



Enhancing Multicore System Performance Using Parallel Computing with MATLAB

By Piotr Luszczek

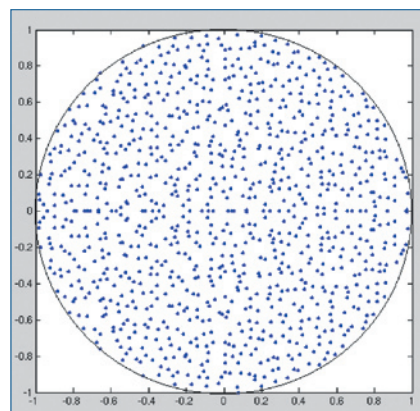
A visit to the neighborhood PC retail store provides ample proof that we are in the multicore era. The key differentiator among manufacturers today is the number of cores that they pack onto a single chip. The clock frequency of commodity processors has reached its limit, however, and is likely to stay below 4 GHz for years to come. As a result, adding cores is not synonymous with increasing computational power. To take full advantage of the performance enhancements offered by the new multicore hardware, a corresponding shift must take place in the software infrastructure—a shift to parallel computing.

MATLAB® and Parallel Computing Toolbox™ address the challenge of getting code to work well in a multicore system by enabling you to select the programming paradigm most appropriate to your application. Using a typical numerical computing problem as an example, this article describes how to use the two most basic of these paradigms: threads and parallel for-loops.

A Typical Numerical Computing Problem

To illustrate the two paradigms, we will use MATLAB to test a hypothesis regarding Girko's circular law. Girko's law states that the eigenvalues of a random N-by-N matrix whose elements are drawn from a normal distribution tend to lie inside a circle of radius `sqrt(N)` for large enough N. Our hypothesis is that Girko's circular law can be

modified to apply to singular values. The hypothesis is justified because singular values are eigenvalues of a modified matrix. Applying our modified Girko's law could save computation time by enabling us to estimate the singular values for any normally random matrix without performing singular value decomposition (SVD).



Products Used

- MATLAB®
- Parallel Computing Toolbox™

This theorem can be represented with the following MATLAB code:

```
N = 1000;
plot(eig(randn(N)) / sqrt(N), 'o');
```

The code produces the visualization shown in Figure 1.

Each dot represents an eigenvalue on a complex plane. Notice that most of the eigenvalues reside inside a circle of radius 1 and are centered at the origin of the axis—a compelling indication that, in accordance with Girko's circular law, the magnitude of an eigenvalue does not exceed the square root of the matrix size.

Figure 1. Eigenvalues of a random matrix of size 1000 scaled by $1/\sqrt{1000}$. The elements of the matrix are drawn from the normal distribution.

To apply Girko's law to SVD, we generate random matrices in MATLAB and then examine their singular values to see if we can formulate a hypothesis based on numerical experiments. In particular, we want to compute the value of `max(svd(randn(N)))` for arbitrary values of variable `N` and then look for a pattern in the results. We can use our theoretical knowledge of SVD to explain the pattern.

The following loop generates normally random matrices and finds their SVD:

```
y = zeros(1000,1);
for n = 1:1000
    y(n) = max(svd(randn(n)));
end
plot(y);
```

Running this loop on a typical desktop computer with only one core enabled would take more than 15 minutes. To reduce computing time, we will run the loop on a four-core machine using threads and parallel for-loops, and then compare the performance results.

Using Threads

Threads are a common software solution for parallel programming on multicore systems, but it is important to bear in mind that multithreading and multicore processors are not synonymous. The best performance is often obtained when the number of threads and the number of cores correspond, but there are circumstances when there should be fewer threads than cores. We will experiment to determine the optimal number of threads for our computation.

We run the code, adjusting the number of threads using either the Preferences window on the MATLAB Desktop or the `maxNumCompThreads()` MATLAB function.

Figure 2 shows results on different thread counts. In addition to time, it shows parallel speed-up and parallel efficiency. The former is a ratio of execution time on `N` cores to execution time on one core—ideally, we'd like to achieve a speedup of `N` on `N` cores. The latter is a ratio of speed-up to the number of cores—ideally, it should be 100%.

Our computational experiment produced mixed results. Using threads did make the computation faster, but in our example, only the call to `svd()` was actually parallelized. This is because threading support in MATLAB is implicit: the user does not determine which parts of the code should run in parallel. Some statements are more amenable to implicit parallelization than others—in our case, only the call to `svd()`. None of the other statements benefit from multithreading because they do not have enough computational load.

On one hand, we can speed up the calculations by using more cores and without changing the original code. On the other, we quickly reach a point of diminishing returns where adding cores does not appreciably reduce execution time.

To explain this, we must again examine the most computationally intensive part of our for-loop: the call to the `svd()` function. The matrices passed to `svd()` range from 1-by-1 to 1000-by-1000: 500-by-500, on average. Such small matrix sizes do not yield sufficient performance gains on multicore machines. Clearly, a different parallelization approach is required.

Using Parallel for-Loops

A `parfor` (parallel for) loop is useful in situations that require many loop iterations of a simple calculation, such as a Monte Carlo simulation. To run `parfor` we use Parallel Computing Toolbox. We begin by adapting our original code (Figure 3).

Number of threads	Time to run the for loop	Speed-up	Efficiency
1	902.6	1.00	100%
2	867.2	1.04	52%
3	842.3	1.07	35%
4	862.3	1.05	26%

Figure 2. Code performance on different thread counts.

<pre>y = zeros(1000,1); for n = 1:1000 y(n) = max(svd(randn(n))); end plot(y);</pre>	<pre>y = zeros(1000,1); parfor n = 1:1000 y(n) = max(svd(randn(n))); end plot(y);</pre>
--	---

Figure 3. Left: Original code. Right: Code adapted to run with `parfor`.

Just as `maxNumCompThreads()` controls the parallelism of the multithreading approach, the `matlabpool` command controls the parallel behavior of the `parfor` syntax. `matlabpool` sets up a task-parallel execution environment in which parallel for-loops can be executed interactively from the MATLAB command prompt.

The iterations of `parfor` loops are executed on labs (MATLAB sessions that communicate with each other). Like threads, labs are executed on processor cores, but the number of labs does not have to match the number of cores. Unlike threads, labs do not share memory with each other. As a result, they can run on separate computers connected via a network. For our example, however, we only need to know that Parallel Computing Toolbox makes `parfor` work efficiently on a single multicore system. Each core, or *local worker*, can host one lab.

A question naturally arises: is changing the code worthwhile? The most accurate answer is, “It depends.” In our case, changing the code is worthwhile because the results clearly indicate the benefits of using the `parfor` syntax (Figure 4).

Adding more cores would further reduce computation time, since we have not reached the point of diminishing returns with four cores. The technical term for this behavior is *scalability*: for our SVD computation, `parfor` scales better than multithreading. It also provides the kind of performance that might be expected from four cores. We see substantial speedup and acceptable efficiency, which was not the case when we used multithreading.

Without delving too deeply into the implementation details, it is necessary to explain the success that resulted from using `parfor`.

The most notable feature of our sample code is that each iteration of the loop is independent. This feature alone makes the application of `parfor` so easy yet so effective. The only tasks left for the runtime system inside `parfor` are distributing the loop iterations to the cores and gathering results for use outside the `parfor` loop.

A word of caution about the effect of `parfor` on random number generation. Matrices generated inside a `parfor` loop with functions such as `randn()` will not be identical to their for loop counterparts because of the way the `parfor` loop iterations are scheduled. In most cases, this discrepancy is perfectly acceptable.

While using `parfor` has advantages, it also has limitations. For example, if there is dependence between the loop iterations and the dependence can be detected through code analysis, then executing the `parfor` loop will cause an error. If the dependence cannot be detected, then the only indication of the problem will be incorrect results. The following code illustrates this problem:

```
total = 0;
A = zeros(1000, 1);
parfor i = 1:100
    total = total + i; % OK: this is ...
    ...a known reduction operation

    A(i+1) = A(i) + 1; % error: ...
    ...loop iterations are dependent
end
```

The expression that accumulates `total` depends on the iteration variable `i`, but this is not a problem. The `parfor` runtime can easily work with such expressions by evaluating partial sum on each available lab and then combining the results.

The second expression that operates on array `A` does, however, pose a problem. The iteration with `i=2` cannot compute the value of `A(3)` until the value `A(2)` is computed in iteration 1. By the same token, the iteration with `i=3` depends on the iteration with `i=2`, and so on.

Let’s try to fix this problem by taking a closer look at what happens at each iteration:

```
Iteration 1: i = 1
A(2) = A(1) + 1 = 0 + 1 = 1
Iteration 2: i = 2
A(3) = A(2) + 1 = 1 + 1 = 2
Iteration 3: i = 3
A(4) = A(3) + 1 = 2 + 1 = 3
```

It soon becomes clear that we can achieve the same effect as the loop in Figure 3 by writing the following loop:

```
parfor i = 1:10
    A(i+1) = i;
end
```

Now the `parfor` loop executes and yields the intended result.

Number of labs	Time to run the for-loop	Speed-up	Efficiency
1	870.1	1.00	100%
2	487.0	1.79	89%
3	346.2	2.51	83%
4	273.9	3.17	79%

Figure 4. Code performance on different lab counts.

Extending Parallel Computing

Multicore processors are here to stay, and so is parallel programming. MATLAB already supports several parallelization methods. Support for additional methods will be provided in future versions of the product.

Consumers and engineers alike believe that we will see more cores inside future computers. The trend so far has been to double the number of cores every few years. This translates into a doubling of computational power. Harnessing that power will require the right software, and writing that software will require the right software tools. MATLAB is well positioned to fulfill that requirement. ■

For More Information

Demos

- Using `parfor` to Run Loops in Parallel
www.mathworks.com/pcomp_demos
- Parallel Programming in MATLAB
www.mathworks.com/pcomp_demos

Articles

- Eigenvalues and Condition Numbers of Random Matrices. Alan Edelman. Ph.D. thesis, Massachusetts Institute of Technology, May 1989.
- Language Design for an Uncertain Hardware Future. Roy Lurie. *HPCwire*, September 28, 2007
www.hpcwire.com/features/17902899.html
- Parallel MATLAB: Multiple Processors and Multiple Cores. Cleve Moler. *The MathWorks News & Notes*, June 2007
www.mathworks.com/clc_multiproc

Resources

VISIT

www.mathworks.com

TECHNICAL SUPPORT

www.mathworks.com/support

ONLINE USER COMMUNITY

www.mathworks.com/matlabcentral

DEMOS

www.mathworks.com/demos

TRAINING SERVICES

www.mathworks.com/training

THIRD-PARTY PRODUCTS AND SERVICES

www.mathworks.com/connections

Worldwide CONTACTS

www.mathworks.com/contact

E-MAIL

info@mathworks.com

© 2008 The MathWorks, Inc. MATLAB and Simulink are registered trademarks of The MathWorks, Inc. See www.mathworks.com/trademarks for a list of additional trademarks. Other product or brand names may be trademarks or registered trademarks of their respective holders.

80367v00 09/08