SPECIAL REPORT

1988 Gordon Bell Prize

Jim Browne, Jack Dongarra, Alan Karp, Ken Kennedy, and *Dave Kuck*

This year's winners tackled a tough problem — handling static structures — yet were able to achieve an impressive speedup.

The Gordon Bell Prize recognizes outstanding achievement in the application of supercomputers to scientific and engineering problems. In 1988, two prizes were offered in three categories: raw performance, price/performance, and compiler parallelizations.

Raw performance means that the problem could not be solved in less time on any other computer using any other method.

The price/performance category recognizes the growing use of minisupercomputers and parallel systems made up of hundreds or thousands of inexpensive processors. The winning entrant is expected to demonstrate that no one could have solved the same problem at less cost, provided the computer meets a minimum performance standard.

The compiler parallelization category is intended to encourage the development of compilers smart enough to make parallelization as easy (relatively speaking) as vectorization. In all cases, the entrants had to prove to the judges that they indeed had the best entry.

Three of this year's four entries were deemed to have met the rules for this year's prize. By coincidence, there was exactly one entry in each category. As has become customary with the Bell Prize, the judges do not let the rules stand in the way of giving prizes. Therefore, one winner and two honorable mentions were awarded.

Gordon Bell, vice president of engineering at Ardent Computer in Sunnyvale, Calif., offered in 1986 to sponsor two

Browne, Dongarra, Karp, Kennedy, and Kuck judged the 1988 Gordon Bell Prize. Browne is a professor of computer science at the University of Texas at Austin and chief scientist at Information Research Associates. Dongarra is chief of parallel-processing research at the Mathematics and Computer Science Division of the Argonne National Laboratory in Illinois. Karp, who chaired the judging committee, is a staff member at the IBM Palo Alto (Calif.) Scientific Center who is on assignment at the IBM Washington Systems Center in Gaithersburg, Md. Kennedy is a professor of computer science at Rice University in Houston. Kuck is president of Kuck & Associates in Savoy, Ill.

0740-7459/89/0500/0078/$01.00 © 1989 IEEE

$1,000 awards each year for 10 years to promote practical parallel-processing research. This marks the second annual award of the prize, which had been called the Gordon Bell Award last year. *IEEE Software* administered the awards and judging. The winners were announced Feb. 28 at the Computer Society's Compcon conference in San Francisco.

This year's winner of the Bell Prize, from the raw-performance category, is a team made up of Phuong Vu of Cray Research, Horst Simon of the National Aeronautics and Space Administration's Ames Research Center, Cleve Ashcraft of Yale University, Roger Grimes and John Lewis of Boeing Computer Services, and Barry Peyton of Oak Ridge National Laboratory. They presented the solution of a static-structures problem that ran at just over 1 Gflop on an eight-processor Cray Y-MP.

The judges felt that both remaining entries represented important work but failed to meet the requirements for a prize.

The price/performance entry came from Richard Pelz, who used a 1,024-processor N-Cube multicomputer to solve a fluid-flow problem using a spectral method with a speedup of about 800. Although this result was better than many would have predicted for a spectral method, it did not beat the performance of last year's winner, whose speedups ranged from 400 to more than 1,000 when scaled (see "First Gordon Bell Awards Winners Achieve Speedup of 400," Soft News, *IEEE Software*, May 1988, pp. 108-112).

Marina Chen, Young-il Choo, Jungke Li, and Janet Wu of Yale University and Eric DeBenedictus of Ansoft Corp. submitted an entry in which a Crystal Compiler automatically parallelized a financial modeling application. This application sped up by about 350 times on a 1,024-processor N-Cube and more than 50 times on a 64-processor Intel iPSC-2. While this parallelism was detected automatically by the compiler, the judges felt that the application was too nearly parallel in nature, making a high speedup all but certain.

First place

The winning entry ran on an eight-processor Cray Y-MP with a 6.49 nanosecond cycle time. The reported speed of 1 Gflop is almost 40 percent of the machine's theoretical peak. In fact, if you consider only the computational part of the code, the winners achieved almost 65 percent of the peak speed. This high a ratio is not easy to achieve on a problem as hard as the one submitted.

The winning team's runtime was reduced from almost 15 minutes to less than 30 seconds, turning a batch job into an interactive one.

The winning team ran a static finite-element analysis. Such analysis has many important applications. For example, a group at NASA Langley Research Center has been using a commercial finite-element package to model the space shuttle on a Cray 2. The winning team was able to solve the same problem nine times faster on the Cray 2, 20 times faster on a single processor of a Y-MP, and 32 times faster on an eight-processor Y-MP. The runtime was reduced from almost 15 minutes to less than 30 seconds, turning a batch job into an interactive one. The judges commended the winning team for demonstrating the gains to be made from both hardware and algorithmic improvements.

Finite-element methods work by dividing up the item being modeled into small pieces — line segments in one dimension, triangles or quadrilaterals in two dimensions, and cylinders or tetrahedra in three dimensions. The solution in each element is approximated by a low-order polynomial; for example, $a+bx+cy$ in two dimensions. The unknown coefficients are found by forcing the solution to be continuous where the elements intersect.

In one dimension, the elements are line segments, and the simplest representation for the solution is a set of piecewise linear functions. If you want to find the positions of masses connected by springs as illustrated in Figure 1, you equate the forces from both sides of the mass. An example is $k_3(x_2-x_3) = k_4(x_4-x_3)$, where the k's are the spring constants and the x's are the unknown positions of the masses. You get one such equation for each interval. There will be two fewer equations than unknowns, but you have two boundary conditions that you are free to specify. Because each equation contains three unknowns, you must solve a system of linear equations $Ax = b$, which in this case is tridiagonal.

You can do the same thing in two dimensions, but the resulting linear system will be more complicated. As Figure 2 shows, the equation giving the forces on mass X depends on the positions of A, B, C, and D. If you number the masses by rows, the matrix will have the structure shown in Figure 3. The nonzero elements are represented by an x, and zero entries are not shown. This banded matrix consists of tridiagonal matrixes and bands five elements off the diagonal.

Although iterative methods are sometimes used to solve such problems, finite-element codes normally use a direct method: LU decomposition using Gauss elimination. If you solve this system of equations with Gauss elimination, most of the zeroes between the bands will become nonzero and the work you must do will be

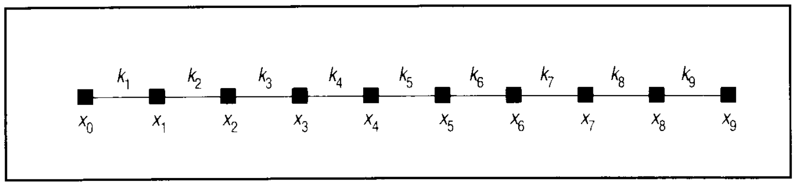

Figure 1. A set of piecewise linear functions for masses connected by springs shows the simplest representation for the solution to the Bell Prize winners' problem.

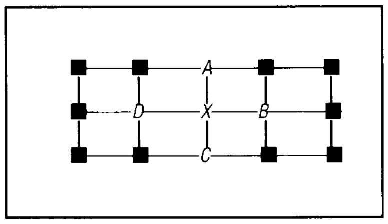

Figure 2. The equation giving the forces on mass X depends on the positions of A, B, C, and D.

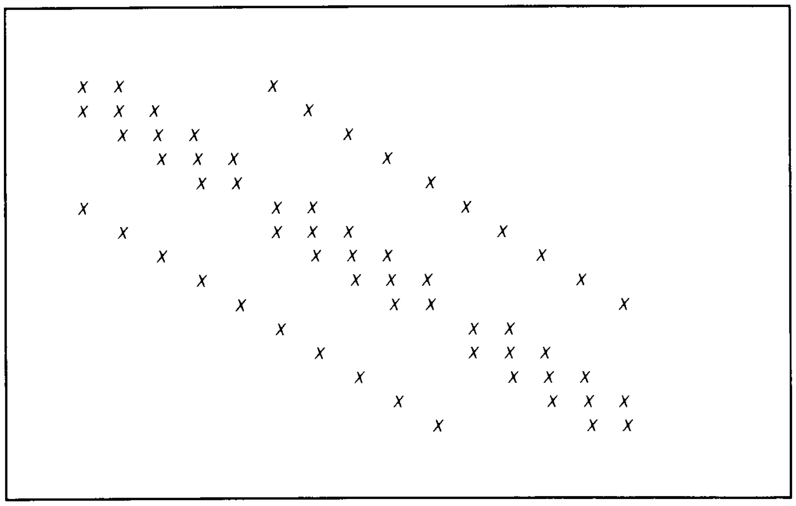

Figure 3. The matrix for Figure 2 when the masses are numbered by rows.

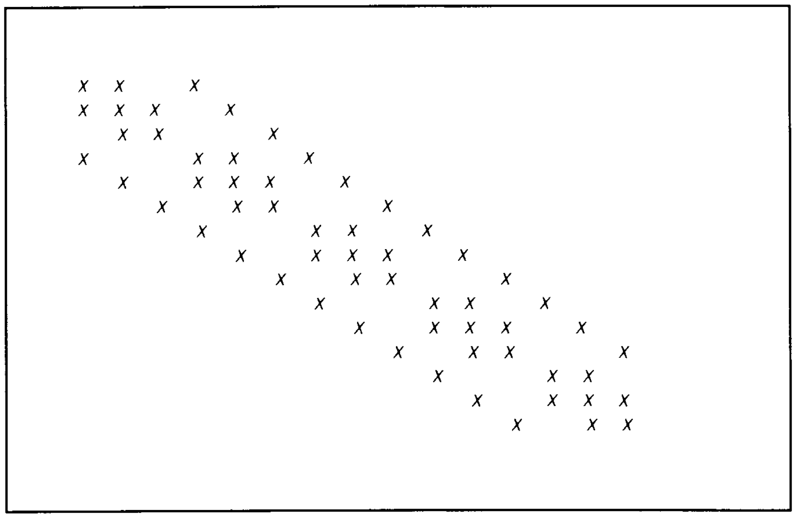

Figure 4. The matrix for Figure 2 when the masses are numbered by columns.

proportional to nm^2, which in this case is $15{*}5^2$. If you number by columns, you get the matrix in Figure 4, which has a bandwidth of only three, so the work is only $15{*}3^2$ — about one third as much work. Clearly, numbering the elements properly can make a big difference in the algorithm's performance.

Real structures are not so simple. For example, the coefficient matrix for a two-dimensional model of the bridge in Figure 5 will not have the simple structure of the previous examples, but it will still have mostly zeroes.

While the numbering scheme can greatly affect the amount of calculation that needs to be done, there is another, often conflicting constraint: the computer's architecture. The Cray Y-MP runs about 25 times faster in vector mode than in scalar mode. There is often a trade-off between operation counts and vectorization. How should the elements be numbered to optimize the solution?

One approach is to number the elements to move as many zeroes as possible outside the band. This method is called skyline ordering because the upper part of the matrix looks like a big-city skyline, as Figure 6 shows. This method's vectorization is good because it has you operate on all elements from the furthest out in each row or column to the diagonal. Unfortunately, you do more arithmetic than needed because there are many zeroes inside the skyline and they become nonzero during the reduction. Thus, skyline methods often achieve very high megaflop rates but run slower than the alternative sparse-solver routines.

Another approach is to number the elements to minimize the amount of fill-in (the number of zeroes that become nonzero during the Gauss elimination). The number of possible orderings is enormous, so as a practical matter you can never find the minimum fill-in ordering. Instead, you do the best job you can using heuristics. The method chosen by the winning team is a minimum-degree ordering. As the reduction proceeds, you choose the next column to eliminate to be the one that will produce the least fill-in.

If you number the nodes according to this scheme, the matrix does not have any apparent structure; nonzeroes appear to be sprinkled around randomly. If you are to use the sparsity in the matrix, you must treat each individual nonzero independently. For example, if you want to eliminate element (i,j), you form a linear combination of rows i and j. If these rows are mainly zeroes, you should do the arithmetic on only the nonzero elements. You keep track of these nonzeroes using an index vector for each row. Thus, for dense matrixes, this linear combination might be coded

```
   DO 10 K = 1, M
10 A(I,K) = A(I,K) + X*A(J,K)
```

while for sparse matrixes you could write

```
   DO 20 K = 1, M(I)
20 A(I,INDX(K)) = A(I,INDX(K)) +
     X*A(J,INDX(K))
```

The array Indx contains an entry for each index that is nonzero in either row I or row J, and $M(I)$ is the number of nonzeroes in row I after the linear combination has been formed.

While this approach reduces the number of arithmetic operations, it can be extremely difficult to vectorize on computers without indirect addressing (gather/scatter) instructions. Even if gather/scatter is available, performance can be dramatically lower than using simple indexing. On a Cray X-MP, loop 10 (which addresses data contiguously) runs at almost 200 Mflops, while loop 20 (which uses indirect addressing) runs at less than 80 Mflops.

The method used by the winning team uses an idea called supernodes. In the bridge in Figure 5, the left vertical support below the bridge is connected to the rest of the structure at a single point. Thus, you can number the elements to reflect the fact that you can almost solve for this support independently of the rest of the structure and plug it in later. In other words, you take many finite elements (nodes) and group them into a supernode.

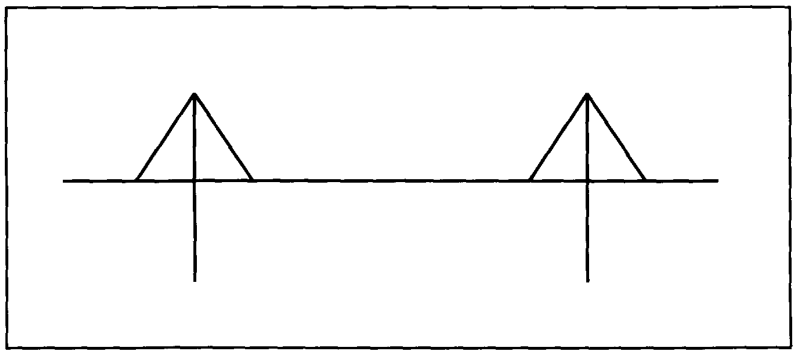

Figure 5. Two-dimensional model of a bridge. Such real-world models are more difficult to apply finite-element modeling techniques to.

$$\begin{array}{cccccccccc}
x & x & & & & & & & & \\
x & x & & x & & x & & & & \\
 & & x & x & & 0 & x & & & \\
 & & x & x & & 0 & x & & x & \\
 & & & & x & 0 & 0 & & x & \\
 & x & 0 & 0 & 0 & x & x & & 0 & x \\
 & & x & x & 0 & x & x & x & x & 0 \\
 & & & & & & x & x & x & x \\
 & & & x & 0 & x & x & x & x & x \\
 & & & & & x & 0 & x & x & x
\end{array}$$

Figure 6. Vectorization of the skyline-ordering method, which seeks to number the elements to move as many zeroes as possible outside the band.

This numbering may not produce as narrow a bandwidth as the optimal scheme. To keep the number of operations small, you should number the nodes according to this heuristic. Fortunately, as the Gauss elimination proceeds, the locations of the nonzero elements in all the columns corresponding to a supernode will be similar. This behavior lets you use one gather operation, operate on many columns of the matrix, and then do one scatter. Thus, almost all the work is done in a mode optimal for the hardware.

The good vectorization achieved by the winning team is demonstrated by the high uniprocessor computation rate of more than 250 Mflops. The rate counting only the computational kernel was even higher: 280 Mflops. The team implemented parallelism with autotasking, a compiler option that automatically parallelizes loops in the program. The reported speed of 1 Gflop with eight processors indicates that about 85 percent of the time was spent doing parallel work. If you only look at the computational kernel, the rate is even higher — more than 1.5 Gflops — and the code is almost 95-percent parallel.

Greatly speeding up programs does create a problem: Parts that used to be such a small part of the whole that they weren't worth optimizing now take a significant amount of time. On the space-shuttle model mentioned earlier, the decomposition step took most of the 800 seconds on the original Cray 2 version but only eight out of 26 seconds on the new parallel version.

The winning team got bitten by this bug on its original submission. Although the decomposition ran at 1.5 Gflops, the overall performance of the code was less than 450 Mflops. The input phase that the team had never bothered to optimize was taking 31 seconds out of a total run of 73 seconds. The judges felt that the only number of interest was the speed of the code as a whole and that 450 Mflops did not beat last year's winner. Once we made the winning team aware of our concern, it optimized the input and reduced the runtime to only two seconds.

Honorable mentions

The judges awarded two honorable mentions: one to Rutgers University's Pelz and one to the Yale/Ansoft team of Chen, Choo, Li, Wu, and DeBenedictus.

Pelz. Pelz parallelized a fluid-flow application on a 1,024-processor N-Cube hypercube parallel processor, which is the same problem last year's winners from Sandia National Laboratories submitted and for which they achieved near-perfect speedup. But while the Sandia team used an explicit method, Pelz achieved his speedup with an implicit one.

The Sandia team used explicit finite differences, which consist of dividing the domain up into small pieces, approximating the space derivatives by differences, and computing the unknowns at the new time step using only values at the old time step. Although such explicit methods work for some problems, the step size needed to achieve a good level of accuracy can be so short that it makes them impractical. Pelz used an implicit spectral method that can be used when other methods fail. The question his research asked was "How well do spectral methods parallelize?"

The application models an infinite medium in which the flow is periodic in all three dimensions. This infinite fluid can be modeled as a cube with periodic boundary conditions (where the values at the left and right boundaries are forced to be the same, and similarly for the other edges). If the flow were in two dimensions, you could picture it taking place on the surface of a torus.

This model can be used to study the evolution of turbulence. Take a periodic-boundary cube, stir things up with an egg beater, and then watch the turbulent eddies decay. From this simulation, you learn how energy is transported from large-scale eddies to small-scale eddies, how small eddies return energy to larger eddies, and what the additional energy losses caused by turbulence are — the so-called eddy viscosity.

These results are used in practical applications. Someone simulating an airplane in flight cannot have a grid fine enough to resolve all the turbulence. To accurately model such things as lift and drag, the turbulence occurring at scales smaller than the grid is parameterized using data from calculations like those described above.

Pelz tackled a fluid-flow problem similar to the one that last year's winners addressed. But he used an implicit method that can be used when other methods fail.

Once you have chosen the model equations, the incompressible Navier-Stokes equation, and the boundary conditions — periodic in three dimensions — the entire behavior of the model is determined by the initial conditions. Pelz chose to use the Taylor-Green vortex and the Arnold, Beltrami, Childress initial conditions to test his code.

The Taylor-Green vortex is useful because it maintains certain symmetry properties over time. For example, each of the eight quadrants of the periodic-boundary cube should behave identically over time. If the numerical method is accurate, you can use the deviation from symmetry as a measure of the accuracy. Of course, you can choose the numerical method to enforce the symmetries, but that would defeat the use of this problem to test the numerical method.

In the ABC initial conditions, the flow is such that the nonlinear term in the Navier-Stokes equations vanishes. If an amount of energy is added through the forcing term to exactly balance the energy lost to viscosity, the solution with ABC initial conditions should be a steady flow with one wave of fixed size. Any numerical errors will show up as other waves in the solution. You can use the amplitude of these aliased solutions to check the numerical method's accuracy.

Because the methods used by last year's winners do not work well for these problems, Pelz tried using a pseudospectral implicit method. Finite-difference methods approximate the derivatives by differences between closely spaced grid points; finite elements approximate the solution on each small element by a known function with unknown coefficients. Spectral methods try to approximate the solution everywhere by a single known function with unknown coefficients. For example, you could use

$$f(x,t) = \sum_{k=0}^{N-1} a_k(t)\, x^k$$

for a one-dimensional problem. If you plug this approximation into your differential equation, you can do all the space derivatives immediately, but now you have only one equation for *N* unknowns: the *a*'s.

You can generate more equations by multiplying your differential equation, with your representation substituted for the unknowns, by powers of *x* and integrating over the domain. Each moment of the equation gives an independent equation, so taking *N* moments lets you find the coefficients. But there are two problems with using powers of *x* as your approximating function. First, the system of linear equations is numerically unstable; the system is ill-conditioned. Second, the coefficient matrix is dense, which means the solution takes a time proportional to N^3.

One class of functions avoids both these problems: the orthogonal functions. The most familiar set are sine and cosine. For example, if you integrate $\sin(ax)$ times $\sin(bx)$ from $-\pi$ to π, the result will be identically zero unless a equals b. Therefore, if you use $\sin(kx)$ to represent the solution, the coefficient matrix will be diagonal when you take the moments of the equation by multiplying by $\sin(mx)$.

There is another advantage to using sines and cosines to represent the solution. Some functions, like the forcing terms, must be multiplied by the appropriate function and integrated over the domain. Because it is unlikely that these integrals can be done analytically, you must devise a numerical quadrature scheme for these terms. However, if you use sines and cosines, these terms are just the Fourier transforms of the functions and you can use the efficient fast-Fourier-transform algorithm to evaluate them. This is the approach that Pelz took.

The model problems assume a noncompressible fluid. The only unknown is the velocity vector u that has three components for the three-dimensional problems solved. The Fourier transform of u is also a three-element vector U. The velocity vector u is in physical space and the three-element vector U is in spectral space. (The word "spectral" comes from the use of Fourier transforms to convert a time series into a frequency spectrum.)

Some terms, such as the space derivatives, are easily computed in spectral space. In fact, because you have the derivative as an analytic function, you can evaluate it at each point independently of all the other functions. You say that the derivative is a point operation in spectral space. Other terms, such as the computation of rotation, are the product of two functions — they are point operations — in physical space. Thus, all operations can be made local in either physical or spectral space. The advantage of point operations is that you can do them in parallel with no communications overhead; the speedup is perfectly linear to the number of processors.

Of course, this simplicity is not achieved without some cost: You must do multidimensional Fourier transforms. In three dimensions, you can compute a variable's Fourier transform by computing the fast Fourier transform first in the x direction, then in the y direction, and finally in the z direction. Although the spectral method used involves continuous functions, you must discretize it to use the fast Fourier transform. The trick is to divide the domain so each processor of the hypercube does as much computing and as little communicating as possible.

Consider first an $N \times N \times N$ grid and a machine with N processors. If you divide the grid so that each processor has exactly one $N \times N$ layer of the three-dimensional space, you can do the x and y fast Fourier transforms completely independently with no communication. Only when you go to do the fast Fourier transform in the z direction do the processors need to share data.

Pelz achieved a very good price/performance ratio. While not quite as fast as a Cray X-MP or Cyber 205, the N-Cube/10 he used costs only a fifth as much.

There are two approaches you can take. First, you can transpose the data to bring all the z values for a given x and y into a single processor. This step is pure communication, but it lets the fast Fourier transform on z proceed at full speed. Second, you can compute the fast Fourier transform, exchanging data among processors as needed. This approach was the one that Pelz submitted.

There is a further complication that Pelz had to deal with: The largest array that could fit in the memory of one processor of his N-Cube is 64×64, while the 1,024-processor machine can hold a $128 \times 128 \times 128$ problem. To solve this large a problem, Pelz had to distribute the data in two dimensions. He could do only the x transforms with perfect speedup; the y and z transforms both had to be done with the distributed fast Fourier transform. Thus, the speedups he reported would have been even higher had he been able to hold an entire plane in the memory of each processor.

The speedup of his largest problem on the N-Cube is an impressive 800, which translates to about 65 Mflops. Pelz also achieved a very good price/performance ratio. The N-Cube/10 he used costs about $1 million. His application ran in 25 seconds of elapsed time compared to 16 seconds of CPU time on a Cray X-MP and 25 seconds of CPU time on a Cyber 205. Each of these processors costs at least $5 million. And these times underestimate the relative performance of Pelz's code, since both the Cray and Cyber codes are highly optimized to use special hardware features of the machines while the N-Cube code is pure Fortran. In addition, the elapsed times on the Cray and Cyber would be substantially greater than the CPU times even if there were no other users on them.

Yale/Ansoft team. The Yale/Ansoft team parallelized a financial application for two distributed-memory machines, a 512-processor N-Cube and a 64-processor iPSC-2. Each machine is made up of independent computers interconnected in a hypercube. They differ primarily in the ratio of computation speed to communication speed and available memory.

The application parallelized helps banks determine the price to charge for packages of mortgages they sell on the secondary market. When you buy a house, you normally get a loan from a bank. These days, the banks do not hold mortgages for the full term. Instead, they bundle them into packages and sell them to investors. The problem run by the Yale/Ansoft team tries to determine a fair price for such a package of mortgages.

Consider a mortgage with a face value of $100,000 amortized over 20 years at an interest rate of 12 percent. The first month's payment will be $1,000 for interest plus about $100 to reduce the principal. The next month, the $1,100 payment will be divided up into $999 for interest and $101 for amortization. It is a simple matter to determine the income generated by such a mortgage and a fair price on the secondary market if the mortgage is not paid off early. Unfortunately for the banks (or fortunately for the mathematicians they hire), most mortgages are paid off early. Because the early payments are nearly all interest and the late payments are nearly all principal, the fair price for the mortgage will depend on when it is paid off.

Clearly, no one can accurately predict at the time a mortgage is granted how soon it will be paid off. However, if you lump a thousand such mortgages together, you should be able to do a pretty good job of guessing the average behavior of the mortgage holders. There is a complication, though: If interest rates plunge from 12 percent to 6 percent, most of the mortgage holders will refinance. In other words, outside economic conditions in the future can affect the fair price of the mortgage today.

The situation in the real world is worse. Many mortgages issued these days do not have a fixed interest rate; the rate varies with some index, often the cost of US Treasury notes. If interest rates go up, the mortgage payments increase, and the value of the mortgage to its holder increases. If rates fall, the mortgage is worth less.

To take all these factors into account, financial experts simulate the performance of many mortgages. Some factors, such as rates of household formation and deaths, can be reliably forecast using the historical record. Other factors, such as the rate of job transfers, cannot because they depend heavily on economic conditions. Even worse is the job of predicting interest rates 20 or more years into the future. While no one could have predicted the Arab oil embargoes of the 1970s or the stock-market crash of October 1987, each expert believes that he knows how interest rates will vary.

The fair price for a package of mortgages is determined by simulating the payments on the mortgages for a given interest-rate scenario using a Monte Carlo method. Monte Carlo methods use random numbers to simulate the range of behaviors expected. For example, if interest rates remain constant, some mortgages will be paid off each month. Clearly, relatively few people pay off the mortgage in its first month, but a big winner in the lottery might. If, on the other hand, interest rates drop precipitously, many people will refinance in a short period of time. The distribution of random numbers is chosen to reflect this kind of behavior. The average of many random runs, called trials, returns the fair price for a given interest rate scenario.

This independence of runs is the reason the Yale/Ansoft entry was disqualified from winning a prize. The judges consider a job to be embarrassingly parallel if it could be run on several independent computers that communicate only at the very start and very end of the job. The mortgage-pricing algorithm submitted can be written to be embarrassingly parallel.

The Yale/Ansoft team's entry was disqualified for a first prize because its problem was nearly embarrassingly parallel. But the Crystal compiler used goes a long way in proving the utility of functional languages.

But there are two reasons the Yale/Ansoft team won an honorable mention despite the obvious parallelism. First, they simply wrote down the equations in their Crystal language and let the compiler find the parallelism. Second, they wrote their code to collect the results of each run in a single, global array. Each processor ran many trials. Instead of collecting the statistics in each processor and sending the results at the end — an approach that would have been disqualified as embarrassingly parallel — the Yale/Ansoft code sent the results to one node for averaging at the end of each trial.

The Yale/Ansoft entry presented the judges with another problem. When this year's rules were published, we were looking for compilers working on conventional languages. We were not prepared to deal with an entirely new language. The Yale/Ansoft team coded the application in the functional language Crystal. It is clear that a Fortran or C program does not explicitly describe the parallelism inherent in the application. But what about Crystal? Has the programmer already done the work of finding the parallelism simply by coding in Crystal?

Functional languages are very different from conventional languages. In a conventional language, you describe both what you want to do and how you are going to do it, including the order in which things will be done. For example, if you want to find out whether an integer, say 1,001, is a prime using C, you could code p = isprime(1001) with

```
isprime(n)
  int n;
{
  f ( !n%2 ) return 0;
  for ( k = 3; k <= sqrt(n)+1; k += 2 )
    if ( !n%k ) return 0;
  return 1;
}
```

In a functional language, everything is represented as a function. In this example, you would say in a functional language

```
p = isprime(1001)
where
  isprime(n) = andof(rem(n))
  where
    rem(i) = 0 if n%index(sqrt(n)+1) = 0
      1 otherwise
  where
    index(n) = 2, 3 ... n by 2
```

The Andof function ands all instances of its arguments while Rem returns a 0 or 1 for all instances of its arguments. It isn't until you get down to index that you find out what the instances of the arguments are: 2 and all the odd integers up to *n*.

This simple example shows two significant differences between functional and conventional languages. First, you have not really said in what order to compute the entries in the functions, only that you want them computed. This independence of sequence is one reason functional languages are attractive for parallel processing. It should be easy for a compiler to see that Andof needs the results of

Rem, which needs the results of Index. In a more complicated example, you could have several different functions being evaluated at the same time. Because the code submitted by the Yale/Ansoft team simply contained the equations to be solved and said nothing about parallelism, the judges decided that the compiler, not the programmer, had found the parallelism.

The second major difference between functional and conventional languages is illustrated by the variable *k* in the C code. Its value changes on each pass through the loop. When analyzing for potential parallelism, this is called the multiple-assignment problem; the same storage location is used to hold different numbers at different times. Although efficient for storage, multiple assignment makes it hard to determine if one value depends on another. In the simple example case above, you know that you can't compute the third value assigned to *k* until you have computed the second.

Functional languages explicitly forbid multiple assignment. While this rule completely eliminates the necessity of doing dependence analysis, it comes at a high price: If you can never reuse a storage location, you must make a copy of a variable each time you change a value. While this copying is a nuisance for scalar variables, it is a very severe problem for arrays.

Look at the simple Fortran example of multiplying a matrix times a vector:

```
      DO 10 I = 1, N
        X(I) = 0.0
        DO 10 J = 1, M
10        X(I) = X(I) + A(I,J)*B(J)
```

If you need to do something like this in a functional language, you have two choices. You can make a copy of the array *X* each time you update one of its elements. In this case, you will end up with *MN+N* copies of the entire array, all but one of which you will never use. Or you can copy only the element being changed, but then you must maintain a data structure pointing to the last instance of each element of *X*. In either case, the performance of the code will be substantially reduced.

Crystal is really a preprocessor that generates a C program as its output. The original submission by the Yale/Ansoft team simply measured the speedup of the Crystal-generated C program as the number of processors varied. What if the functional code was a thousand times slower than a conventional version? The judges asked for a comparison with a sequential version that happened to exist in Fortran. The Fortran code ran 2.5 times faster than the Crystal code on one node. Of this 2.5-times difference in speed, 1.4 times was due to the fact that the Yale/Ansoft team's C compiler is not optimized as well as its Fortran compiler. But even if you reduce the reported speedups by 1.8 (the 2.5 Fortran factor divided by the 1.4 C factor), it is clear that the Crystal compiler has gone a long way toward proving the practicality of functional languages. ❖

When it comes to choosing compilers, MetaWare is the right choice!

Developing the next generation of software products is serious business. *You need the best tools to produce the best code. You need* **MetaWare**.

Superior Compilers

A compiler that handles large programs with ease, while producing the expected results, is the key to developing the highest quality applications. *Many of* **MetaWare's** *customers say that* **High C** ™ and **Professional Pascal** ™ are the highest quality compilers in the industry. They are reliable, and well documented, and their superior diagnostic messages help to produce better products more quickly. **There are no surprises with MetaWare compilers.**

Compiler Features

ANSI Standard with extensions • Generates small, fast executables • Support of 80x87, Weitek 1167/3167, 68881, and Am29027 math co-processors • Global common subexpression elimination • Live / dead code analysis • Constant propagation, copy propagation • Tail merging (cross jumping) • And many more!

Multiple Platform Support

MetaWare uses its own **Translator Writing System** to create all of its compiler products. Common components and library functions are shared across the product line. Improvements to compilers are quickly realized on all platforms.

MetaWare, High C and Professional Pascal are trademarks of MetaWare Incorporated; UNIX is a trademark of AT&T; Other products mentioned are trademarks of the respective companies indicated.

Supported Platforms (re: Dhrystones)

• Sun-3	— >50% > resident compiler.
• Sun 386*i*	— >50% > resident compiler.
• Sun-4	— >25% > resident compiler.
• PC: DOS, OS/2	— 3-10% > Microsoft C; 30%>MS Pascal, Lattice C.
• 386 32-bit DOS	— No real competition.
• 286, 386 UNIX	— 66% > than pcc on 386.
• VAX VMS	— ≈ DEC's excellent C and Pascal; Host for cross compilers and TWS, not Native.
• VAX Ultrix	— 19% > pcc on Dhrystone; much > Berkeley Pascal.
• RT PC	— 93% > 4.3bsd port of pcc.
• AIX/370	— Much better than any 370 C and VS Pascal.
• AMD 29K	— >40,000 Dhrystones.
• Intel i860	— >70,000 Dhrystones at 33 MHz.

So when it comes to selecting a compiler company, **you need the best!**

Can you really afford anything less?

MetaWare™
INCORPORATED

The Compiler Products for Professional Software Developers

2161 Delaware Avenue • Santa Cruz, CA 95060-5706
Phone: (408) 429-6382 • FAX: (408) 429-9273

Reader Service Number 12