CONF-9503181--1

LA-UR 95-1659

Los Alamos National Laboratory is operated by the University of California for the United States Department of Energy under contract W-7405-ENG-36

TITLE: **WAVELET SUBBAND CODING OF COMPUTER SIMULATION OUTPUT USING THE A++ ARRAY CLASS LIBRARY**

AUTHOR(S): Jonathan N. Bradley, Christopher M Brislawn, Daniel J. Quinlan, Hua D. Zhang

SUBMITTED TO: Proceedings of the NASA Space and Earth Science Data Compression Workshop
Univeristy of Utah,
Salt Lake City, UT
March 27, 1996

DISCLAIMER

MASTER

Los Alamos

Los Alamos National Laboratory
Los Alamos New Mexico 87545

Wavelet Subband Coding of Computer Simulation Output Using the A++ Array Class Library

Jonathan N. Bradley, Christopher M. Brislawn, Daniel J. Quinlan,
and Hua D. Zhang
Los Alamos National Laboratory, Los Alamos, New Mexico 87545–1663

Veyis Nuri
School of EECS, Washington State Univ., Pullman, WA 99164

I. Introduction.

In previous work presented at the 1993 IEEE Data Compression Conference and NASA Workshop [1, 2], the first two authors presented results demonstrating the viability of applying techniques from digital image compression to the problem of reducing the data storage requirements for output from large-scale supercomputer simulations. The data under consideration was the output of global ocean circulation models [3] being run on Cray X-MP's at the National Center for Atmospheric Research and on a Thinking Machines CM-200 at the Los Alamos Advanced Computing Lab. The half-degree resolution model (grid size $768 \times 288 \times 20$) then in use on the CM-200 produced about 6.5 gigabytes (GB) of floating point output for data visualization (six 2-D fields archived at three-day intervals) per decade of simulation.

The compression experiments described in [1, 2] were based on trained (Linde-Buzo-Gray) vector quantization of an octave-scaled discrete wavelet transform (DWT) subband decomposition. Since the data formed a time series, a first-order (frame difference) predictor was used to exploit temporal redundancy in the output. The experiments done at that time were run off-line: no attempt was made to compress the data as it was being generated in the simulations. These experiments demonstrated that the data could be compressed to approximately 0.25 bits per pixel (bpp) with little or no perceptual loss when displayed in false color "movie" visualizations, a common way of presenting the results for human assessment.

The current model, which is running on the new CM-5 at the Advanced Computing Lab, has a spatial resolution of 1/5 of a degree (on average) and is being computed on a grid of size $1280 \times 896 \times 20$. The 2-D visualization data mentioned above now amounts to about 34 GB per decade. In addition, 3-D restart dumps (every three months) add another 64 GB/decade, and time-averaged 3-D diagnostic data (dumped monthly) contributes 192 GB/decade, for a total floating point storage requirement on the order of 300 GB/decade of simulation. This is expensive data, too: a decade of simulation consumes about 200 hours of run-time on a 512-node partition of the CM-5.

The goal of the project described here is to produce utility software for off-line compression of existing data and library code that can be called from a simulation program for

on-line compression of data dumps as the simulation proceeds. Naturally, we would like the amount of CPU time required by the compression algorithm to be small in comparison to the requirements of typical simulation codes. We also want the algorithm to accomodate a wide variety of smooth, multidimensional data types. For these reasons, the subband vector quantization (VQ) approach employed in [1, 2] has been replaced by a scalar quantization (SQ) strategy using a bank of almost-uniform scalar subband quantizers in a scheme similar to that used in the FBI fingerprint image compression standard [4]. This eliminates the considerable computational burdens of training VQ codebooks for each new type of data and performing nearest-vector searches to encode the data. The comparison of subband VQ and SQ algorithms in [5] indicated that, in practice, there is relatively little additional gain from using vector as opposed to scalar quantization on DWT subbands, even when the source imagery is from a very homogeneous population, and our subjective experience with synthetic computer-generated data supports this stance. It appears that a careful study is needed of the tradeoffs involved in selecting scalar vs. vector subband quantization, but such an analysis is beyond the scope of this paper.

Our present work is focused on the problem of generating wavelet transform/scalar quantization (WSQ) implementations that can be ported easily between different hardware environments. This is an extremely important consideration given the great profusion of different high-performance computing architectures available, the high cost associated with learning how to map algorithms effectively onto a new architecture, and the rapid rate of evolution in the world of high-performance computing. For instance, co-author Nuri and another Los Alamos graduate intern, David Alvarez of UC Berkeley, spent the summer of 1993 trying to write a WSQ codec on the CM-5 using CM Fortran, a language implementing many of the array-processing features of Fortran 90 in a data-parallel manner. Since CM Fortran is customized for the CM architecture, however, any such code would require extensive rewriting before it could be ported to another platform. Moreover, the use of a parallel platform-dependent language made it difficult to develop and evaluate code effectively on workstation computers.

Given their relatively low duty-cycle and the limited access time available for developing specialized programming expertise on high-performance machines, we decided in the summer of 1994 to start over again and write a WSQ implementation using the A++/P++ array class library [6, 7]. The array class library is a C++ library originally designed for adaptive mesh algorithms; until now, applications at Los Alamos have focused on PDE solvers for hydrodynamics problems. Using a C++ class library has the advantage of allowing us to write the scientific algorithm in a high-level, platform-independent syntax; the machine-dependent optimization is hidden in low-level definitions of the library objects. Thus, the high-level code can be ported between different architectures *with no rewriting of source code* once the machine-dependent layers have been compiled. In particular, while "A++" refers to the current serial library, the same source code can be linked to "P++" libraries, which contain platform-dependent parallelized implementations of the array operations, once they have compiled for a given parallel machine. Writing a P++ array class library for the CM-5 is a current project in the Computer Research Group at Los Alamos Laboratory. The present paper compares the overhead incurred in using A++ library operations for WSQ implementation with a "traditional" serial direct-form implementation.

When a P++ library is completed for a parallel machine, we will be able to make the same comparison of serial direct-form implementation with a parallelized P++ implementation by linking the *same* array class source code to the P++ library. While we presently incur a performance penalty in exchange for the platform-independent high-level A++ syntax, in a parallel environment we expect that the syntax penalty should be completely dwarfed by the difficulties in parallelizing the serial direct-form implementation.

Remarks. Over the past two years, the formulation of the Bryan-Cox-Semtner ocean circulation model used at Los Alamos has been improved by replacing the "rigid lid" approximation implemented in [3] with a free-surface model [8]. This change allows the model to compute meaningful predictions of the free surface height above mean sea level. The numerical experiments presented in this report all refer to calculations performed on this 2-D free surface height field. The height data was extrapolated smoothly across the continental land-masses by a Poisson equation-solver, resulting in logically rectangular 2-D data. While this is not a realistic approach to take in an on-line library, it does provide an indication of how well the WSQ approach is capable of performing on smoothly extrapolated data. Finding more efficient methods of obtaining smooth extrapolations in two or more dimensions appropriate for subband coding remains an area of ongoing study.

Acknowledgments. Kristi Brislawn of Los Alamos National Lab provided valuable help in generating the timing results (presented below) on the Cray Y-MP.

II. The WSQ Compression Algorithm

The data compression algorithm employed in this work incorporates three well-known concepts from the image compression literature: (1) a subband decomposition of the input image, (2) scalar quantization of the subbands, and (3) Huffman coding of the quantized coefficient indices [9, 10, 11]. Decompression involves Huffman decoding, dequantization of the indices, and subband synthesis; see Figure 1. The WSQ algorithm used here is closely related to the algorithm that was developed for the compression of gray-scale fingerprint images [4], the main difference in this case is that an octave-scaled DWT decomposition is used instead of the nonuniform wavelet packet decomposition employed in the FBI standard. As mentioned above, the algorithm is also similar to the one proposed in [1, 2], with LBG vector quantization replaced by almost-uniform scalar quantization. Although this work deals with still image compression, we expect that it will eventually be modified to incorporate predictive coding such as that described in [2] and [12] in order to improve compression performance by exploiting interframe dependencies.

The image subband decomposition is based on a 1-D two-channel perfect reconstruction multirate filter bank that is cascaded to form a discrete wavelet transform (DWT) decomposition. Linear phase filters are used with the lowpass filter consisting of 9 taps and the highpass filter of 7. The particular filter pair used corresponds to a biorthogonal wavelet basis constructed in [13]. The 2-D DWT is implemented by applying a 1-D DWT first to the rows and then to the columns of the image, yielding a four-channel decomposition. The lowpass subband is then cascaded back through the two-dimensional analysis bank to produce a more refined decomposition. The cascade is repeated a number of times (up to

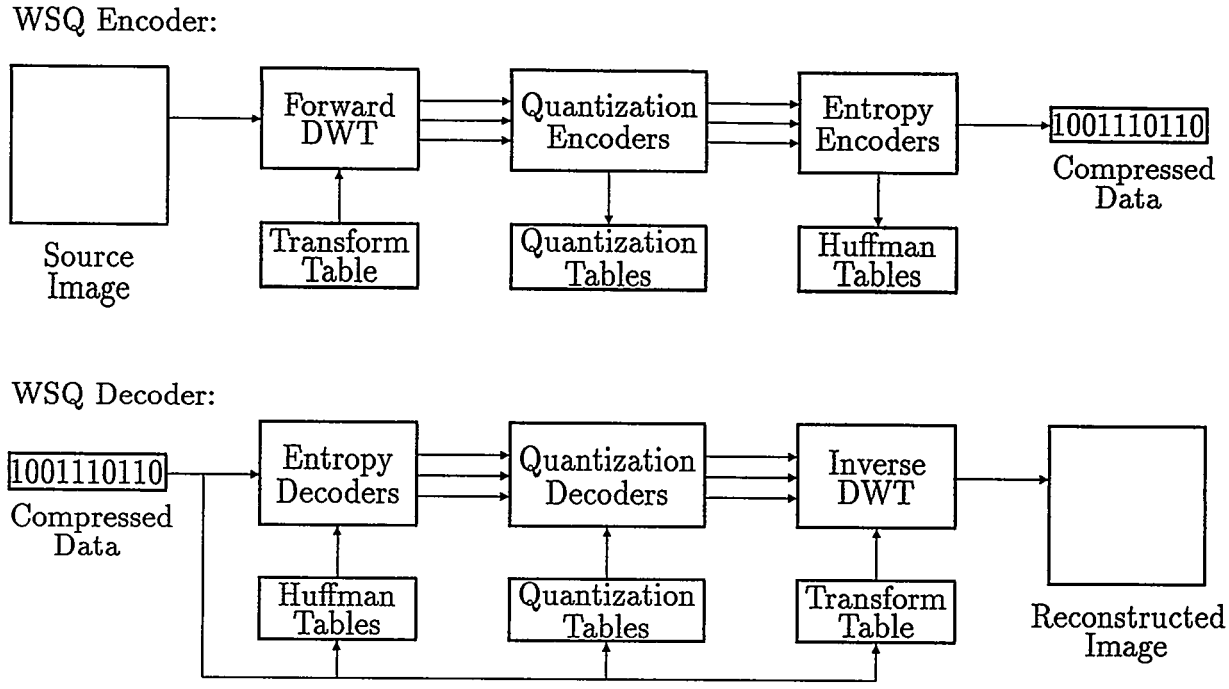

Figure 1: Simplified WSQ encoder and decoder diagrams.

four in the experiments described below). For image processing applications, it is necessary to specify how boundary conditions are to be handled when the input is a finite-duration signal, such as a row or column vector from a digitized image. This is handled with the symmetric extension method detailed in [14].

After the subband decomposition is computed, scalar quantization is performed on the subbands. This requires an initial bit allocation procedure that determines a bin-width for each scalar quantizer. A target bit rate is specified to constrain the final compression ratio, as described in [15, 16]. The bit allocation represents the solution to a constrained optimization problem: the bin widths are selected to minimize the quantization distortion subject to a prespecified overall bit rate. The distortion model used here is a weighted mean-square error of the image subbands and requires that the variance of each subband be calculated. The scalar quantization procedure involves mapping each wavelet coefficient to a quantization bin index; this requires a single multiplication for each coefficient. Scalar dequantization is a table lookup procedure.

Huffman coding involves a tabulation of the frequency of occurrence of each symbol (i.e., bin index). Based on the symbol probabilities a set of unique prefix code words is constructed. Huffman encoding consists of the substitution of appropriate code words for symbols in the data stream. Huffman decoding entails parsing the transmitted codeword tree, with a decision branch for each bit when decoding the encoded bit stream.

The accompanying prints give false-color visualizations of 768×768 arrays of original ocean surface height data and of data that has been compressed to 0.5 bpp by the WSQ method.

II-A. The Serial Code

The 1-D DWT in the serial code was realized with a direct form implementation [17]. This uses a sliding stencil that is shifted by two points at each iteration. Data management of the 2-D DWT is somewhat involved. At each level in the cascade, the subband to be split (i.e., the lowpass subband) is copied from the DWT array to an array referred to here as work array A. A second work array, work array B, is allocated having the same dimensions as the first. Each row of work array A is processed by the one-dimensional DWT and the results are stored in work array B. Work array B is then transposed and processed by a similar row-wise DWT that writes to work array A. Work array A is then inserted back into the DWT array. The transpositions are necessary since the one-dimensional DWT was not implemented to have data strides other than one.

Due to the serial nature of the implementation, the variance computation in each subband was realized with a C `for` statement, looping over the coefficients. After the bit allocation is determined the scalar quantization is performed serially: for each subband, a C `for` statement loops through all of the wavelet coefficients and assigns the appropriate bin index. This binning procedure involves a single floating point multiplication for each wavelet coefficient.

III. The A++/P++ Array Class Libraries.

A++ and P++ are serial and parallel C++ array class libraries, respectively, which support architecture independent development of numerical algorithms. The motivation for their use is both the simplified array syntax and the abstraction away from the machine architecture that comes from the high level array syntax. Both reasons are important since the resulting codes are intended to be run on a wide range of computers (e.g. Unix Workstations, Cray YMP, Cray C90, CM-5 (parallel machine), Intel Paragon, etc.), and insufficient resources are available for explicit development of separate versions on each machine. The array syntax greatly simplifies the development of structured grid iterative methods for the solution of partial differential equations (PDE) by eliminating error-prone programming tasks like explicit subscript usage and loop control. Additionally, the use of array classes in not limited to the solution of PDE's, though that has been the focus of research until now. Other difficulties encountered when programming parallel architectures, especially message passing problems, can also be largely abstracted away. Thus, the use of array objects for the development of numerical software allows programmers to implement more complex and ambitious numerical methods than would be possible using traditional serial programming. It also facilitates program execution on a wide range of high performance machines.

Performance remains an important consideration with the use of array classes, otherwise we can hardly justify their use on expensive machines. Finite difference computations, the original motivation for the A++/P++ array classes, arise commonly in the solution of PDE's; this paper presents an extension of array class techniques to another important area of scientific computation. We examine performance problems and analyze solutions in the context of array classes on two different architectures, restricting ourselves to the serial A++ array class library. Future work will address the performance of parallelized P++ subband coding implementations.

```
/****** Copy Low-Pass Filter into Array ******/

  I = Index (0, height, 1);

  for (k=0; k<low_filt_len; k++)
        H0(I, k) = h0(k);

/****** Low-Pass Filtering Along Rows of Y ******/

  J = Index (0, low_filt_len, 1);

  for (k=0; k < width/2; k++)
        X0(I, k) = sum( Y(I, J + 2*k) * H0 , 1 );
```

Figure 2: Examples of A++ array syntax implementing vectorized 1-D convolution.

III-A. The Array Class Code.

This summary concentrates on the two procedures that map well into the array class syntax; namely, the filter bank implementation and the scalar quantization. Entropy coding is proving to be more difficult to analyze in the array context because of its inherently serial nature. Unlike the serial case described in Section II-A, the filter bank is implemented as a 2-D array operation in the array class code. The basic principle is to write high-level syntax involving pointwise unary or binary operations on multidimensional array objects and functions defined on such objects. For the purposes of this paper, the concept is best illustrated by examples.

Figure 2 presents two simple operations implemented in array class syntax. The first involves duplicating a lowpass impulse response, `h0`, in each row of a 2-D array, `H0`. `I` is an *index object*, which can be thought of as a 1-D stencil of length `height`, base 0, and stride 1. It can be used to index an array object; e.g., the expression `H0(I, k)` represents a "view" of the k-th column of `H0`. The `for` loop copies `h0` into each row of `H0` (there are `height` many rows), one column at a time.

The next code sample applies `h0` to each row of a data array, `Y`, using `H0`. The expression `Y(I, J + 2*k)` represents a view of a subarray of size `height` $\times$ `low_filt_len`, which is the same size as `H0`, with a horizontal offset of 2k. The product `* H0` represents the pointwise product of this view of `Y` with `H0`, a result that is stored in a temporary array by the program. Note that the * operator has been *overloaded*; i.e., multiplication has been redefined to mean the pointwise (or Schur) product of two array objects. The unary function, `sum(` *tmp*`, 1 )`, sums *tmp* along the first coordinate direction, returning the inner products of `h0` with the 2k-th offsets of each row in `Y`. Syntactically, the lowpass 2-D subband array, `X0`, is being computed one column at a time.

Two points bear emphasizing about the array syntax. First, the details of evaluating the low-level loops implicit in the high-level syntax are hidden in the machine-dependent layer

of the array class library, along with the responsibility for performing these tasks efficiently on multiprocessor architectures when using a P++ library. Second, the high-level syntax conceals a great deal of temporary storage allocation and data copying. It is the cost of this hidden computational complexity that we now address.

IV. Numerical Results.

This section presents the results of timing experiments designed to compare serial and A++ implementations of the octave-scaled WSQ algorithm. Our goal is not to produce best-possible performance results on a single platform but rather to compare the performance of implementations with the same floating-point complexity and analyze the performance hit one takes as a result of mapping the algorithm into the high-level A++ array syntax. We could have obtained significantly better results for serial code performance on a workstation computer, for instance, by implementing the DWT using a reduced-complexity lattice factorization: the structure developed in [18] has only two-thirds as many multiplies as the direct-form implementation tested in this paper, as well as a more efficient use of machine registers. It is not at all clear, however, how to map such a structure into array operations, so we have decided (for now) to compare implementations based on conventional direct-form convolution. The ultimate goal is to develop readable, error-free software on a workstation using abstract, high-level syntax and then obtain decent performance on a parallel processing machine with a minimum amount of source code rewriting.

We compare a serial code implementation (written in C) with an A++ implementation on two vastly different hardware platforms: a 40 MHz Sun SparcStation IPX (Sparc 2 processor) with 32 MB of memory and a Cray Y-MP/M98. (The Y-MP/M98 has a couple gigabytes of memory.) Because the multiprocessor (P++) array class library is still being written for the Cray, all testing was conducted on a single Y-MP processor using the A++ library. Since no multiprocessor capabilities were being exploited, the same A++ library source code was used on both the Sparc and the Cray. Both the serial and A++ codes were compiled using what we found to be the best levels of optimization available. In particular, on the Cray we forced the compiler to ignore potential pointer aliasing in all loops and function parameters and to perform moderate vectorization.

The DWT and scalar quantization (SQ) procedures in the codes were timed independently by running each procedure through multiple iterations and bracketing each iteration loop with clock readings. The numbers of iterations were chosen to ensure that the granularity of the `clock()` function was negligible in comparison to the execution times, measured as total elapsed loop time and reported in seconds per iteration. No i/o time is included in the results presented below. Input data consisted of square 2-D arrays of ocean surface height data, as described in Section I. Input array sizes were $N = 32$, 64, 128, 256, 512, and 768. Only two levels of 2-D subband decomposition in the octave-scaled DWT algorithm were computed for array size 32; three levels of cascade were computed for size 64 and 4 levels for all other sizes.

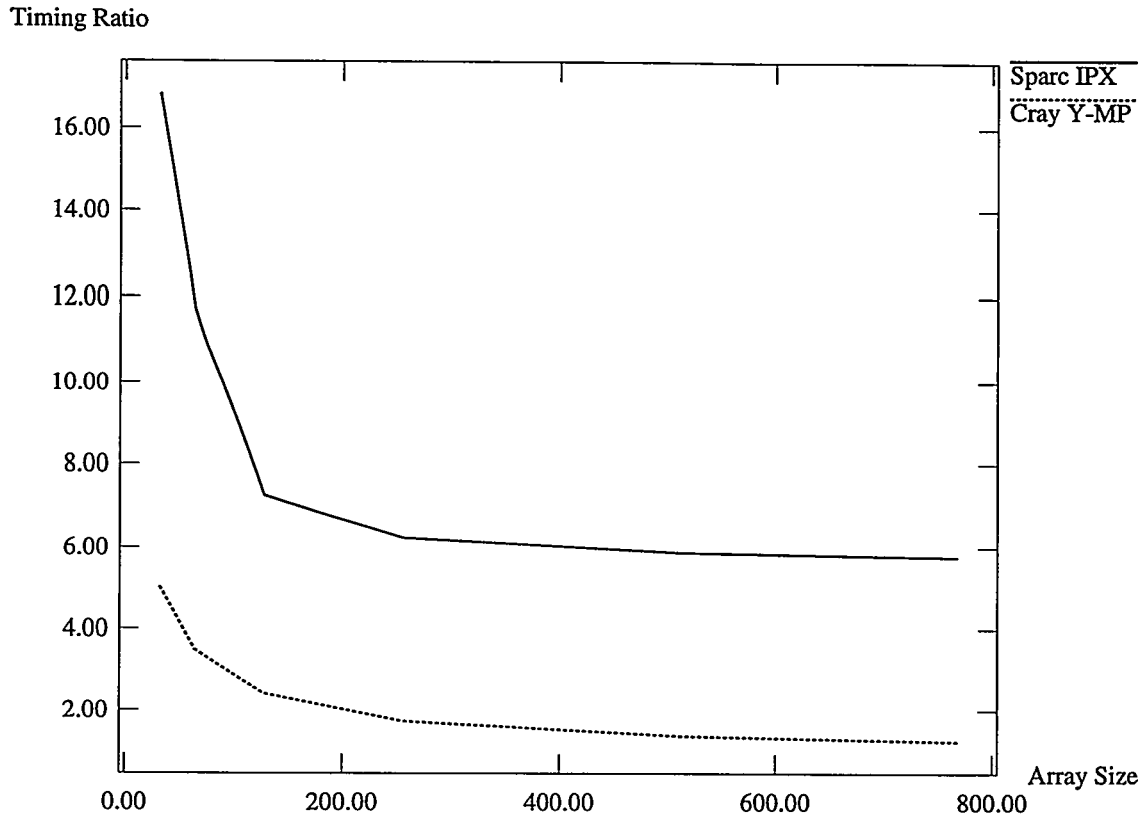

Figure 3: Discrete wavelet transform performance: ratio of A++ to C runtime.

IV-A. Performance comparisons: A++ vs. C.

Run-times for the DWT procedure on the Sparc using the serial implementation ranged from about 11 milliseconds per iteration for array size 32 to about 8 seconds/iteration for array size 768. Since we are primarily interested in comparing the performance of various implementations, all graphs display ratios of run times.

Figure 3 shows the ratio of A++ time to C time for the DWT on each platform as a function of array size. The cost of the library overhead is significant for small array sizes but seems to be amortized completely by array size 256. The asymptotic limit represents the relative expense of the specific 2-D DWT implementation described briefly in Figure 2 compared to the cost of the direct-form convolution employed in the serial C code. The poor asymptotic limit (about 5.75:1) on the Sparc 2 suggests that this particular A++ implementation is not competitive with the serial implementation, although a difference of less than a single order of magnitude is not enough to preclude running the A++ code on a workstation for purposes of software development and algorithm testing. Results obtained by replacing the innermost array operations in the A++ code (i.e., the inner products depicted in Figure 2) with explicit C loops indicates that roughly half of this performance disparity is due to the way the low-level loops are implemented in array computations; the other half is evidently due to higher-level library overhead. It will take a great deal of testing

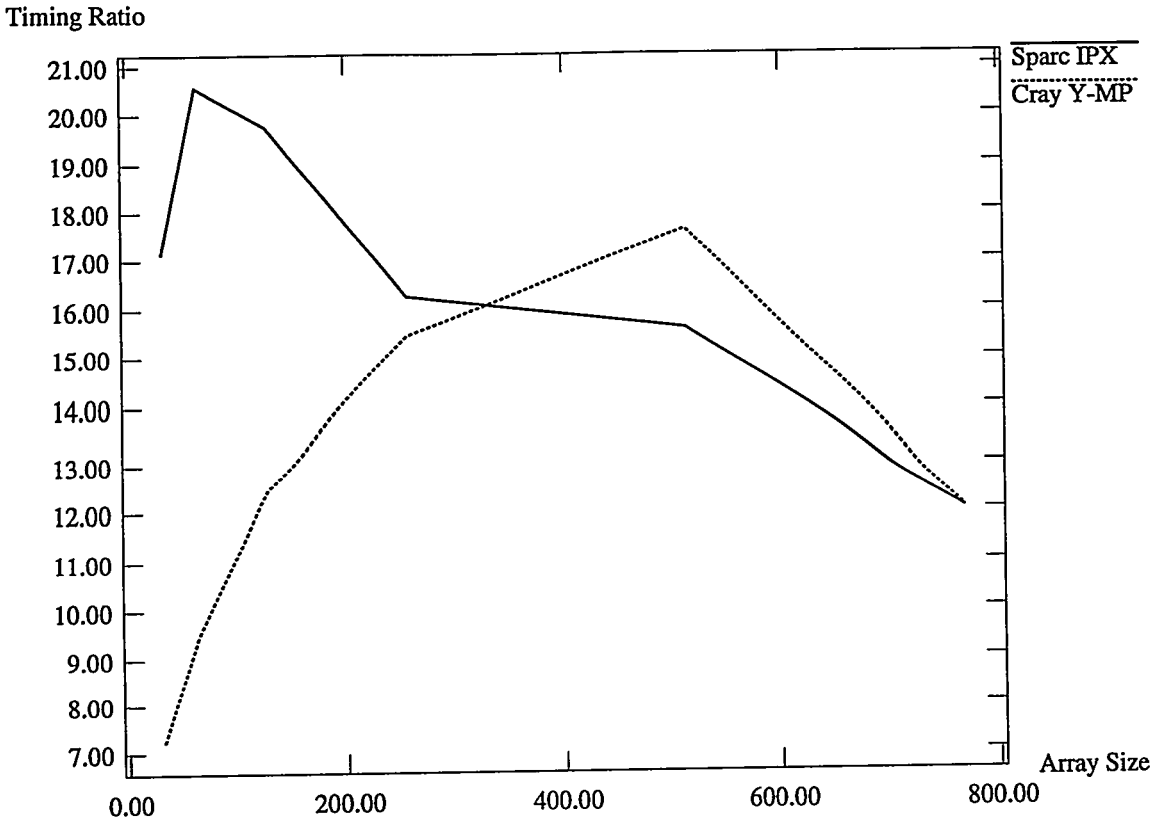

Figure 4: Scalar quantization performance: ratio of A++ to C runtime.

and rewriting of the A++ implementation to determine whether it is possible do better on this particular platform. One possibility is to replace the product filter bank approach used here with true 2-D stencil operations; this should map into array syntax much more efficiently and should compete better with serial product-filter implementations.

The comparison of DWT times on the Cray are much more encouraging for A++, with an apparent asymptotic limit of about 1.25 (a performance hit of only about 25%). Remember that this is the arena in which the array class library is intended to operate. In a moment we will argue that this performance is primarily attributable to superior vectorization of the A++ library code. First, however, let us look at the results for scalar quantization.

Figure 4 plots the ratio of A++ time to C time for the scalar quantization procedure, including the tasks of computing subband means and variances and determining the quantizer bin widths corresponding to an optimal bit allocation. The SQ comparison on the Sparc 2 is roughly similar to the performance of the DWT procedure, although it appears that the SQ performance ratio has not yet reached an asymptotic limit by array size 768. (The peak at array size 64 appears consistently in repeated timings:) Separate analysis indicates that a factor of about 4 in the A++ SQ performance is attributable to a poorly optimized A++ `where(`*inequality=TRUE*) statement that executes the pointwise evaluation of the inequalities involved in quantizing the DWT output arrays. These poor results using `where()` were unexpected since it has been little-used to date in PDE applications, but this will likely

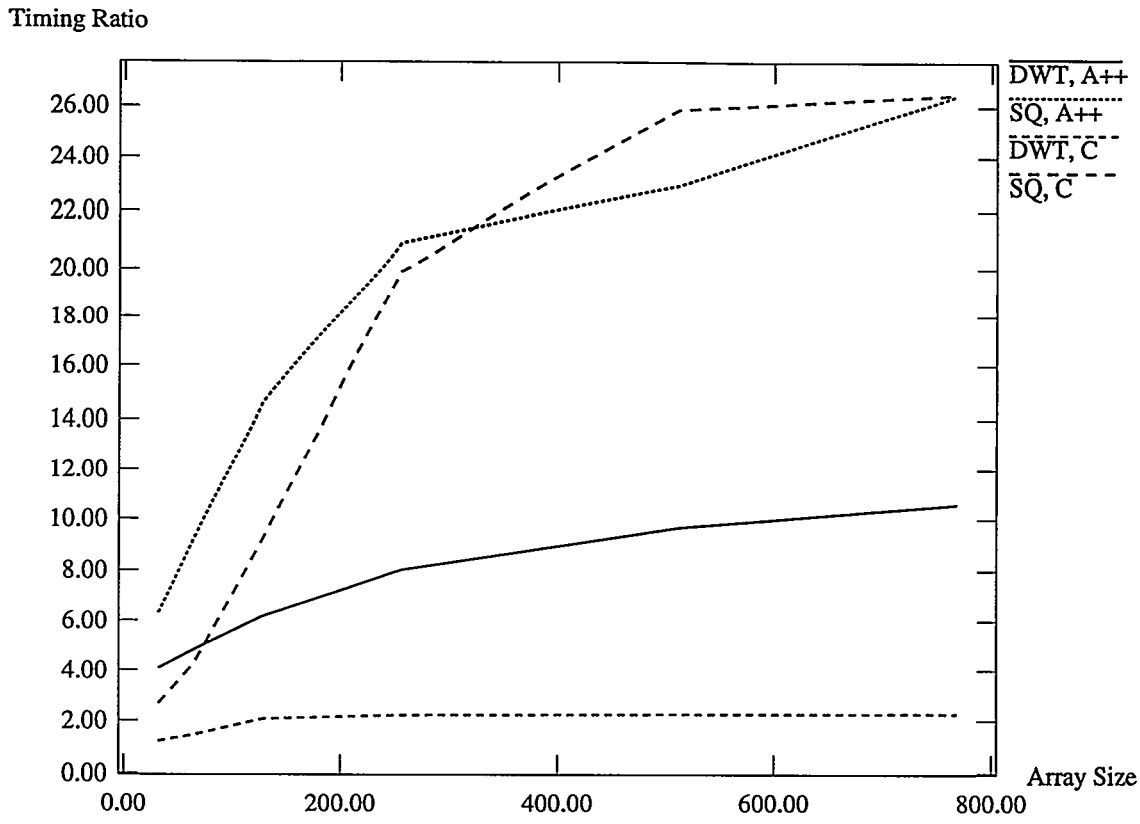

Figure 5: Vectorization performance: ratio of SparcStation to Cray runtime.

influence future refinements in the A++/P++ low-level implementation.

What is most striking, however, is the behavior of the SQ performance ratio on the Cray: here, the serial C code actually *improves* its performance advantage over A++ as array size increases, apparently peaking at array size 512. To understand this phenomenon, we analyze the effectiveness with which the Cray compilers are able to vectorize these codes.

IV-B. Vectorization performance.

To evaluate the extent to which the Cray C and C++ compilers are able to vectorize a given code, we next plot the ratio of Sparc run time to Cray run time as a function of array size. Code that vectorizes well will show improved relative performance at larger array sizes (since long loops vectorize better than short ones), while code that does not vectorize well will exhibit constant relative performance at all array sizes. Since the clock speed on the Y-MP/M98 is around 200 MHz (vs. 40 MHz on the Sparc 2), we expect a nominal speedup of around 5:1.

The graphs in Figure 5 present the speedup ratios for each of the four procedures being tested. Both of the A++ routines start at a speedup factor of around 5:1 and increase with the array size. Both SQ routines appear to get significantly more performance enhancement from vectorization than the DWT does. While the A++ DWT exhibits modest gains due to

vectorization with increasing array size, the C DWT gets essentially no performance boost from vectorization. Note, too, that the performance ratio for the C DWT is only around 2:1 rather than the nominal 5:1 speedup we might expect on the faster processor. The C DWT is using one function call for each 1-D convolution it performs, however, and this is undoubtedly responsible for much of the subpar performance. We expect rewriting the C DWT code to eliminate such calls would enhance Cray performance.

The SQ, on the other hand, is getting a great deal of performance enhancement from vectorization. Results compiled with un-coerced vectorization show a similar (though somewhat less dramatic) trend, indicating that scalar quantization is inherently highly vectorizable, as might be expected. Note that the serial C code shows a greater gain due to vectorization at array size 512 than does the A++ SQ. This accounts for the peak at size 512 in the Cray SQ performance ratio shown in Figure 4. From Figure 5, however, it appears that the serial SQ has reached a vectorization threshold by array size 768.

Since scalar quantization is well-suited for pointwise array operations, these results suggest that we should be able to get further improvements out of the A++ implementation of scalar quantization. More analysis is needed, however, to reveal the shortcomings of the current A++ implementation. While we do not yet fully understand why the A++ SQ is apparently failing to perform as well as the serial implementation, it appears that the use of several poorly optimized A++ functions, such as the `where()` statement, are responsible for most of the discrepancy not attributable to general library overhead.

References

[1] J. N. Bradley and C. M. Brislawn, "Wavelet transform–vector quantization compression of supercomputer ocean models," in *Proc. Data Compress. Conf.*, (Snowbird, UT), pp. 224–233, IEEE Computer Soc., Mar. 1993.

[2] J. N. Bradley and C. M. Brislawn, "Applications of wavelet-based compression to multidimensional earth science data," in *Proc. Space Earth Science Data Compress. Workshop*, no. 3191 in NASA Conf. Pub., (Snowbird, UT), pp. 13–24, Nat'l. Aeronaut. Space Admin., Apr. 1993.

[3] R. D. Smith, J. K. Dukowicz, and R. C. Malone, "Parallel ocean general circulation modeling," *Physica D*, vol. 60, pp. 38–61, 1992.

[4] Federal Bureau of Investigation, *WSQ Gray-Scale Fingerprint Image Compression Specification*, IAFIS-IC-0110v2, Feb. 1993. Drafted by T. Hopper, C. Brislawn, and J. Bradley.

[5] T. Hopper and F. Preston, "Compression of grey-scale fingerprint images," in *Proc. Data Compress. Conf.*, (Snowbird, UT), pp. 309–318, IEEE Computer Soc., Mar. 1992.

[6] D. J. Quinlan, *Adaptive mesh refinement for distributed parallel architectures.* PhD thesis, Univ. of Colorado—Denver, Denver, CO, 1993.

[7] M. Lemke and D. J. Quinlan, "P++, a C++ virtual shared grids based programming environment for architecture-independent development of structured grid applications,"

Arbeitspapiere der GMD 611, Gesellschaft fur Mathematik und Datenverarbeitung, St. Augustin, Germany, Feb. 1992.

[8] J. K. Dukowicz and R. D. Smith, "Implicit free-surface method for the Bryan-Cox-Semtner ocean model," *J. Geophys. Res.*, vol. 99, pp. 7991–8014, Apr. 1994.

[9] N. S. Jayant and P. Noll, *Digital Coding of Waveforms.* Englewood Cliffs, NJ: Prentice Hall, 1984.

[10] M. Rabbani and P. W. Jones, *Digital Image Compression Techniques.* Bellingham, WA: Soc. Photo-Opt. Instrument. Engin., 1991.

[11] A. Gersho and R. M. Gray, *Vector Quantization and Signal Compression.* Norwell, MA: Kluwer, 1992.

[12] J. N. Bradley, C. M. Brislawn, J. E. Brown, C. A. Rodriguez, and L. A. Stoltz, "Video imaging for nuclear safeguards," in *Proc. Data Compress. Conf. Industry Workshop* (R. L. Renner, ed.), (Snowbird, UT), TRW Corp., Apr. 1994.

[13] I. C. Daubechies, *Ten Lectures on Wavelets.* No. 61 in CBMS-NSF Regional Conf. Series in Appl. Math., (Univ. Mass.—Lowell, June 1990), Philadelphia, PA: Soc. Indust. Appl. Math., 1992.

[14] C. M. Brislawn, "Classification of nonexpansive symmetric extension transforms for multirate filter banks," Tech. Rep. LA-UR-94-1747, Los Alamos Nat'l. Lab, May 1994. Revised draft of LA-UR-92-2823 (Aug. 1992). Still under review for publication.

[15] J. N. Bradley and C. M. Brislawn, "Proposed first-generation WSQ bit allocation procedure," in *Proc. Symp. Criminal Justice Info. Services Tech.*, (Gaithersburg, MD), pp. C11–C17, Federal Bureau of Investigation, Sept. 1993.

[16] J. N. Bradley and C. M. Brislawn, "The wavelet/scalar quantization compression standard for digital fingerprint images," in *Proc. Int'l. Symp. Circuits Systems*, vol. 3, (London, England), pp. 205–208, IEEE Circuits Systems Soc., June 1994.

[17] R. E. Crochiere and L. R. Rabiner, *Multirate Digital Signal Processing.* Englewood Cliffs, NJ: Prentice Hall, 1983.

[18] C. M. Brislawn, "A simple lattice architecture for even order linear phase perfect reconstruction filter banks," in *Proc. Int'l. Symp. Time-Freq. Time-Scale Analysis*, (Philadelphia, PA), pp. 124–127, IEEE Signal Process. Soc., Oct. 1994.