Efficient Storage of Weather Radar Data

A. KRUGER AND W.F. KRAJEWSKI

Iowa Institute of Hydraulic Research, University of Iowa, Iowa City, IA 52242, U.S.A. (email: kruger@ihih.uiowa.edu  wfkrajewski@icaen.uiowa.edu)

SUMMARY

Weather radars produce large amounts of data and this has important implications for the archiving and analysis of data. The need for methods to deal with weather radar data sets will only increase as the United States National Weather Service (NWS) continues deployment of its 137 WSR-88D radars as part of the NEXRAD program. In this article we describe a compression and archiving strategy for weather radar data and present results for 62 days of reflectivity data from a radar operated by the NWS, as well as results for 60 days of reflectivity data for a radar operated by the Bureau of Meteorology in Australia. In their original format, these two sets require 60 GB of storage. In the format we describe, they require 4.8 GB and the data is portable across many platforms. The software for manipulating the converted data is simple, efficient, and easy to implement in C or Fortran. The savings in disk space and reduction in reading time compare favorably with what is attainable with deflation, the algorithm used in the popular gzip compression program. ©1997 John Wiley & Sons, Ltd.

KEY WORDS: weather radar data; NEXRAD; run-length encoding; compression; large data sets

INTRODUCTION

The work described in this paper is a result of the participation of the Hydrometeorology Laboratory at the Iowa Institute of Hydraulic Research at the University of Iowa in NASA’s Algorithm Intercomparison Workshop (AIW). The AIW was part of NASA’s Tropical Rainfall Measuring Mission (TRMM), a joint scientific satellite project between NASA and NASDA, the Japanese Space Agency. TRMM was designed to measure global-tropical rainfall via an assortment of instruments on a single spaceborne platform, including the first ever spaceborne weather radar. The AIW required the analysis of radar and raingage data for two radar sites. This paper deals with the radar data only.

The first radar data set consists of 62 days of reflectivity data from a WSR-88D radar at Melbourne, Florida, operated by the United States National Weather Service (NWS) as part of its Next Generation Weather Radar (NEXRAD) program. This program was established in 1980, and the goal is to bring on-line 137 WSR-88D radars throughout the United States and parts of the Caribbean by 1997. The viewers of the Weather Channel® in the United States already enjoy comprehensive weather information provided by many of these radars. The second radar data set consists of 60 days of reflectivity data from a radar at Darwin, Australia, operated by the Australian Bureau of Meteorology. The radar data was supplied to us on magnetic tapes in Universal Format (UF), compressed with the Free Software Foundation’s gzip program, along with a software library for reading the UF files, developed by NASA’s TRMM Office.
Data collected from weather radars are voluminous. For example, a single volume of radar reflectivity data from the Melbourne, Florida radar in UF requires about 1.6 MB of disk space. With a volume collected approximately every 6–10 minutes, this corresponds to 230–380 MB per day. For the AIW, the reflectivity data for this radar required about 11.15 GB of disk space. This disk space is less than (230 MB per day) × 62 days, but some days had missing volumes. A volume of radar reflectivity data from the Darwin, Australia radar in UF requires about 6.1 MB of disk space, and translates to about 0.88–1.46 GB per day. The space required for this radar’s reflectivity data for the AIW was about 48.8 GB.

**CONSTRAINTS**

Archiving and processing such large data sets are formidable tasks. The brute force approach of loading a few days’ radar data from tape to a large hard disk, uncompressing the data, analyzing, loading another chunk, and so on, is impractical. For one, it limits access to the data and prevents parallel research activities. Second, many passes over the data are required to test algorithms and the sensitivity of results. Third, as part of the AIW, we have created a software tool, called VRAD for browsing and visualizing the radar data. For this tool to be useful, rapid access to individual volumes, sweeps, and rays in the data is required. (These elements of the radar data are explained later.) Fourth, NASA’s software for reading the data in UF is written in C, but many of the potential users of the radar data have large investments in Fortran routines, and prefer or even require Fortran. Thus, easy access to the radar data in this language, possibly using some common language extensions, was an important consideration. One approach is to provide a C library for reading the data, accompanied with interface routines to facilitate interlanguage calls, but this has a host of problems too. Using Fortran 90 is another option, but the language is not yet in widespread use.

To address these constraints, we have developed a lossless compression scheme and flexible file format called ASCII-RLE that greatly reduces the space requirements for weather radar data sets. The two radar data sets in ASCII-RLE require about 4.8 GB of disk space and can be comfortably kept on-line on one or two fast hard disks. Reading time is small (1.5 hours for the combined data sets), the data is portable across many platforms, and the software for reading the data is simple.

Finally, the time-consuming part of the conversion process was loading the data from tape and uncompressing it, and not the UF to ASCII-RLE conversion part. While it was desirable to have the conversion time small, this was not an overriding concern. This is because the data were converted once, but used many times. In the rest of the paper, we do not consider the conversion time.

**RADAR DATA**

A typical radar collects data as follows: it emits a coherent train of microwave pulses and coherently processes reflected pulses in a pulse-Doppler radar processor. To achieve a desired resolution, many pulse measurements must be integrated. Conceptually, one can view the processed and integrated pulses as bins of a radar beam or ray. The radar then moves to the next azimuth and the process is repeated. When one azimuthal sweep is complete, the radar increases the antenna elevation angle and collects the next sweep. A complete set of azimuthal sweeps collected over all elevation angles is called a volume, and once a volume has been collected the radar moves the antenna to its lowest elevation angle in preparation for the base sweep of the next volume. The data collection strategy (e.g. number of bins in a ray) and
the antenna sweeping strategy (e.g., number of azimuths in an azimuthal sweep, number of sweeps in a volume) depend upon the radar, and are in many cases under control of the radar operator.

Weather radar data are usually archived in a format that closely mimics the way the data are collected. Volumes are typically saved one per file, though it is common to store volumes of multiple variables in the same file. Thus, a single volume file may contain pure reflectivity, quality-controlled reflectivity, as well as Doppler velocity information. Each volume file has a header that contains information about the volume such as date, location, number of sweeps, and so on, followed by the rays that make up the volume. Each ray is preceded by a ray header that contains information about the ray, such as ray date and time, ray elevation and azimuth, the number of bins for the ray, and so on. Following the header are the data for the ray as a number of bins. Each bin may contain the reflectivity factor $Z$ or another measurement such as Doppler velocity. Because the dynamic range of the reflectivity factor $Z$ is very large, it is normally converted to decibels and the resulting quantity is commonly referred to as dBZ:

$$\text{reflectivity factor in dBZ} = 10 \log_{10}(Z)$$

It is this quantity that is usually archived. The data range depends upon the variable, but it is common to allow for 256 different values (8-bit precision).

DEALING WITH FLOATING-POINT NUMBERS

NASA’s radar library, which was used to read the UF files, returns the radar data as a mixture of floating-point and integer data types. For example, ray numbers are integers but ray azimuths are floating-point. The bin values are returned as integers in the range 0-255 with the following interpretation. A value 0 means the reflected energy is below the signal-to-noise threshold, values 1 and 2 are used as quality control flags, and the following function maps the values in the range 3-255 to dBZ:

$$F(x) = \frac{(x - 3)}{2.0} - 64.0$$

When converting from UF to ASCII-RLE we do not apply this mapping, since it is in general quite difficult to deal with floating-point data in a portable fashion. Rather, we code the integer bin values, and apply the mapping from integer to dBZ after decoding.

Floating-point header fields in the ASCII-RLE files are formatted as ASCII strings using the %g fprintf conversion character in C. This conversion character suppresses trailing zeros and a trailing decimal point. The result is a compact ASCII string that preserves the precision. The only floating-point header field not treated this way is the ray azimuth field. This field is mapped to an integer by multiplying with 64.0 and discarding the fractional part, which is always 0.

REDUNDANCY IN RADAR INFORMATION

Many parameters such as the radar location, volume date, sweeps/volume, bins/ray, and so on, are archived as part of the ray headers, and the headers constitute a large part of a volume file. This is especially true for higher-elevation sweeps, since the number of bins/ray is generally less than at lower elevation sweeps.

For a specific radar site, most of the parameters in the ray headers typically do not change.
and need not be stored. For example, while a radar may conceivably be a small mobile unit, the radar location is almost always fixed. Other parameters that change little include the number of rays/sweep and the number of bins/ray. On the other hand, while the radar elevation angle for a specific sweep (e.g. the second sweep) is nominally the same between volumes, and the azimuth for a specific ray (e.g. the third ray) is nominally the same between sweeps, in practice radars break down, are taken off-line for maintenance, and their operational parameters are changed. Also, given their size and complexity, they are subject to small mechanical and electrical changes. Furthermore, a volume may straddle two days, so that rays within the same volume may have different days recorded in their headers. This means each ray header contains potentially unique information about that ray. Thus, there are some parameters of the ray headers that (a) change for certain with each ray, (b) may potentially change, and (c) almost never change. The parameters in cases (b) and (c) are clearly redundant and need not be included with every ray.

The data contain a substantial amount of redundancy. Some relevant statistics for 144 volumes of data for 28 December 1993 and 14 March 1994 for the radar at Darwin, Australia are summarized in Table I. About 99% of the rays for 28 December contain data above the signal-to-noise threshold, suggesting substantial atmospheric activity. This is confirmed by displaying the radar reflectivity data as well as surface-based raingages. However, only about 32% of the bins contain data above the signal-to-noise threshold. There is also a high degree of serial correlation between bin values in a ray, and the bins form runs of similar values. The average length of these runs is about 2.9. The redundancy and serial correlation are even more pronounced in data sets for days with little atmospheric activity, such as the 14 March 1994 set, where only 12% of the rays contain bins above the signal-to-noise threshold and as little as 0.13% of the bins contain data above the signal-to-noise threshold. Furthermore, the average length of these runs is about 42.

Table I. Example of daily statistics for the radar at Darwin, Australia. Each day corresponds to 144 volumes

<table>
<thead>
<tr>
<th></th>
<th>28 December 1993</th>
<th>14 March 1994</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a) Total number of rays</td>
<td>881,280</td>
<td>881,280</td>
</tr>
<tr>
<td>(b) Rays with bins above signal-to-noise threshold</td>
<td>873,677</td>
<td>104,972</td>
</tr>
<tr>
<td>(c) Total number of bins</td>
<td>406,270,080</td>
<td>406,270,080</td>
</tr>
<tr>
<td>(d) Bins above signal-to-noise threshold</td>
<td>129,975,345</td>
<td>545,558</td>
</tr>
<tr>
<td>(e) Number of runs in (b)</td>
<td>137,683,602</td>
<td>926,198</td>
</tr>
<tr>
<td>(f) Average length of runs in (b)</td>
<td>2.918</td>
<td>42.11</td>
</tr>
<tr>
<td>(g) Number of runs in (b) with length &gt; 3</td>
<td>4,828,835</td>
<td>334,499</td>
</tr>
<tr>
<td>(h) Average length of runs in (g)</td>
<td>55.7</td>
<td>144.82</td>
</tr>
</tbody>
</table>

DATA COMPRESSION

The compression strategy we have developed exploits the redundancy in radar data in two ways. First, rays in which all the bins have values below the signal-to-noise threshold are not coded. Second, the other rays are run-length encoded as follows. A ray is examined for adjacent bins of identical values. If the length of such a run exceeds 3, then it is coded as
where $M$ is a special marker symbol that signals the start of a run-length code, $count$ is the run length, and $value$ is the integer value of the bins in the run. Bins that do not form part of runs with lengths greater than 3 are not coded; they are simply written to the output stream. Run lengths greater than 255 are coded as multiple runs, with each $count$ smaller than 256. Thus, a run of 640 $values$ is coded as

$$M \ 255 \ value \ M \ 255 \ value \ M \ 130$$

Since the $values$ are in the range 0–255, this ensures that everything in the output file can be represented as byte-sized variables. This fact is important for moving the resulting files between little-endian and large-endian platforms. $M$ is also an integer in the range 0–255, so to distinguish between a valid marker and a bin that has the same value as the marker, such bins are coded as

$$M \ 0$$

In other words, this zero length run-length code is interpreted as a bin with value $M$. By choosing $M$ appropriately one can ensure that this occurs very infrequently. We computed a histogram of the bin values from a representative subset of the reflectivity data for the two radars and used a value that occurred very infrequently. This value turned out to be one of the quality-control flags described above, namely, ‘1’. Thus, with the marker symbol $M$ ‘1’, the sequence

$$8 \ 3 \ 5 \ 5 \ 5 \ 5 \ 5 \ 5 \ 7 \ 1 \ 6 \ 6 \ 6 \ 6$$

is coded as

$$8 \ 3 \ 1 \ 6 \ 5 \ 7 \ 1 \ 0 \ 1 \ 4 \ 6$$

One can use the data in Table I to estimate how much is gained by run-length encoding, as follows. Only rays that have some values above the signal-to-noise threshold are saved. The number of bins in these rays are given by multiplying entries (e) and (f) in the table. For 28 December, 1993:

Uncoded Size = $137,683,602 \times 2.918$

= $401,760,751$

Each run of bins longer than 3 is run-length encoded, the length of a run-length code is 3, and each run-length code replaces an average of 55.7 bins—see entry (h) in the table. Thus, the number of bins saved by run-length encoding is found by multiplying $(55.7 - 3) = 52.7$ with entry (g) in the table:

Savings = $4,828,835 \times 52.7$

= $254,479,605$

The compression ratio now follows

$$\frac{\text{Uncoded Size}}{\text{Coded Size}} = \frac{401,760,751}{401,760,751-254,479,605} = 2.73$$
That is, with the run-length encoding the rays require roughly a third of the space when compared to the uncoded rays. For days with very little atmospheric activity, such as 14 March 1993, the savings are even more—the run-length encoded rays require only 4% of the space the uncoded rays require. The actual compression ratios are somewhat lower in practice, since some runs are longer than 255 and for these runs more than 3 bytes per run-length code are required.

The application of run-length encoding to radar reflectivity data was inspired by similar work done at the National Weather Service that used a simpler run-length encoding scheme where all runs, not just those runs longer than 3 bins per run, were coded as count value pairs. With this method, the length of each run-length code is 2, and the size of the run-length coded bins for 28 December 1933 is found by multiplying entry (e) in the table with 2:

\[
\text{Uncoded Size} = 137,683,602 \times 2 = 275,367,204
\]

The compression ratio with this method is

\[
\frac{\text{Uncoded Size}}{\text{Coded Size}} = \frac{401,760,751}{275,367,204} = 1.46
\]

and is about half of the compression ratio achieved by the run-length scheme we have used.

**HEADER COMPRESSION**

Our compression strategy exploits the substantial amount of redundant ray header information by treating the ray header of the first ray in an azimuthal sweep as a sweep header that provides default values for all the rays in the sweep. With each ray, only those header fields that normally differ from the sweep header are encoded. These fields are the ray number, the ray azimuth, the number of run-length codes necessary to run-length encode the ray, and the difference in seconds between the first ray of the azimuthal sweep’s time and the current ray time. Since only those rays that contain values above the signal-to-noise threshold are run-length encoded, it is necessary to store the ray numbers to identify a ray.

Ray headers in volumes that are not properly quality-controlled may contain errors or inconsistencies in some header fields. For example, some ray headers had values such as 70 in their seconds subfield for the ray time, while the valid range is 0–59. Also, the assumption that a ray header can be coded by overriding four fields in the sweep header sometimes fails. For example, a volume may straddle a day so that some ray dates differ from the sweep date. Both of these situations are handled by replacing the four fields normally written to the output file with a special code:

\[-1 \ -1 \ -1 \ -1\]

followed by the full header for that ray. This special code signals the decoding software to ignore the defaults for this ray and instead get its header values from the custom header that follows this line.

**FILE FORMAT**

We decided to use an ASCII file format, and call the combination of the run-length encoding and ASCII file format, ASCII-RLE. With the exception of white space (tabs, spaces, and new-lines) an ASCII-RLE file contains only printable ASCII characters. Additionally, lines
are limited to no more than 82 characters, including new-lines. Headers are saved as ASCII fields separated by white space. Strings in the headers are enclosed by double quotes. The 82 character limit on the line length requires that the header be several lines long. Thus, the first field of the first header line is a number that indicates the number of header lines.

Each ray’s corresponding run-length codes are stored as one or more lines of ASCII characters. Groups of 3 bytes are stored in four ASCII characters, 6 bits per character. All are offset by an ASCII exclamation character (decimal 33) to make the characters printable. Each line consists of a character count, followed by at most 80 printable ASCII characters, followed by a new-line. The character count is a single printable character that represents an integer and is the number of ASCII-RLE codes in the rest of the line. It always ranges from 1 to 60. The count can be determined by subtracting the equivalent decimal value of the ASCII exclamation character (decimal 33) from the equivalent decimal value of the count character. Extra meaningless data are included, if necessary, to make the character count a multiple of 4. This ASCII-encoding process is analogous to, and modelled after the UNIX uuencode program.

Below is a fragment of data for the radar at Melbourne, Florida that shows one ray header and slightly edited ASCII-RLE data. The ‘KMLB’ is the official designation for this radar. The first header field is ‘3’ and indicates that there are three lines in the header. Following the header, there is a line that contains four numbers. These are the ray number, ray azimuth 64 (in degrees), the number of run-length codes that follow, and the difference between the ray time and the first ray in the sweep. Since this is zero, one can tell that this is the first ray in the sweep.

```
3 1 20 "KMLB" 3 14 366 -1 0 1000 -1 321 19.2632 -1 0.9375
0.4375 -1 -1 11 -1 17 1.71452 0 0 -1 -1
2449182 1993 7 13 0 4 55 UT -1 0 1 460 141 0 255 226.266 DEG
0 14481 122 0
]!8E!CK;GL\;SL;/@I*79E*W@KK?FM+[LML'ML[WVM\$[PM$!P\T& ...
]NL#OMLX!PLOULK+I+3IM,?_PM,$T-L$Q<SOK[\_VK:6!!##=L< ...
#SQ#0
```

The first two ASCII-RLE data lines start with the ‘]’ character, which is the count character for these lines. The decimal equivalent of this character is 93, and keeping in mind that all characters are offset by 33 to make them printable it follows that there are 93 - 33 = 60 ASCII-RLE codes on each of these two lines. The last of the ASCII-RLE data lines starts with the ‘#’ character, and translates to 35 - 33 = 2 ASCII-RLE codes. Thus, these three lines contain 60 + 60 + 2 = 122 ASCII-RLE codes. Since groups of 3 bytes are packed into four ASCII characters this means there are (60 × 4) / 3 = 80 ASCII characters on the first two ASCII-RLE lines, excluding the count characters. For the last line (2 × 4) / 3 is less than 4, and as was mentioned before, meaningless data (ignored during decoding) are added to make the character count a multiple of 4.

One important advantage of ASCII files is that they are easily portable between different machine architectures, since issues related to the word length and byte ordering of machines are greatly simplified. The result is that software for manipulating the data is also simplified. Standard Fortran 77 does not have intrinsic bit-manipulation functions that are required for efficient implementation of the ASCII encoding/decoding routines. However, practically all Fortran 77 compilers implement the required bit-manipulation functions as extensions and this is not a real obstacle.
Because the headers are in ASCII, one can easily and efficiently extract basic information from a file that contains multiple sweeps or volumes with standard text processing utilities. For example, on a UNIX machine the command

```
sed -n '/"KMLB"/,/ DEG/p' file
```
displays all the sweep headers in `file`, where `file` is assumed to be an ASCII-RLE file for the KMLB radar. The resulting sweep headers can be piped into AWK scripts to extract a surprising amount of information.

THE SOFTWARE

The software was written in ANSI C, and consists of two parts. First, programs to convert from a radar’s native format or from some other intermediate format to the ASCII-RLE format described in this article. Second, a library of routines to read the radar data in the ASCII-RLE format, called `libaiw`. There is presently one conversion program, `uf2rle`, that reads radar data in UF and converts it to the ASCII-RLE format. The library for reading the ASCII-RLE data contains several routines, but a user typically requires only two functions and a macro to read an ASCII-RLE file. The following listing shows the skeleton of a typical program:

```c
float value[2000]; /* dBZ Values */
sweep_header_t sweep; /* declared in "aiw.h" */
ray_header_t ray; /* declared in "aiw.h" */

/* Scan through whole file, computing average of the ISDATA values in each sweep. Print sweep headers and average values as we go. */
while (!feof(fp1)) { /* get next sweep header */
    readSweepHeader(fp1,&sweep);
    /* Compute statistics for all the ISDATA values in this sweep */
    sum = count = 0; /* get the next azimuth */
    /* 0 = North, 90 = East */
    for (i=0;i<=sweep.nonzero-1;i++) { /* values > SN threshold */
        readAzimuth(fp1,sweep,&ray,value;
        ray_azimuth = ray.azimuth;
        ray_number = ray.ray_number;
        for(j=0;j<=ray.nbins-1;j++)
            if (ISDATA(value[j])) {
                sum = sum + value[j];
                count++;
            }
    }
    printf("Sweep: %d, Average: %7.4f\n",sweep.number,sum/count);
    writeSweepHeader(stdout,sweep);
}
```

The `readSweepHeader` routine reads the header for the next sweep in the file and fills the structure `sweep`. This structure is declared in an include file `aiw.h`. In addition to the default
values for all the ray headers, this structure contains the number of rays that were coded in the 
nonzero member. The term ‘nonzero’ is a misnomer and really is a short-hand for ‘above 
signal-to-noise threshold’. In other words, the nonzero member is the number of rays that 
have values above the signal-to-noise threshold. The routine readAzimuth reads the next 
ASCII-RLE azimuth from the file and returns the ray bin values in the array values. The 
macro ISDATA returns TRUE if a particular bin value is above the signal-to-noise threshold.

TESTING

To test the conversion software, a radar data file in UF was first converted to an ASCII file that 
served as a reference. Next u2rle was used to convert the UF file to an ASCII-RLE file. This 
ASCII-RLE file was then read back with the decoding software and the output was written 
to a second ASCII file that was compared to the reference ASCII file using the UNIX diff 
utility. This process was repeated for every UF file, and while it was was time-consuming, it 
detected a number of UF files that were not properly quality-controlled. It also provided a very 
thorough test for the software in general. For the AIW in particular, it provided an exhaustive 
test of the coding/decoding software, since every ASCII-RLE file had been tested against its 
corresponding UF file. In other words, while there may be a bug in the software (very unlikely) 
it was irrelevant for the AIW, since the coding/decoding software had been tested for all the 
relevant data. Thus, the researchers had confidence that data read from ASCII-RLE files were 
identical in all respects to the original data in the UF files.

A COMPRESSION EXPERIMENT

To evaluate the effectiveness of our compression scheme, we have experimented with three 
other basic algorithms, described briefly below.

Splay compression uses splay trees to approximate adaptive Huffman Coding. It is relatively 
simple to implement and requires little memory. It is generally faster, but achieves slightly 
less compression than adaptive Huffman coding. In most cases it provides significant better 
compression than normal Huffman coding. The splay compression routines used in the 
experiment are C translations of previously published Pascal routines. Dynamic Markov coding (DMC) is a method of building complex statistical state models 
that are then used to drive a coder. It starts with a simple Markov model of the data and 
extends this model by increasing the number of states as compression progresses, using 
a cloning strategy. In DMC, the model can grow without bounds, depleting the available 
memory. In practice, when the number of states reaches a preset limit, or equivalently, the 
memory allotted have been used, the model is reset. DMC achieves better compression when 
more states are allowed, but at the expense of longer execution time. In this article DMC refers 
to a dynamic Markov modeler driving an arithmetic coder. The modeler is reset when 
needs more than 16 MB of RAM, and this limits the number of states to 1 million for the 
particular implementation.

Deflation is a composite algorithm used by the MS-DOS pkzip and Free Software Foun-
dation’s gzip programs. The compressed data set consists of a series of arbitrary-sized 
blocks, and each block is compressed using the LZ77 algorithm. Phrases, distances, 
and lengths in the compressed data are represented using Huffman codes—one Huffman code 
tree for phrases and lengths, and another code tree for distances. The Huffman trees for blocks 
are independent each other, and the Huffman trees themselves are compressed using Huff-
man coding. Properly implemented, deflation is fast, has moderate memory requirements, and
generally outperforms LZW on which the UNIX compress program is based. We have culled routines from the gzip sources, slightly adapted it for our purposes, and in this article, DEFLATE is roughly equivalent to compression with gzip using the -9 option. This option favors compression over execution speed.

These methods were applied to the radar reflectivity data from the two sites, and the combined results are shown in Figure 1. In the figure UF refers to the radar data in Universal Format, SPLAY refers to splay compression, RLE-SPLAY refers to run-length encoding followed by splay compression, RLE-DEFLATE is run-length encoding followed by deflation, and so on. An R in parenthesis means rays were compressed independently of each other. An S in parenthesis means that a whole sweep was treated and compressed as a block of data.

In the figure, ‘read-time’ refers to the elapsed (wall-clock) time for reading the data after it has been compressed using a specific method. A simple model for this time is

$$t_{\text{read}} = t_{\text{decompress}} + t_{I/O} + t_{\text{network}}$$

where $t_{\text{decompress}}$ is determined by the compression algorithm, and $t_{I/O}$ is determined by the program input-output (I/O) time. Many computing centers follow the file-server paradigm where user and data files are on a centralized machine that serves requested files to client machines over a network. The parameter $t_{\text{network}}$ is a catch-all for factors such as network bandwidth and traffic, as well as server activity. The experiment was performed on a 125 MHz, HP735 series 9000 workstation. The machine was dedicated to the experiment, and storage was local, so one can assume $t_{\text{network}} = 0$. The figure represents about three weeks of CPU time.

The programs were written in C and the running times of the programs were first profiled using a small subset of the data and a performance analysis tool. Run-time bottlenecks were identified and removed as much as possible using techniques such loop-unrolling and subroutine in-lining, and then compiled using an optimizing compiler. The size of file I/O buffers can dramatically influence the I/O time of a program. We have used 16 KB buffers after experimentally determining that larger buffers do not reduce I/O time significantly for the experimental setup.

‘Disk Space’ is the actual space used, and is generally different than the file sizes: disk sector or file block size influence disk requirements and is highly machine-dependent. For example, if files are stored as 512-byte blocks then a 513-byte file requires two blocks or 1024 bytes. Thus, in practical files systems there is a waste-factor that can be a problem with many small files. However, in our experiment the file sizes are large enough that this effect can be safely ignored.

As one would expect, sweep-based compression performs better than ray-based compression with a smaller read time as well as smaller disk space requirements. However, the differences are relatively small, except for DEFLATE, where sweep-based compression is much better than ray-based compression. The reason for this is that with ray-based compression the dictionary is flushed at the start of each ray, leading to a loss of valuable information. We did not perform extensive ray-based measurements using DMC, but considering the nature of the DMC algorithm we expect ray-based DMC to perform substantially worse than sweep-based DMC. This observation was borne out by cursory measurements.
Figure 1. Disk space and read time statistics for the radar data in UF and several alternative formats. (S) designates sweep-based compression and (R) designates ray-based compression. (Refer to the text for details)
DISCUSSION

Of the methods tested, DMC gave the smallest file size, but with an unacceptable read time—it takes about five times longer to read and uncompress the DMC-compressed data than reading the data in UF. Clearly $T_{\text{decompress}}$ dominates in DMC. Even if the data are preprocessed with run-length encoding, the read time is still longer than the read time with UF. By comparison, DEFLATE (S) provides almost the same compression but the read time is roughly 50 times smaller than DMC’s read time. Any of the SPLAY variants give substantial compression ratios, but read times are poor. As with DMC, $T_{\text{decompress}}$ dominates in these algorithms.

From the measurements it is clear the run-length encoding is overall very effective in terms of reading time and disk space, and the only realistic alternative is a DEFLATE variant. In addition to this, run-length encoding is simple and almost trivial to implement in C and Fortran 77. By comparison, DEFLATE is very complicated, and requires tree data structures and language features such as pointers, and to a lesser extent dynamic memory allocation, for efficient implementation. While one can in principle implement DEFLATE in Fortran 77, this would be a challenge for most programmers. As was mentioned before, implementation of software in Fortran to read the compressed data was an important consideration. Thus, run-length encoding was an easy choice.

Figure 1 shows that sweep-based compression (RLE (S)) gives the smaller file sizes and read times when compared to ray-based (RLE (R)) compression. We have, nevertheless chosen ray-based run-length encoding, since it allows convenient access to individual rays in a sweep/volume without having to read and uncompress the whole sweep/volume. This possibility is desirable in many uses of the radar data. Consider for example, the construction of a range height indicator (RHI) for a volume. An RHI is a vertical slice of a radar volume at a specific azimuth, and RHIs are fundamental to vertical profile studies. In a sweep-based compression scheme, one has to read and uncompress every sweep of a volume to construct an RHI. On the other hand, in a ray-based compression scheme, only those rays that are needed are read and uncompressed, which is much faster. Another example is rainfall estimation algorithms that operate on sectors (a group of adjacent rays in a sweep) of radar data, and require multiple reading passes over the data.

An effective strategy is to create an index that contains the start locations of the sectors in each sweep. Programs use this index to rapidly navigate the data by looking up the start location of a sector, positioning the file pointer at the start location, reading and processing the sector, looking up the location of the next sector, moving there, and so on. The savings in reading time obtained with this ray-based approach far outweigh the comparatively small savings in disk space one would get by using a sweep-based scheme instead. All the compression algorithms we have explored treat the data as a serial stream, and exploit one-dimensional redundancy to achieve compression. Two- and three-dimensional correlations exist between reflectivity measurements and one can exploit this to achieve higher compression ratios. However, as with the sweep-based schemes discussed so far, it makes accessing individual rays cumbersome.

SUMMARY

ASCII-RLE is a very effective compression scheme and flexible file format that greatly reduces the disk space and reading time requirements for radar reflectivity data sets. Two large radar reflectivity data sets were converted to ASCII-RLE and were successfully used in a NASA research program, the AIW. We have evaluated the compression and reading-time efficiency of ASCII-RLE by performing an experiment that compares the disk space requirements and
EFFICIENT STORAGE OF WEATHER RADAR DATA

reading times of the radar data in ASCII-RLE and several other compressed formats. There are compression schemes that provide better compression but their reading times are unacceptable. Then there are compression schemes with smaller disk space requirements and reading times, but that are much more complex to implement. ASCII-RLE provides substantial savings in disk space and reading time, is portable, and it is easy to access individual rays in an ASCII-RLE file. Finally, ASCII-RLE is sufficiently simple (all-ASCII, line lengths limited to 82 characters) that Fortran can be used to read the data.

Since the AIW, substantial parts of the ASCII-RLE data as well the software for reading have been transferred without problems to several UNIX-based platforms including HP-UX, IBM AIX, and SGI, and are being used by colleagues at several universities in the US and France, as well as at NASA. The software source code is available from the authors.

ACKNOWLEDGEMENTS

This work was made possible through the support of NSF (award EAR-9419103) and NASA (grant NAG 5-2774). We gratefully acknowledge this support. We also gratefully acknowledge the valuable comments of two anonymous reviewers.

REFERENCES