from *SIAM News*, Volume 32, Number 9

Faster than a Speeding Algorithm

By Barry A. Cipra

The 1999 Wilkinson Prize for Numerical Software was awarded at ICIAM 99 to Matteo Frigo and Steven Johnson of MIT for their development of software dubbed the FFTW. That's short for "Fastest Fourier Transform in the West." Frigo, who recently completed a PhD in computer science, and Johnson, a graduate student in physics, began work on the project in 1997, as a sidelight to their respective theses. Some sidelight. The FFTW has become the poster child for a new paradigm in computer programming: self-tuning software that automatically adapts itself to the hardware it encounters.

The Wilkinson Prize, named in honor of the matrix maven James Hardy Wilkinson, is sponsored by Argonne National Laboratory in the U.S. and the U.K.-based National Physical Laborato-ry (NPL) and Numerical Algorithms Group (NAG). The prize has been awarded at each ICIAM since 1991.

The fast Fourier transform (FFT) itself, one of the workhorses of scientific computation, has been around for more than three decades. A lot of work has gone into writing efficient implementations. But programmers are increasingly hard pressed to make full use of the processing speeds of today's computers: Code that flies on one machine may crawl on another. Given the growing complexity of computer architectures and the rapid evolution of chip designs, finding an optimal implementation of the FFT for a new computer is tedious, painstaking—and often pointless—work.

You can "spend all your life trying to figure out how to best work on a particular machine," Johnson says, and "by the time you're done, that machine is obsolete."

That's a problem for scientific computing in general. "You can no longer program for performance," Frigo says. In the future, he thinks, software will have to optimize itself.

Jack Dongarra, a computer scientist at the University of Tennessee and Oak Ridge National Laboratory, agrees. "There's untapped potential there that's not being claimed by today's compiler technology," he says. In a plenary address at ICIAM, Dongarra described other efforts to solve the performance problem, including his own project for general matrix algorithms, ATLAS (Automatically Tuned Linear Algebra Subroutines), developed in collaboration with Clint Whaley and Antoine Petitet.

Since 1991, the J.H. Wilkinson Prize for Numerical Software has served to encourage young researchers to turn their ideas and algorithms into high-quality software that would be available to the community. Shown here at ICIAM 99 are the FFTW team, Matteo Frigo (left) and Steven Johnson (third from left), with Linda Petzold, the first to receive the prize (in 1991), and Alan Carle (who shared the prize with Christian Bischof in 1995).

The high-performance impasse stems from a mismatch between the blazing speeds of today's processors, now edging into the teraflops range on super-computers (and slouching toward gigaflops even on desktop machines), and the slower speeds at which data saunter around in memory. The hardware answer to the mismatch has been to create a hierarchy of memory, from the slow main memory, through several levels of increasingly fast, but smaller and smaller cache, up to the processor's registers, which run at top speed.

Ideally, data should move as little as possible. That is, if your calculation uses a number several times—as the Fourier transform does in spades—you should try to structure your algorithm to do all those calculations while the number is in the fast part of the memory. It's not possible to do that for all numbers simultaneously, however, so a certain (actually rather large) amount of back-and-forth shuttling becomes necessary. The trick is to find an algorithm that minimizes the delays that result from data traffic.

"The traditional way of doing this is called blocking," Johnson explains. "You structure your algorithm in blocks that correspond to the size of the cache that you want to work with. It reads as much as it can into that cache, works with it, and then reads the next block in. The problem with this is that the algorithm is structured explicitly for the size of the cache." When a new machine comes along, "everything is completely different, and you have to rewrite your code."

The FFTW takes a different tack. In effect, it ignores all questions of cache size and other technicalities of machine architecture. "We don't want to know about any of those things," Frigo says. Instead, the FFTW simply generates a multitude of different fast Fourier transform implementations, compares their performance, and picks the winner.

The FFTW can do this because the fast Fourier transform is not so much an algorithm as a strategy. The basic idea is one of divide and conquer—more technically called recursion: If $N = N_1N_2$, you can compute a transform of size N by computing N_1 transforms of size N_2 followed by N_2 transforms of size N_1. The same idea applies, recursively, to the smaller transforms. By the time you're done, the total cost of the com-putation is on the order of N log N. That's a far cry better than the N^2 effort implied by the straightforward definition of the (discrete) Fourier transform.

The classic FFT applies when N is a power of 2, say $N = 128$. The classic decomposition is shown in Figure 1a. But there are other possibilities as well, such as the one in Figure 1b. Depending on the machine, one may work better than the other. And of course the classic decomposition is tied to powers of 2. With $N = 30$, say, the best decomposition might be something like the one in Figure 1c.

"The FFT algorithm actually gives you a lot of freedom in how you decompose the transform," Johnson explains. What makes the FFTW innovative is its systematic exploration of all possible decompositions.

Well, not literally all. The tree of possibilities grows exponentially with the number of factors of *N*. To keep the computation within reason, "we have to make a simplifying assumption," Johnson says. Technically known as the assumption of optimal substructure, the simplification is that once, say, 2×3 has been determined to be the best decomposition for 6 (as opposed to 3×2 or 6 directly), then 2×3 will be the best decomposition for 6 wherever it appears. This may not actually be true. But the assumption is empirically valid—and more important, it trims the tree of possibilities to a linear twig.

When instructed to compile on a new machine, the FFTW begins by generating "codelets." These are highly optimized blocks of C code that compute transforms of various small sizes. The generator initially produces an abstract description of how the desired transforms are to be calculated. These descriptions tend to contain a lot of junk code, such as multiplications by 1 and additions of 0. The generator strips away such unnecessary code and applies other rules to streamline the final code.

At runtime, the FFTW's "planner" finds the optimal decomposition for transforms of a specified size. This needs to be done only once. The resulting plan is then available for repeated use—which is where the FFTW earns its keep.

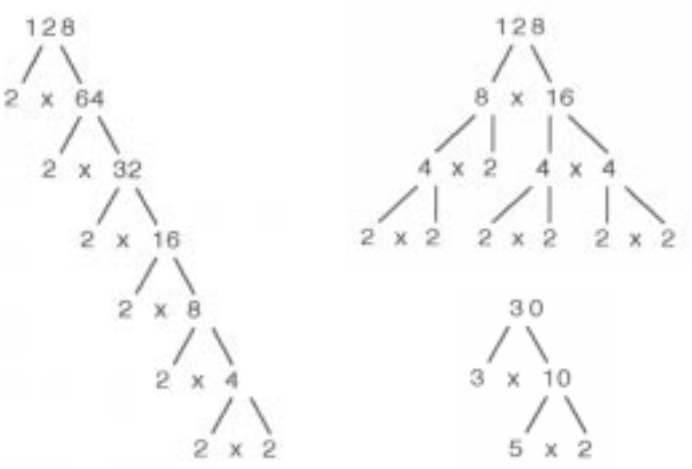

Figure 1. *The FFT applies when N is a power of 2, with the classic decomposition shown in Figure 1a (left) and an alternative in Figure 1b (top right). When N is no longer restricted to powers of 2, the best decomposition might look like that shown in Figure 1c (bottom right).*

The results are impressive. The FFTW not only clobbers the most common portable codes, it also rivals the FFT subroutines that vendors supply for their own machines. As Bruce Greer, a senior software architect at Intel, puts it, "Our developers will use FFTW as a benchmark against which to judge the quality of our software. If we can't beat FFTW, then we probably haven't tried hard enough."

Frigo and Johnson have made the FFTW freely available on the Web (www.fftw.org). "Several hundred people download the software every week, and we're getting like 3000 hits to our Web site every day," Johnson says. Users range from music enthusiasts who want to use the computer to tune their guitars to astrophysicists with gigabytes of pulsar data to analyze.

Dongarra and others, meanwhile, have been developing their self-tuning software for some of the basic linear algebra subroutines (BLAS). The effort began with PhiPAC, developed by James Demmel and colleagues at the University of California at Berkeley. Dongarra's ATLAS effort has taken the lead, in part because it works well on Pentium chips. (PhiPAC is targeted at RISC-based architectures, Demmel explains.) Each program aims to optimize the way large matrices are blocked for efficient multiplication.

"ATLAS probes the cache structure," Dongarra explains. "It tries to determine the size, the levels of cache, policies of cache, the way information flows from memory into cache and then into the registers. It tries to determine the number of functional units that are there and how they relate to the memory hierarchy of the machine. And then, given a number of things like that, it generates code, thousands of 'templates' of code, and literally runs them on the machine, and then selects the best set of options from those thousands of experiments that have been done.

"In some sense it's a brute-force approach. However, it's not as simple as that. There's a lot of techniques and strategies that have to be put in place to determine the correct set of parameters to search from, because the search space is so large, you just can't do everything."

One of the challenges for any package of BLAS is that the programs need to work on all kinds of matrices. A code that runs fast only on, say, 128×128 matrices is not going to win many races. "You have to get it right for a range of sizes," Demmel says.

Consequently, PhiPAC and ATLAS take hours to run—a hefty investment. But you need to run them only once, say during lunch. The code they generate pays for itself in the long run.

The gains from self-tuning "can be dramatic," Dongarra says. "I'm not talking about 10 or 20%," he adds. "I'm talking about 300% improvement."

Like the FFTW, the implementations ATLAS finds usually keep pace with vendor-supplied BLAS and leave reference BLAS trailing in the dust. (Comparison graphs can be seen at http://www.netlib.org/atlas/.)

"These are great tools," says Intel's Greer. "The concept behind them is very sound, and the results have shown their value."

Demmel points to one reason the FFTW, PhiPAC, and ATLAS do so well: The algorithms they're aiming at have a rich mathematical structure. In each case, he says, "the space of the problem is well defined." But the insights researchers gain from these highly structured problems may help in the design of self-tuning software for more complicated problems. One thing is certain: No matter how big and fast the computers get, scientists will always have problems that use every bit and cycle.

Barry A. Cipra is a mathematician and writer based in Northfield, Minnesota.