Parallel MATLAB®: Multiple Processors and Multiple Cores

BY CLEVE MOLER

Twelve years ago, in the Spring of 1995, I wrote a Cleve's Corner titled "Why There Isn't a Parallel MATLAB®." That one-page article has become one of my most frequently referenced papers. At the time, I argued that the distributed memory model of most contemporary parallel computers was incompatible with the MATLAB memory model, that MATLAB spent only a small portion of its execution time on tasks that could be automatically parallelized, and that there were not enough potential customers to justify a significant development effort.

The situation is very different today. First, MATLAB has evolved from a simple "Matrix Laboratory" into a mature technical computing environment that supports large-scale projects involving much more than numerical linear algebra. Second, today's microprocessors have two or four computational cores (we can expect even more in the future), and modern computers have sophisticated, hierarchical memory structures. Third, most MATLAB users now have access to clusters and networks of machines, and will soon have personal parallel computers.

As a result of all these changes, we now have Parallel MATLAB.

MATLAB supports three kinds of parallelism: multithreaded, distributed computing, and explicit parallelism *(see sidebar)*. These different kinds can coexist—for example, a distributed computing job might invoke multithreaded functions on each machine and then use a distributed array to collect the final results. For multithreaded parallelism, the number of threads can be set in the MATLAB Preferences panel. We use the Intel Math Kernel Library, which includes multithreaded versions of the BLAS (Basic Linear Algebra Subroutines). For vector arguments, the MATLAB elementary function library, which includes exponential and trigonometric functions, is multithreaded.

Choosing which form of parallelism to use can be complicated. This Cleve's Corner describes some experiments that combine multithreaded and distributed parallelism.

Parallel Computing Clusters

Figure 1 is a schematic of a typical parallel computing cluster. The gray boxes are sepa-

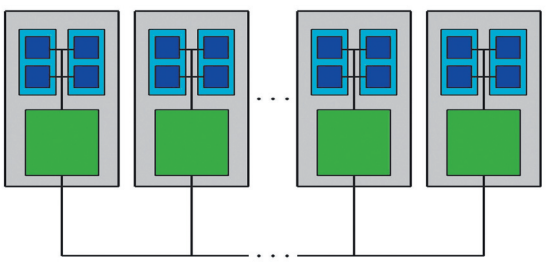

Figure 1. A typical parallel computing cluster.

Three Types of Parallel Computing

Multithreaded parallelism

In multithreaded parallelism, one instance of MATLAB automatically generates multiple simultaneous instruction streams. Multiple processors or cores, sharing the memory of a single computer, execute these streams. An example is summing the elements of a matrix.

Distributed computing

In distributed computing, multiple instances of MATLAB run multiple independent computations on separate computers, each with its own memory. Years ago I dubbed this very common and important kind of parallelism "embarrassingly parallel" because no new computer science is required. In most cases, a single program is run many times with different parameters or different random number seeds.

Explicit parallelism

In explicit parallelism, several instances of MATLAB run on several processors or computers, often with separate memories, and simultaneously execute a single MATLAB command or M-function. New programming constructs, including parallel loops and distributed arrays, describe the parallelism.

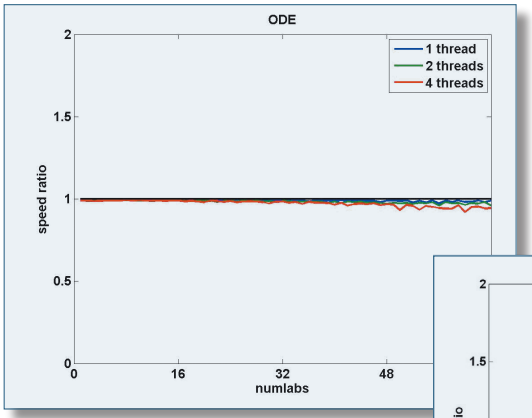

Figure 2. Execution speed ratio for the ODE task.

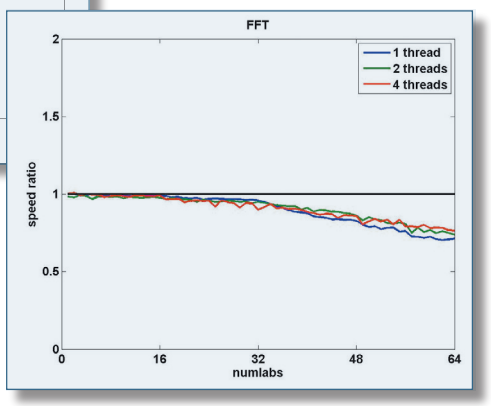

Figure 3. Execution speed ratio for the FFT task.

rate computers, each with its own chassis, power supply, disc drive, network connections, and memory. The light blue boxes are microprocessors. The dark blue boxes within each microprocessor are computational cores. The green boxes are the primary memories. There are several different memory models. In some designs, each core has uniform access to the entire memory. In others, memory access times are not uniform, and our green memory box could be divided into two or four pieces connected to each processor or core.

At The MathWorks, we have several such clusters, running both Linux and Windows operating systems. One has 16 dual-processor, dual-core computers. Each machine has two AMD Opteron 285 processors, and each processor has two cores. Each computer also has four gigabytes of memory, which is shared by the four cores on that machine. We therefore have up to 64 separate computational streams but only 16 primary memories. We call these clusters the HPC lab, for High Performance or High Productivity Computing, even though they are certainly not supercomputers.

Our HPC lab has one important advantage over a top-of-the-line supercomputer: one person can take over the entire machine for a session of interactive use. But this is a luxury. When several people share a parallel computing facility, they usually must submit jobs to a queue, to be processed as time and space on the machine permit. Large-scale interactive computing is rare.

The first version of Distributed Computing Toolbox, released in 2005, provided the capability of managing multiple, independent MATLAB jobs in such an environment. The second version, released in 2006, added a MATLAB binding of MPI, the industry standard for communication between jobs. Since MATLAB stands for "Matrix Laboratory," we decided to call each instance of MATLAB a "lab" and introduced `numlabs`, the number of labs involved in a job.

Beginning with version 3.0 of Distributed Computing Toolbox, The MathWorks added support for new programming constructs that take MATLAB beyond the embarrassingly parallel, multiple jobs style of computing involved in this first benchmark.

Using Multithreaded Math Libraries Within Multiple MATLAB Tasks

The starting point for this experiment is `bench.m`, the source code for the `bench` demo. I removed the graphics tasks and all the report generating code, leaving just four computational tasks:

ODE—Use ODE45 to solve the van der Pol ordinary differential equation over a long time interval

FFT—Use FFTW to compute the Fourier transform of a vector of length 2^{20}

LU—Use LAPACK to factor a 1000-by-1000 real dense matrix

Sparse—Use A\b to solve a sparse linear system of order 66,603 with 331,823 nonzeros

I used Distributed Computing Toolbox and MATLAB Distributed Computing Engine to run multiple copies of this stripped-down `bench`. The computation is embarrassingly parallel—once started, there is no communication between the tasks until the execution time results are collected at the end of the run. The computation is also multithreaded.

The plots in Figures 2 through 5 show the ratio of the execution time for one singly threaded task on one lab to the execution time of many multithreaded tasks on many labs. The horizontal line represents perfect efficiency—p tasks can be done on p labs in the same time that it takes to do one task on one lab. The first and most important point is that for all four tasks, the blue curve is indistinguishable from the horizontal line, up to 16 labs. This indicates that single-threaded tasks get perfect efficiency if there is at most one task per computer. This is hardly surprising. If it weren't true for these embarrassingly parallel tasks, it would be a sign of a serious bug in our hardware, software, or measurements. With more than 16 labs, or more than one thread per lab, however, the performance is more complicated. To see why, we need to look at each computation separately.

The ODE task (Figure 2) is typical of many MATLAB tasks: It involves interpreted M-code, repeated function calls, a modest amount of data, a modest amount of arithmetic for each step of the ode solver, and many steps. With this task we get perfect efficiency, even up to 64 labs. Each individual core can handle all the computation for one lab. The memory demands do not dominate. Use of more than one thread has no effect because the task does not access any multithreaded library. In fact, it is not clear how multithreading could be used.

The FFT task (Figure 3) involves a vector of length $n = 2^{20}$. The $n \log n$ complexity

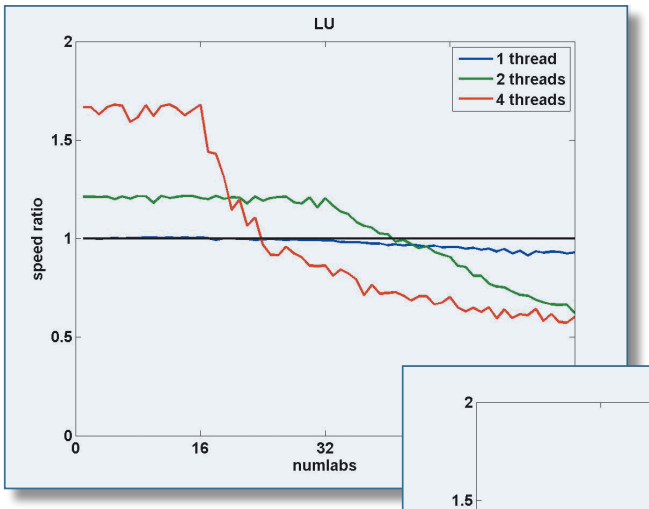

Figure 4. Execution speed ratio for the LU task and different levels of multithreading.

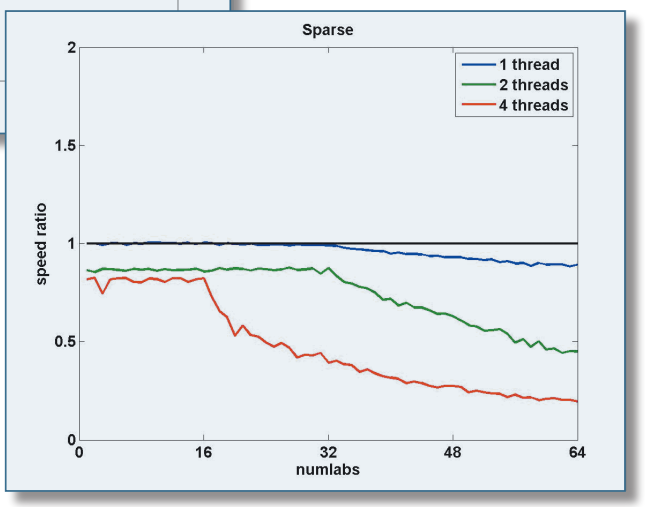

Figure 5. Execution speed ratio for the sparse task and different levels of multithreading.

implies that there is only a handful of arithmetic operations for each vector element. The task gets nearly perfect efficiency for 16 labs, but with more than 16 the ratio deteriorates because the multiple cores cannot get data from memory fast enough. The time it takes 64 labs to complete 64 tasks is about 40% more than the time it takes one lab to do one task. Again, multiple threads have no effect because we are not using a multithreaded FFT library.

The blue line in the LU plot (Figure 4) shows that with just one thread per lab we get good but not quite perfect efficiency, up to 64 labs. The time it takes 64 labs to complete 64 tasks is only about 6% more than the time it takes one lab to do one task. The matrix order is $n = 1000$. The n^2 storage is about the same as the FFT task, but the n^3 complexity implies that each element is reused many times. The underlying LAPACK factorization algorithm makes effective use of cache, so the fact that there are only 16 primary memories does not adversely affect the computation time.

The green and red lines in the LU plot show that using two or four threads per lab is an advantage as long as the number of threads times the number of labs does not exceed the number of cores. With this restriction, two threads per lab run about 20% faster than one thread, and four threads per lab run about 60% faster than one thread. These percentages would be larger with larger matrices and smaller with smaller matrices. With more than 64 threads—that is, more than 32 labs using two threads per lab, or more than 16 labs using four threads per lab—the multithreading becomes counterproductive.

Results for the sparse task (Figure 5) are the least typical but perhaps the most surprising. The blue line again shows that with just one thread per lab, we obtain good efficiency. However, the red and green lines show that multithreading is always counterproductive—at least for this particular matrix. CHOLMOD, a supernodal sparse Cholesky linear equation solver developed by Tim Davis, was introduced into MATLAB recently, but before we were concerned about multithreading. The algorithm switches from sparse data structures, which do not use the BLAS, to dense data structures and the multithreaded BLAS when the number of floating-point operations per nonzero matrix element exceeds a specified threshold. The supernodal algorithm involves operations on highly rectangular matrices. It appears that the BLAS do not handle such matrices well. We need to reexamine both CHOLMOD and the BLAS to make them work more effectively together.

For More Information

- CHOLMOD Linear Equation Solver
  www.cise.ufl.edu/research/sparse/cholmod
- Cleve's Corner Collection
  www.mathworks.com/res/cleve

Resources

VISIT
www.mathworks.com

TECHNICAL SUPPORT
www.mathworks.com/support

ONLINE USER COMMUNITY
www.mathworks.com/matlabcentral

DEMOS
www.mathworks.com/products/demos

TRAINING SERVICES
www.mathworks.com/training

THIRD-PARTY PRODUCTS
www.mathworks.com/connections

WORLDWIDE CONTACTS
www.mathworks.com/contact

E-MAIL
info@mathworks.com