make sense if an in-core "pigeon-hole sort" were done. The pigeon-holes would be regions of the grid small enough that the scatter/compress would be efficient. Partitions gathered from the separate pigeon-holes could be safely concatenated into much larger partitions for the gridding operation.

Finally, the adaptation of all of these ideas to parallel processors (e.g., the dual X-MP, four-CPU Cray-2, ETA GF-10 and Denelcor HEP) is bound to become an important research problem during the next few years.

## 7.0 BIBLIOGRAPHY

### General Background on Vector Hardware:

The general background sources listed below will be useful as tutorial material for programmers who are new to vector programming. For example, a programmer who is assigned to vectorize algorithms for either a Cray or a CDC machine will find that other programmers have worked on similar problems on these machines during the past decade, and that they have written descriptions of their experiences and conclusions. We did not discover most of these references until very late in our work; we probably would have saved some time if we had read them earlier.

Hockney, R.W., and Jesshope, C.R., 1981, "Parallel Computers---Architecture, Programming and Algorithms", Adam Hilger Ltd., Bristol, LOC=QA76.6.H62, ISBN=0-85274-422-6. Good on comparisons of the architectures of the Cray-1, Cyber 205 and AP-120B, especially concerning the vector half-lengths.

Ibbett, R.N. 1982, "The Architecture of High Performance Computers", Springer-Verlag, New York, LOC=QA76.9.A73, ISBN-0-387-91215-0. Good historical source on hardware developments. Good discussion of Cray-1 architecture in section 6.4. Especially good discussion of the STAR-100 and Cyber-205 in sections 7.3 and 7.4.

Kuck, D.J., Lawrie, D.H., and Sameh, A.H. (eds.) 1977, "High Speed Computer and Algorithm Organization", Academic Press, New York, LOC=QA76.5.S94, ISBN=0-12-427750-0. A SUPERB reference source! Contains both facts and food for thought on a variety of issues, machines, applications, etc. See especially pp. 71-84, "An Evaluation of the Cray-1 Computer", and pp. 287-298, "A Large Mathematical Model Implementation on the STAR-100 Computers", and don't overlook pp. 3-12, "It's Really Not as Much Fun Building a Supercomputer as it is Simply Inventing One".

Kuhn, R.H., and Padua, D.A. (eds.) 1981, "Tutorial on Parallel Processing", IEEE Computer Society, LOC=QA76.6.I548. See especially pp. 464-472, "Sorting on STAR", by H.S. Stone.

Metcalf, M. 1982, "Fortran Optimization", Academic Press, New York, LOC=QA76.73.F25, ISBN=0-12-492480-8. Mostly concerned with scalar optimization strategies; somewhat weak on details in places. Note Chapter 10 ("Fortran Portability"). See especially the Hitachi Integrated Array Processor discussion in Chapter 11 ("Vector Processors"), and the brief discussion in Chapter 12 ("Future Fortran") of the array processing language extensions proposed for Fortran "8X" by the ANSI X3J3 committee.

Peterson, W.P. 1983, "Vector Fortran for Numerical Problems on CRAY-1", Comm. of the A.C.M., vol. 26, pp. 1008-1021. Contains an excellent discussion of vectorization strategies appropriate for vector register machines such as Convex and Cray, plus much food for thought about other architectures.

Rodrigue, G. (ed.), 1982, "Parallel Computations", Academic Press, New York, LOC=QA76.6.P348, ISBN=0-12-592101-2. An excellent tutorial source. See especially pp. 129-151, "Swimming Upstream: Calculating Table Lookups and Piecewise Functions", by P.F. Dubois (parts of this discussion are reminiscent of our "Scatter/Compress" method).

Zakharov, V. 1984, "Parallelism and Array Processing", IEEE Trans. on Computers, vol. C-33, pp. 45-78. Good on history and overview.

Manuals for Particular Systems and Machines:

NRAO AIPS Group, "Going AIPS! (Programmer's Guide)", 15MAY84 edition. See chapter 11, "Using the Array Processors", which documents the AIPS model of a virtual vector device (called the "Q-routines" in the 15JAN85 release).

Cotton, W.D. and Wells, D.C., "AIPS and Array Processors", NRAO AIPS Memo No. 30, 02 December 1983.

Floating Point Systems, "AP Math Library", Vol. 2, Publ. No. 860-7288-005, November 1979. Documents the AP-120B library, which is the basis for the AIPS Q-routines.

Cray Research, "X-MP Series Mainframe Reference Manual", Publ. HR-0088, 1984.

Cray Research, "Fortran (CFT) Reference Manual", Publ. SR-0009, 1984. See the discussion of compiler directives and vectorization in pp. 1-13 thru 2-16 of part 3 of the manual.

Cray Research, "Library Reference Manual", Publ. SR-0014, 1984. See GATHER and SCATTER on p. 4-56, ISRCH on p. 4-59, and WHENNE on p. 4-67.

Control Data, "Model 205 Hardware Reference Manual", Publ. 60256020, Rev. C, November 1983. Note that most of the capabilities of the 205 hardware are supported by the "Q8" routines of the Fortran library.

Control Data, "Cyber 200 Fortran Version 2 Reference Manual", Publ. 60485000, Rev. B, June 1983. See chapter 11 ("Vector Programming") and the description of the "Q8" subroutines in chapter 11 ("Predefined Functions").

Convex Computer, "Architecture Handbook", Doc. 080-000120-000, 1984. See especially chapters 13 thru 16 which document the vector hardware, including vector mask, merge, compare, gather, and scatter instructions.

Convex Computer, "FORTRAN Language Reference Manual", Doc. 720-000150-000, 1984. See Appendix D which documents the optimization/vectorization compiler directives.

Convex Computer, "FORTRAN User's Guide", Doc. 720-000130-000, 1984. See chapter 4, "FORTRAN Compiler Optimizations".

Star Technologies, "ST-100 Array Processor, Processor Handbook", Publ. 90000003, Rev. A, August 1983.

Star Technologies, "Application Support Library User's Guide", Publ. 90000023, Rev. A, May 1983.

Masscomp, "Array Processor Subroutine Library -- Functional Specification", Rev. 0.3, preliminary version received March 1984.

Sky Computers, "SKYWAR Specifications--Preliminary", received August 1984. See Appendix B, "SKYWAR Subroutine Library".