from *SIAM News*, Volume 31, Number 10

Parallel Computation of Hessian Matrices Under Microsoft Windows NT

By Brien Alkire

We describe an algorithm for the optimization of low-pass analog filters. The algorithm utilizes nonlinear programming techniques to select component values that minimize the error energy in the magnitude response of the filters, subject to constraints on the peaks of the magnitude response and on the component values.

Intermediate steps in the optimization process involve computations of Hessian matrices. Each Hessian matrix computation requires $O(k^5)$ complex-valued multiplications of double-precision floating-point numbers, where k is the number of components in the filter circuit. Fortunately, the elements of the Hessian matrices are independent and can be computed in parallel.

APPLICATIONS ON ADVANCED ARCHITECTURE COMPUTERS
Greg Astfalk, Editor

We describe a procedure for dividing the task and scheduling parallel threads on a Microsoft Windows NT-based multiprocessor system.

Computational Complexity

The optimization algorithm uses a finite set of discrete frequencies, known as a frequency grid, for evaluating the magnitude response of the filter. With a low-pass filter, we can distinguish between the pass band and the stop band. Let m denote the total number of frequencies on the grid, p of which are in the pass band and s in the stop band. In a typical application, there are $m = 1000$ frequencies on the grid, approximately half in the pass band and half in the stop band. In our problem the number of components k is 20.

In the pass band, we want to keep the magnitude response between an upper and a lower bound, and there are thus two constraints for each grid frequency. In the stop band, where we do not have a lower bound, we seek to keep the magnitude response below a certain peak; the ideal response in the stop band would be zero. There is thus one constraint for each grid frequency in the stop band.

To achieve the goal of practical component values, we set an upper and a lower bound on each component value. Therefore, there are two constraints for each of the k component values. Let N_c denote the total number of constraints. By adding the constraints for the pass band, stop band, and component values, we get the total number of constraints:

$$N_c = 2k + 2p + s \tag{1}$$

The constraints discussed so far are inequality constraints. An inequality constraint that has been transformed into an equality constraint is said to be an "active" constraint. Our algorithm performs active set iterations, taking n-tuples of the inequality constraints and transforming them into equality constraints. During a particular active set iteration, the algorithm may force some component values to their upper bound and others to their lower bound while ignoring the values of the remaining components. In theory, all possible combinations of constraints can be activated. The worst-case total number of active set iterations, denoted N_A, is given as:

$$N_A = \sum_{i=0}^{N_c} \frac{N_c!}{i!\left(N_c - i\right)!} = 2^{N_c} \tag{2}$$

In practice, useful filters are realized in fewer than ten active set iterations. In real-world applications we are concerned not with the exponential complexity of equation (2), but rather with the operations that take place during each active set iteration.

The most computationally complex operation that occurs multiple times during each active set iteration is the computation of Hessian matrices. The elements of the Hessian matrix are the second partial derivatives of a Lagrangian function with respect to the component values:

$$L(\mathbf{v},\lambda) = W_p \sum^{p} \left\| H\left(f_i, \mathbf{v}\right) \right| - G_p \Big|^2 + W_s \sum^{s} \left| H\left(f_i, \mathbf{v}\right) \right|^2 - \sum_{N_A} c_j(\mathbf{v}) \lambda_j \tag{3}$$

The magnitude of the frequency response at a grid frequency f_i with a vector of component values v is denoted $\left|H\left(f_i, v\right)\right|$. The ideal pass-band magnitude response is a constant, G_p. The ideal stop-band magnitude response, as mentioned earlier, is zero. The first two terms of equation (3) thus represent the weighted squared contributions in the pass band and the stop band, respectively, to the error of the filter from the ideal response. The real-valued constants W_p and W_s are weighting factors. The last term is the sum of the product of all equality constraints (i.e., equality constraints and active inequality constraints) with their sensitivity coefficients λ.

Let $v_1, v_2, \ldots, v_k$ denote the component values. The Hessian matrix, denoted **W**, is:

$$\mathbf{W} \equiv \nabla^2_{\mathbf{v}} L(\mathbf{v}, \lambda) =$$

$$\begin{bmatrix} \frac{\partial^2 L}{\partial v_1 \partial v_1} & \frac{\partial^2 L}{\partial v_1 \partial v_2} & \cdots & \frac{\partial^2 L}{\partial v_1 \partial v_K} \\ \frac{\partial^2 L}{\partial v_2 \partial v_1} & \frac{\partial^2 L}{\partial v_2 \partial v_2} & \cdots & \frac{\partial^2 L}{\partial v_2 \partial v_K} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 L}{\partial v_K \partial v_1} & \frac{\partial^2 L}{\partial v_K \partial v_2} & \cdots & \frac{\partial^2 L}{\partial v_K \partial v_K} \end{bmatrix} \tag{4}$$

Therefore, **W** is a $k \times k$ symmetric matrix. Each element is a second derivative of the Lagrangian function (3). Symmetric difference formulas are used for the numerical computation of the derivatives. For the second derivatives, the function must be evaluated twice. Each evaluation requires computing the frequency response and summing over all of the grid frequencies.

Computation of the frequency response involves the solution of an $N_n \times N_n$ matrix, with N_n being the number of nodes in the circuit. Each element is a complex-valued number. N_n is at most $2k$. The matrix, because it is positive-definite and can be solved by Cholesky decomposition, requires approximately $O((2k)^3/6)$ complex-valued multiplications (see [4]). For most applications, the numbers of grid frequencies in the pass band and the stop band are approximately equal. Therefore, the number of complex multiplications required for computing the Hessian matrix **W**, denoted N_w, is approximately:

$$N_w \approx 2(2p)\frac{(2k)^3}{6}k^2 = \frac{16}{3}pk^5 \tag{5}$$

The computational complexity in terms of complex-valued multiplication is therefore $O(k^5)$. For typical values, say $k = 20$ and $p = 1000$, the number of complex-valued multiplications $N_w \approx 1.71 \times 10^{10}$.

Division of Labor

Suppose that U_0 processors are available in a system. We can divide the Hessian matrix into nearly equal portions in terms of the number of elements of the matrix and divide the task among the processors.

The first row and the first column of the Hessian matrix contain $k + (k - 1) = 2k - 1$ elements. The second row and the second column, if we disregard the elements in common with the first row and column, then contain $(k - 1) + (k - 2) = 2k - 3$ elements. By induction, we can show that row i and column i contribute $2k - (2i - 1)$ elements that are not contained in the previous rows and columns. Theorem 1 is a statement that the elements of q rows and columns contribute $2kq - q^2$ elements not found in previous rows and columns.

Theorem 1:

$$\sum_{i=1}^{q}\left(2k-(2i-1)\right) = 2kq - q^2 \quad q \in \mathbf{Z}_+$$

Proof: $\sum_{i=1}^{1}(2k - (2i - 1)) = 2k - 2 + 1 = 2k \cdot 1 - 1^2$ so the theorem holds for $q = 1$. Now suppose that it is true for $q = m$, i.e., $\sum_{i=1}^{m}(2k - (2i - 1)) = 2km - m^2$. Consider $q = m + 1$. Then $\sum_{i=1}^{m+1}(2k - (2i - 1)) = \sum_{i=1}^{m}(2k - (2i - 1)) + (2k - (2(m + 1) - 1)) = 2km - m^2 + 2k - 2m + 1 = 2k(m + 1) - (m^2 + 2m + 1) = 2k(m + 1) - (m + 1)^2$. Therefore, it is true $\forall q \in \mathbf{Z}_+$ by induction. □

Using these relations, we can choose the number of rows and columns q that will provide k^2/U elements, with k^2 being the total number of elements in the matrix. That is, we can assign a processor to the task of computing k^2/U elements by determining which rows and columns of the Hessian matrix the processor should compute. Setting $2kq - q^2 = k^2/U$ and solving for q, we get the result of equation (1); the value is approximate, since we round q to the nearest integer value.

We begin with the $k \times k$ matrix and define $k_0 \equiv k$. We assign the first q_0 rows and columns to an available processor, where q_i is given as:

$$q_i \equiv k_i\left(1 - \sqrt{1 - \frac{1}{U_i}}\right) \tag{6}$$

The value q_i will have to be rounded to the closest integer. To assign rows and columns to the next processor, we can use recursive relations:

$$\begin{aligned} k_{i+1} &\equiv k_i - q_i \\ U_{i+1} &\equiv U_i - 1 \end{aligned} \tag{7}$$

Multiprocessing Considerations in Windows NT

We have implemented the parallel processing techniques in a program developed with the Microsoft Visual C++ development system (VC++). The Windows NT 4.0 operating system was targeted. Windows NT uses preemptive multithreading and multitasking and a symmetric multiprocessing model in which any thread can be assigned to any processor.

A function called **`GetSystemInfo`** can be used at runtime to query NT for the number of system processors. A separate thread can then be created for each processor. The primary thread is of course available. Separate worker threads are created when more than one processor is available. These threads behave like the primary thread but do not have a user interface.

The VC++ documentation reveals three choices for scheduling threads on a multiprocessor system: (1) We can let the operating system schedule the threads automatically. (2) Using the **`SetThreadAffinityMask`** function, we can explicitly specify which threads are allowed to run on which processors. (3) Using the **`SetThreadIdealProcessor`** function, we can suggest to the operating system which processor a thread should run on.

The documentation does not elaborate on how the **`SetThreadIdealProcessor`** function works, except to say that the function provides a useful hint to the scheduler. The documentation recommends that **`SetThreadIdealProcessor`** be used instead of manual assignment of the threads, which could interfere with the scheduler's ability to schedule threads effectively across processors. We evaluated all three methods.

Threads are assigned on a priority basis. There are 16 possible levels of thread priority. Each process is assigned one of four priority classifications: *Idle*, *Normal*, *High*, and *Real-time*. Each thread in a process can then be assigned a priority that modifies the priority classification of the process. By default, each thread has a priority level of *normal*, so that overall its priority is the same as the priority class of the owning process.

Our goal is for all the threads that compute the Hessian matrix to be of the same priority. In our implementation, only the priority class for the process can be selected; the priority level for all threads is *normal*.

Results

The processing time for an algorithm in a multitasking/multithreaded environment like Windows NT is necessarily non-deterministic. The operating system has to schedule processor time for numerous system and user threads, in addition to the algorithm under consideration. Since many factors contribute to the variations in processing time, we appeal to the central limit theorem and treat the distribution as Gaussian. Observations of the histograms of the processing times for the algorithm suggest that this assumption is reasonable.

We used a single system to gather results. The primary system consisted of two 150-MHz Standard Pentium processors, a 512-Kbyte cache, 40 Mbytes of memory, and the Windows NT 4.0 operating system with Service Pack 3. Our algorithm is not input/output-intensive, it is just math-intensive. No applications other than system processes and system threads were run during the tests. Each test consisted of evaluating the Hessian matrix as part of the optimization of a ninth-order elliptical low-pass filter. We repeated this computation to gather statistics for each different set of conditions.

We varied the following conditions: the process priority level, the number of threads used, and the method used for scheduling the threads on the system. To simplify the analysis, the thread and process priority levels were initially set to *normal*. Each of the methods for assigning threads to processors was then evaluated for a given number of threads. The method with the best processing times was then evaluated for different priority levels.

Given the large number of test cases, the number of trial runs for each set of conditions had to be kept low due to time constraints. In most cases, 30 trial runs were used. Therefore a Student-t model was used to perform hypothesis testing and compare the mean processing times for different sets of conditions. Because the Student-t model requires that the variances of the two distributions be identical, each Student-t test was preceded by a hypothesis test to determine whether the variances were indeed equal.

It was not always the case that the variances were equal. When they are not, an approximation based on the Student-t test is used. The approximation is:

$$\begin{aligned} t_{n_1+n_2-2} &= \frac{m_1 - m_2}{\sqrt{\frac{s^2}{n_1} + \frac{s^2}{n_2}}} \\ s^2 &= \frac{(n_1-1)s_1^2 + (n_2-1)s_2^2}{n_1+n_2-2} \end{aligned} \tag{8}$$

The quantities m_i, s_i, and n_i are the mean, standard deviation, and number of samples for distribution i. The quantity $n_1 + n_2 - 2$ is the modified number of degrees of freedom, and s^2 is the approximate pooled variance. This approximation is given in [3], page 196.

In all cases, the best performance occurred when the number of threads equalled the number of processors available (see Table 1). The table shows the average processing time as a function of the number of threads and the method of thread scheduling.

The differences in performance when two threads were used on a dual-processor system, with the scheduling technique varied, are not so obvious (see Table 2). In the table, *OS* refers to automatic scheduling of the threads by the operating system, *assigned* to explicit scheduling with the **`SetThreadAffinityMask`** function, and *suggested* to use of the **`SetThreadIdealProcessor`** function to suggest the optimal assignment to the operating system (the manufacturer's suggested technique).

Thread schedule	No. of threads 1	2	3	4
`AutoScheduling`	340	308	370	979
`SetAffinityMask`	350	228	387	630
`SetIdealProcessor`	350	316	388	627

Table 1. *Average processing time versus number of threads for a dual-processor system.*

Scheduling technique	Time (sec)
OS	308
Assigned	228
Suggested	316

Table 2. *Average processing time versus scheduling technique.*

Process priority	Time (sec)
Idle	245
Normal	228
High	231
Real-time	230

Table 3. *Average processing time versus process priority.*

Consider the *assigned* scheduling technique. With two threads on the dual-processor system, the computation time was reduced on average by 34.9%, with a statistical significance level of 1.22×10^{-22}. That is, there is a 1.22×10^{-20}% probability that the average processor times are actually equal, which implies that we can claim a 34.9% improvement with a high level of confidence. This result is obvious, as we would expect the best performance to occur when the number of threads equals the number of processors.

The best performance was realized with the *assigned* technique, rather than with the *suggested* technique recommended by the manufacturer of the operating system. Use of the *assigned* technique resulted in an average reduction of 26% in computation time, as compared with both of the other techniques, with a statistical significance of 2.57×10^{-32}.

Table 3 shows the results for the use of two threads on the dual-CPU system and the *assigned* thread-scheduling technique, with the process priority varied. No other user applications were running when these tests were performed; the results could differ under other conditions. Here we see that performance for *idle* priority was, on average, 7.32% worse than that for *normal* priority, with a statistical significance of 0.024. The relative performances of the *normal*, *high*, and *real-time* priority cases did not differ by much. The lowest level of statistical significance was 0.167.

Conclusions

Use of two threads, *normal* priority, and explicit scheduling resulted in an average reduction in processing time of 34.9% compared with the single-threaded version on the dual-CPU system. The difference is significant at the 1.41×10^{-31} level. A 95% confidence interval for the reduction in processing time is 30.6%–35.3%.

Explicit scheduling of the threads resulted in the best performance. On average, the computation time was reduced by 26% compared with the other methods, with a 2.57×10^{-32} level of significance.

The process priority had little impact on performance under the condition that no other user applications were run during the tests. The *idle* priority performance was degraded by 7.32% as compared with the *normal* priority, with a 0.024 level of significance. The relative performances of *normal*, *high*, and *real-time* priorities differed with a minimum level of significance of 0.167. That is, there is no significant difference under the test conditions.

Data synchronization is an important consideration in multithreading environments. If multiple threads attempt to access a shared resource simultaneously, the results can be unpredictable.

The data-synchronization objects available in Visual C++, while beyond the scope of this article, are worth mentioning. Among them are mutexes, semaphores, critical sections, and events. Some of these objects work across process boundaries as well as across thread boundaries.

Another technique, the one we used in this application, is to make separate copies of the resources needed by each thread. Synchronization in this application would have necessitated pausing the threads to prevent collisions and resulted, consequently, in a drop in performance. Care has to be taken when passing pointers of objects created in one thread to another. This should be avoided if possible.

The availability of multiprocessor systems in a low-cost PC form-factor opens the door to improvements in performance of applications that ran previously on single-processor systems. Application designers should keep multiprocessing issues in mind when developing software in order to take advantage of the potential benefits.

References

[1] B.F. Alkire, *Peak constrained least squares optimization of analog filters using nonlinear programming,* MS Thesis, California State University Northridge, 1997.

[2] M.G. Kendall and A. Stuart, *The Advanced Theory of Statistics,* Hafner Publishing Company, New York, 1961.

[3] S. Dowdy and S. Wearden, *Statistics for Research,* John Wiley & Sons, New York, 1983.

[4] R.L. Burden and J.D. Faires, *Numerical Analysis,* PWS Publishing Company, Boston, 1993.

Brien Alkire (brien@alkires.com) is a PhD student in electrical engineering (operations research) at the University of California, Los Angeles, and a senior engineer at Litton Guidance and Control Systems in Woodland Hills, California.