Measuring MATLAB Performance

Bill McKeeman

August 2008

Improving the performance of MATLAB is a MathWorks priority. We encourage our customers to tell us their observations. When an issue has business consequences, we elevate the issue on our list of tasks. As becomes apparent to those dealing with MATLAB performance, it is a complicated subject. The sections of this paper deal with measuring performance, one topic at a time.

Isolating an Issue

Before attempting to make an M application faster, it is important to find out what parts of the application are taking the time. It is usually the slowest part that determines the overall performance. Use the MATLAB **profiler**. When a time-consuming part of the program is identified, write a smaller test that includes as little else as feasible, then do further analysis on the smaller test instead of the whole application.

Running Clean

The minimal preparation for an experiment is to get everything else off the computer: **email**, **browsers**, **editors**, and so on.

Measuring Time

Of the several ways to measure elapsed time in MATLAB, a pair of calls to tic/toc before and after the computation of interest provides wall-clock elapsed time to approximately 1 microsecond accuracy.

```
>> tstart = tic();
>> interesting computation
>> telapsed = toc(tstart);
```

Passing the output result of **tic** to the input of **toc** prevents these calls from interfering with other calls to tic/toc, since the state is contained in the output value **tstart** rather than in a global variable. Measurements taken on the command line (as above) may include the time it takes you to move your fingers and also do not use the MATLAB JIT, so it is a better idea to write a function harness containing the tic/toc calls and run the harness instead.

Comparing across MATLAB Releases

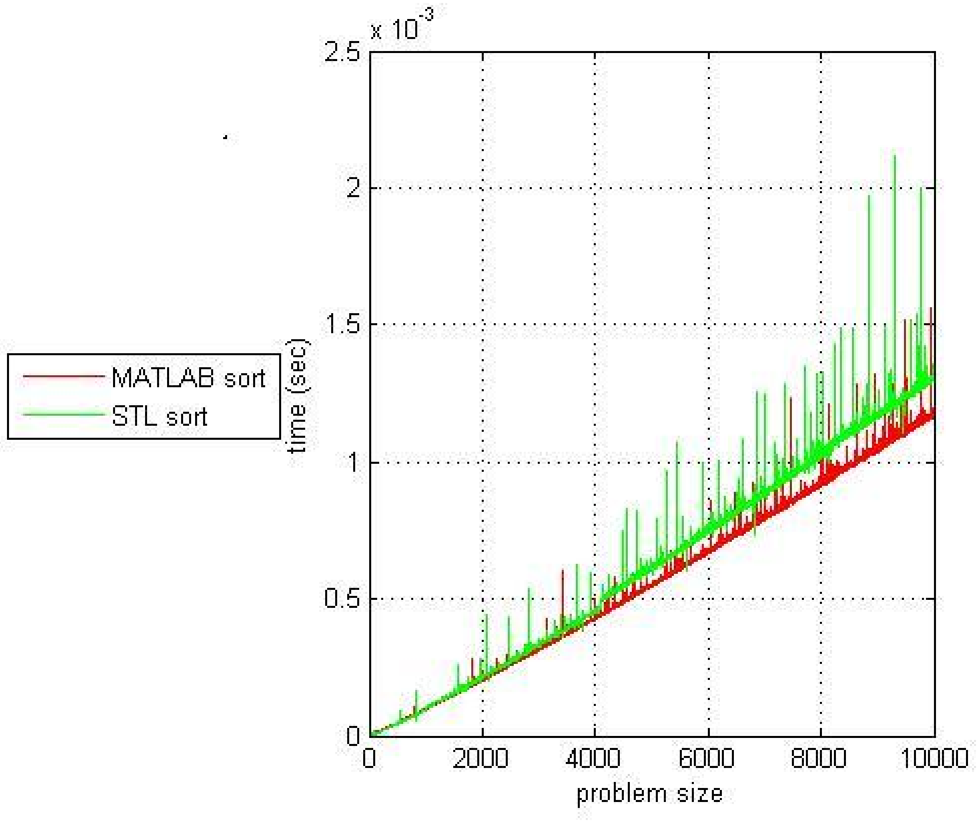
Experiments comparing **different releases of MATLAB** contain a hidden trap. The tic/toc pair has been improved over the last few releases. Older releases may show apparently better results just because the older tic/toc pair was not as accurate.

Noise Supression

The hallmark of science is repeatable results. It benefits nobody to have a performance query end in **Cannot Repeat Customer Data**. Nothing can be fixed if it cannot be measured. All performance measures are going to be **infested with noise**. The most serious consequence is that bottlenecks in the program under test simply cannot be seen through the obscuring noise. Leading sources of noise are other programs, external devices, PIT interrupts, hardware memory cache, paging, variable speed CPUs, multicore, shared memory, real parallelism and MATLAB compilation.

Solutions to getting usable data in the face of noise depends on the nature of the experiment. In the case of a small, CPU-bound program a relatively simple test can be run. As the complexity of the tested program and the amount of memory needed increases, more sophisticated tests are needed. Each of several sections below focus on a specific measurement problem and its resolution.

To illustrate this idea, here is some carefully gathered data of run time versus problem size that was expected to be entirely regular but showed noise spikes.



The graph above contained enough information. It is reasonable to assume, in this case, that noise is positive (increasing the time measured). The lower bound of the two curves above was sufficiently convincing to validate that the MATLAB sort was performing up to industry standards.

Measuring Small Programs

Measuring the time taken by **very small computations** often tells more about the overhead of the experiment than about the computation itself.

As a rule of thumb, trust values of tic/toc when they are about 0.001 seconds. In some cases, the interesting computation must be put in a loop to get useful data. Here is a typical very small experiment, where REPS is chosen to make each trip through the loop take a reliable amount of time. The *interesting computation* itself is duplicated to amortize the overhead of the loop.

```
function tmeasured = timeit
  REPS = 1000;                % to be modified for optimal value
  tstart = tic();             % start the timer
  for i=1:REPS
    interesting computation
    interesting computation (cut and paste)
    interesting computation
    interesting computation
    interesting computation
  end
  telapsed = toc(tstart);     % record the test time
  assert(telapsed > 0.001, 'inaccurate result: increase REPS');
  assert(telapsed < 0.01, 'inaccurate result: decrease REPS');
  tmeasured = telapsed/REPS/5;
end
```

Trends are more Informative than Single Data Points

Looking at the data for sort (above), it is clear that a single data point could be misleading. When there is an independent variable (data size in the case of sort), a plot of run-time versus the variable is better at showing the underlying truth. Suppose you have two CPU-bound functions F1(len) and F2(len) that do the same thing, and that you want to compare their execution speeds. You might like fancier output, but this will do for our purposes here. Since MATLAB has to get F1 and F2 compiled, and then get the compiler out of the way, the test harness starts with a few initial throw-away runs.

```
function timeit
  for i=1:3         % get tests compiled, let paging settle down
    F1(1); F2(1);
  end
  MAXLEN = 100000;                    % sample big enough range
  REPS = 5;                           % enough tries to avoid noise
  sample = 1:1000:MAXLEN;             % stride the data
  t1 = zeros(REPS,numel(sample));
  t2 = zeros(REPS,numel(sample));
  for i = 1:numel(sample)
    for r=1:REPS
      tstart = tic();
      F1(sample(i));                  % first function
      telapsed = toc(tstart);
      t1(r,i) = telapsed;             % record time

      tstart = tic();
      F2(sample(i));                  % second function
      telapsed = toc(tstart);
      t2(r,i) = telapsed;             % record time
    end
  end
  for i=1:numel(sample)
    d1(i) = min(t1(:,i));             % min supresses noise
    d2(i) = min(t2(:,i));
  end
  plot(sample, t1, 'g');
  hold on
  plot(sample, t2, 'r')
end
```

If the purpose of the experiment is to improve the code, the recommended experimental setup consists of taking the **minimum** time of multiple runs (as is done in `timeit` above). The idea is that there is some base time required to do the user's computation, and it will always be present. Noise beyond the minimum is not "fixable", so is best ignored by the developer. The more runs made, the more repeatable the minimum value will be. Paying the extra test time cost to get a repeatable number saves the occasional wild-swan chase.
http://en.wikipedia.org/wiki/Black_swan_theory

If the purpose is to measure user experience, the same setup can be used but **median** time is a more reasonable metric. The user will experience the noise, so it should be included. The median will discard outliers. The median includes some noise and therefore may have a different value when rerun, say at a different time of the day. The reader might be interested in the min, max, and distribution as well as the median.

The **average** usually does not give more repeatable data; instead it hides variability of the data in a sum. A bad outlier will skew the results of the average more than it does min and median.

One-time CPU-bound Tasks

If the computation is CPU limited but is going to be run only infrequently, elapsed wall-clock time is still the appropriate metric, but the experiment should be run only once. There is one more consideration: the time it takes just after machine boot and MATLAB startup is likely to be longer than measured for a **warm machine** and **warm MATLAB**. The experiment should measure the situation that is important to the customer.

```
function telapsed = timeit
  tstart = tic();
  one-time computation
  telapsed = toc(tstart);
end
```

Memory Allocation

MATLAB allocates and frees memory when computing with large operands. The memory activity can be a large contributor to costs. To find out what memory activity is taking place, there is a pair of features that work much like tic/toc. The following example allocates a million doubles (and a few more things). If you run it again, TotalFreed will be large because you overwrite x.

```
>> feature mtic;
>> x=ones(1000);
>> feature mtoc

ans =
      TotalAllocated: 8145528
          TotalFreed: 144296
    LargestAllocated: 8000000
           NumAllocs: 780
            NumFrees: 779
                Peak: 8002604
```

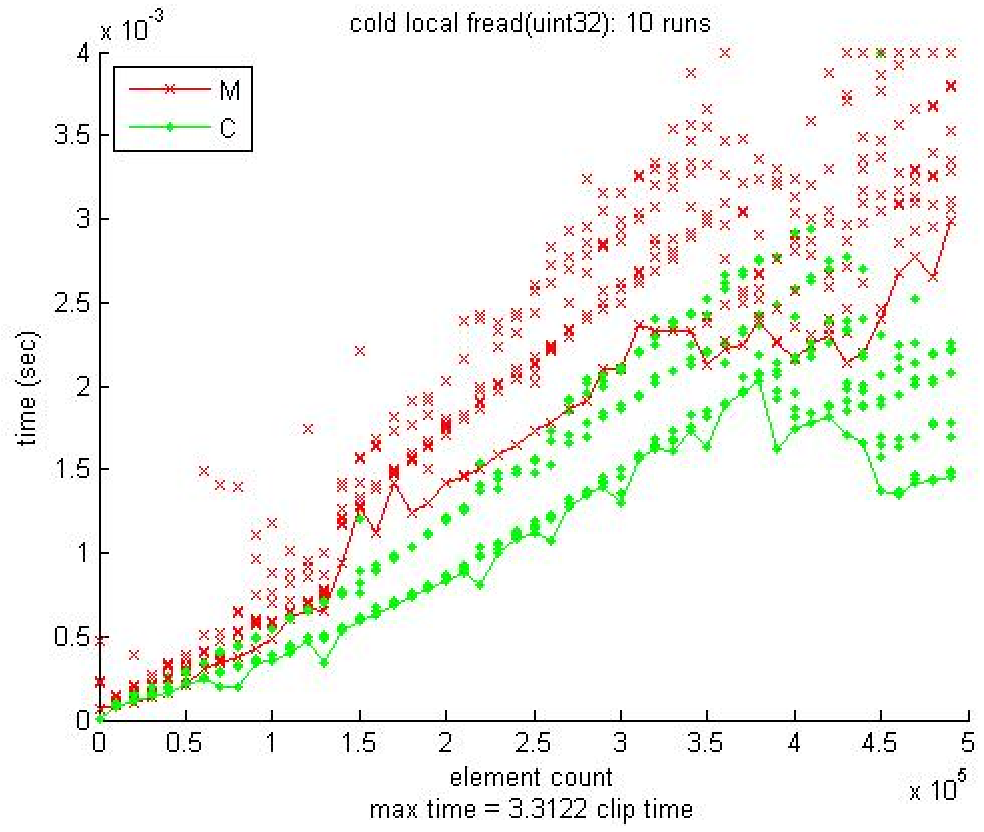
Pushing Memory to the Limit

It is sometimes the case that the experiment pushes computer memory to the limit. At some point the computer will run out of memory and will put a stop to things. Short of that, expect the cost of paging between disk and main memory to show up in the measurements. See what can be insight can be gained by calling the MATLAB memory function. Try

```
 >> help memory
```

File and Network Traffic

If the program necessarily uses external devices, the data is going to include the underlying management software that is moving the data around. In this case the idea of taking the minimum of several runs is wrong because the data-movement costs are not noise, but are rather the heart of the problem; the variation, minimum, and maximum are more informative. The idea is to make several runs (as in the case of noise suppression), but instead plot all of the data. The following graph is an example of **MATLAB fread** versus **C fread** for increasingly long blocks of data. The data is clipped at the top of the graph (some points are off-scale by a factor of 3 or more). Note how much less would be learned by just plotting the minimum (which is also shown on the graph).



Drilling Down

When a test shows unexplained noise, one might want to look deeper. On Windows, the task manager shows several continuous plots of resource consumption. In particular, on multiple CPUs, it plots the usage of each over time. If the task manager does not supply sufficient insight, then Windows program **perfmon** might. You can customize perfmon to sample a large number of counters. For instance, processor time, user time, context switches/second, page faults, and so on. Each of the following problems, alone or in combination, can obscure test results.

A Catalog of Noise

1. Other irrelevant programs, such as email and browsers. Experiments need to be run "clean". **Kill everything you can**.
2. External Devices. The use of networks and mass storage can dominate CPU costs. Unless the point of the experiment is to measure the effect of external devices, their access should be **avoided** in test programs. File servers often have server-side caches and client-side caches. Local disks also cache data in memory. The time to access a cache is usually an order of magnitude faster than the device it serves.
3. The PIT or Program Interval Timer. This hardware timer interrupts 100 times a second. The operating system does (sometimes very time consuming) chores before returning to the application under test. On a single CPU computer, all of this time shows up as noise in performance data. The trick in avoiding the PIT is to run several tests that take less than **0.01 seconds**. Some data will not include the interrupt. The minimum of the results of the results should be noise-free.
4. Hardware Memory Cache. The architecture of modern CPUs is based on the assumption that, for short intervals of time, part of memory is much more heavily accessed than the rest. A memory cache miss will show up as noise. Repetitions should eventually settle down the cache to repeatable behavior.
5. Paging. In addition to managing the memory cache, the operating system attempts to keep a much larger working set in main memory. A page miss will show up as a large spike of noise because at least one disk access is needed. This effect is pronounced when the program under test is pushing the amount of available memory or MATLAB is cold (paged out). Paging behavior can be detected with Windows program **perfmon**.
6. Variable CPU speed. Modern hardware is able to change its **clock speed** many times a second, in response to temperature sensors, battery level, keystrokes, or even the current computational load. The wall clock tic/toc speed will not change with the clock speed so a test on a hot laptop will take longer. The antidote is to run with wall current, on a clean hard surface, without touching the keyboard.
7. Multicore. The operating system (these days) has more than one available CPU. Even if the program under test is entirely sequential (and therefore will not use additional CPUs) the operating system may. With one core, everything gets charged to the program under test. With more cores, some of the operating system housekeeping may be done in parallel, effectively reducing the time measured during the test. The number of cores used by a single program (e.g. MATLAB) can be set by bringing up the **task manager**, **right-mouse** the process name MATLAB.exe, select **Set Affinity**, marking the CPUs you want to run. You can, in this way, force uniprocessor operation.
8. Shared Memory (SMP). Multiple cores may contend for residence in main memory. It is a peculiarity of performance that adding CPUs may slow down an application because each CPU can clobber the working set of the other CPUs. If you suspect a **CPU fight**, use Set Affinity to turn one or more CPU off and see if it helps.
9. Parallelism. The program under test may actually utilize multiple CPUs. Much of the MATLAB library exploits mathematical properties to split tasks into parallel subtasks. While improving performance, a consequence is that measurements become dependent on the number of available CPUs and also can introduce unexpected noise from the first earlier sources of noise listed above.
10. MATLAB compilation. M code is compiled into executable form by MATLAB upon first access. I have seen my PC essentially hang for several seconds waiting for a cold MATLAB to move its considerable bulk into memory for execution. Compile time will show up in the first data point. The next data point may show MATLAB paging out its compiler code to make room for execution. It can take several throw-away runs to get past these transient effects. I usually throw away three.

Conditions of the Experiment

The program **configinfo.m** lists the characteristics of the customer's machine: hardware, operating system, MATLAB version and settings, and the like. Including the output of configinfo.m with a report simplifies the task of repeating the experiment in-house. Here is some typical output.

```
>> configinfo
Matlab configuration information
This data was gathered on: 24-Jan-2008 13:23:42
MATLAB version: 7.6.0.80 (R2008a)
MATLAB root: C:\Program Files\MATLAB\R2008a
MATLAB accelerator enabled
MATLAB JIT: enabled
MATLAB assertions: disabled
MATLAB Desktop: enabled
Java JVM: enabled
Java version: Java 1.6.0 with Sun Microsystems Inc.
CPU: x86 Family 6 Model 14 Stepping 8, GenuineIntel
The measured CPU speed is: 1838 MHz
Number of processors: 2
RAM: 1022 MB
Swap space: 2456 MB
Microsoft Windows XP
Number of cores: 2
Number of threads: 2
```

Reports to The MathWorks

Performance reports from customers fall into two broad categories

- *A customer needs to make an M program run faster*. The solution could be to change the M program or to implement an improvement in MATLAB. For the former, our Application Engineers give advice about ways the M can be changed to get better results. The MLINT program contains a catalog of compiled wisdom of this kind. For the latter, we take the supplied information and factor it into our development priorities for the next releases. In either case the customer can request and expect a reply from The MathWorks.

- *A customer has discovered something interesting*. There are too many such discoveries for MathWorks to guarantee individual replies to each one of them. Often there is no simple answer and equally often the answer is about a current situation that is going to be changed by the next release. In some cases, there is no way to answer a question without revealing The MathWorks' proprietary information. It is helpful to MathWorks if the customer indicates that a report is for our information only and a personal reply is not necessary. The accumulated knowledge often shows up in the public blogs maintained by our senior developers.

Final Advice

The internals of MATLAB, and therefore the performance characteristics, are under intense development. The current implementation is always open to change in the next release. For this reason, we do not advise customers to go to undue lengths to get their programs to run optimally in the current release. Rather, we advise customers to write elegant M code and depend on The MathWorks to make it run fast.

Useful Pointers

MathWorks Technical Support
http://www.mathworks.com/support/contact_us/index.html

Steve Eddins' blog
http://blogs.mathworks.com/steve/

Steve Eddin's program timeit
http://www.mathworks.com/matlabcentral/fileexchange/loadFile.do?objectId=18798

Loren Shure's blog
http://blogs.mathworks.com/loren/

MathWorks White Paper: **Improving Performance and Memory Usage**
http://www.mathworks.com/access/helpdesk/help/techdoc/matlab_prog/f8-705906.html

MathWorks White Paper: **Programming Tips**
http://www.mathworks.com/access/helpdesk/help/techdoc/matlab_prog/f10-57434.html