The Art of Benchmarking: Evaluating the Performance of R on Linux and OS X

Jasjeet S. Sekhon*

Published in *The Political Methodologist*, 14(1), 2006

*I thank Nate Begeman of Apple for software optimizations and Michael Herron and Jeff Lewis for helpful suggestions. `sekhon@berkeley.edu`, `http://sekhon.berkeley.edu/`, Associate Professor, Travers Department of Political Science, Survey Research Center, 2538 Channing Way, UC Berkeley, Berkeley, CA, 94720.

With the growing use of computational statistical methods which tax even today's powerful computer chips, it is of interest how various applications perform on modern operating systems. Of course, there is more to picking an operating system than speed (e.g., ease of administration, viruses, and most importantly applications). But speed is key especially when purchasing servers and clusters which many of us are doing. How various statistical packages perform is also an important consideration. For example, is Matlab generally faster than R ("yes") and are both faster than Stata ("yes")? But since much of the statistics and political methodology community has coordinated on R, I focus on it and examine how efficiently it runs on various operating systems.

The short summary is that for some key operations Linux is faster than Windows XP and both are faster than OS X unless the default OS X memory allocator is replaced. In fairness to OS X, the Windows XP version of R already uses a modified memory allocator instead of the system default. Although some Windows benchmarks are presented here, the focus is on Linux and OS X benchmarks because few if any methods people (I don't know of any!) would run computational servers with Windows.

Modern computers and operating systems are so complex and the tasks they are ask to perform are so variable that there is no single measure or test of overall performance. Such a measure of performance is even more elusive than g—the general intelligence factor. Therefore, it is of utmost importance when conducting benchmarks that they be as closely related as possible to the computations one will perform with production code. If R is the application whose performance one cares about most, relying on benchmarks from seemingly closely related tasks such as video encoding can often lead to erroneous inferences even though both tasks are floating-point intensive.[1] Even within R, one should benchmark the code one is actually going to be using. For example, if one is going to be running Matching estimators

[1]Floating-point numbers are the way that a subset of real numbers are represented on a modern computer. Such numbers consist of an integer exponent and a signifcand which consists of the significant digits of the number. On a computer, floating-point operations are handled differently than integer operations and different chips may be better at one than the other. For example, the latest generation AMD Opteron chips generally have better floating-point performance than Intel x86 chips, but the latter have better integer performance.

using a given algorithm, then it is best to run benchmarks as similar to this usage scenario as possible. The relative performance of various operating systems or computer chips when inverting matrices may be different than the relative performance when sorting the contents of a matrix even though both tasks are done in R and use the same data. The different tasks may, for example, have different memory requirements.

The difficulty of understanding and separating out what is happening on a computer is a cautionary tale regarding the difficulty of making inferences in general. Modern computers, which are Turing machines, are deterministic systems. There is nothing stochastic about them, and there are no unobservable factors at play. For example, even the "random" number generators most statistical software rely on are generated by deterministic algorithms such as the Tausworthe-Lewis-Payne generator (Bratley, Fox, and Scrage 1983, 190) which is one of the better ones currently available—it is, for example, used by `rgenoud`.[2] Notwithstanding these deterministic properties, it is difficult to determine why a certain algorithm performs better on one operating system than another. To make our inferential task easier, I try to limit the number of factors in play by using exactly the same hardware for each operating system. Thanks to Apple's switch to Intel chips, this matching exercise is now straightforward to do. Back when Apple was still using PowerPC chips, comparisons between OS X and Windows were more difficult to conduct, and for any given benchmark Apple or Microsoft could blame the chip instead of the operating system or claim that a given application was tuned to one particular chip or another.[3]

All benchmarks are conducted on what is at the time of this writing, Apple's fastest Intel hardware: a MacBookPro with Intel Duo (2.16GHz) and 2GB of RAM and 120GB hard disk. This machine has two CPU cores, but all of the presented benchmarks only use one of the cores.

[2] `rgenoud` is an **R** package for Genetic Optimization Using Derivatives and it is used below. See `http://sekhon.berkeley.edu/rgenoud`.

[3] On a more mundane level, the same deterministic random number generator will produce different "random" numbers on different chips because of architectural differences. There are obvious ways around this issue, but this issue and many others like it complicate benchmarking across chips.

Given that one should benchmark the algorithm one will actually be using, benchmarks are based on my Matching package for R (Sekhon 2007).[4] The package provides functions for multivariate and propensity score matching and for finding optimal balance based on a genetic search algorithm. A variety of univariate and multivariate tests to determine if balance has been obtained are also provided. The most computationally intensive part of the package is the `GenMatch` function which finds optimal balance using multivariate matching where a genetic search algorithm determines the weight each covariate is given (Sekhon 2007; Diamond and Sekhon 2005). Because the genetic algorithm calls the matching function many, many times, a great deal of time has been spent optimizing the matching procedure itself. Indeed, after I had posted earlier benchmarks on my website which looked particularly poor for OS X, programmers at Apple, including members of Apple's OS X Performance Group, helped optimize my code. After these and other optimizations, my matching algorithm—implemented in the `Match` function—is the fastest I know of in any language. The computationally intensive portion of the code is written in C++, and key matrix operations are handled by the BLAS libraries.[5] So any benchmark using this algorithm becomes a benchmark of floating-point performance, the system BLAS implementation, compiler performance, and operating system memory management.[6] In order to eliminate the possibility of these factors confounding our results, we match on all of the non-operating system factors. The same BLAS implementation is used for both Linux and OS X as well as the same compiler (gcc) and optimization flags. And since we are using the same computer, the CPU's floating-point unit, cache and other hardware components are identical in all simulations. We are then left with differing operating systems.

[4]See `http://sekhon.berkeley.edu/matching`

[5]The BLAS (Basic Linear Algebra Subprograms) are routines for performing basic vector and matrix operations. They provide a consistent programming interface across hardware specific implementations.

[6]The same BLAS implementation was used for all of the benchmarks, ATLAS (non-threaded). The OS X vecLib Framework is based on the ATLAS BLAS. Goto's BLAS are currently the fastest, but at the time these benchmarks were run they were not available for x86 OS X. I have since helped Goto produce a beta patch for x86 OS X.

Linux versus Mac OS X on Intel Dual Core

In early May I posted on my website benchmarks comparing Linux and OS X (I later added Windows XP benchmarks).[7] In one of the original benchmarks, both Linux and Windows XP were more than twice as fast as OS X. And in a second (more representative) benchmark, Linux was about 20% faster than OS X. The benchmarks were posted on Digg (`http://digg.com`) and a variety of other high traffic Internet websites such as OSnews (`http://OSnews.com`). This attention generated a lot of comments and suggestions.

With the help of a variety of developers working at Apple and elsewhere, the large OS X performance gap previously reported was significantly reduced. The most important improvement is the use of a more efficient algorithm which relies on optimized BLAS to perform key matrix operations. This change increased the performance of the code on all platforms. The performance gap was further closed by compiling and linking R on OS X against Doug Lea's malloc (called dmalloc for short). Malloc is a function which allocates memory for an object (such as a variable or matrix) requested by a program, in this case R.

However, a Linux speed advantage remains which varies with dataset size. For example, the gap ranges from 0% for a small dataset to 10% for what is a medium size dataset for the algorithm in question. The gap shrinks again to 0% for a larger dataset. The performance gap is much greater if the default OS X malloc is used notwithstanding the new algorithm: the gap goes from essentially zero for a small dataset, to 40% for a medium one, and up to 50% for a large one. Therefore, I recommend that the OS X version of R be compiled so that memory is handled by dmalloc instead of the default OS X malloc. R for OS X should be linked against dmalloc just as it is for Windows.

The default malloc on OS X, like the default malloc on Windows XP, causes a large performance degradation relative to the default malloc on Linux. R developers use the default system malloc on every operating system but Windows. It turns out that this decision is a bad one in the case of OS X because OS X makes more frequent system calls

[7] `http://sekhon.berkeley.edu/macosx`

when allocating memory. A system call when allocating memory is done so that the kernel can allocate and clean up memory. This helps with memory defragmentation and insures that free memory is recovered by the operating system so it can be used by other processes. However, all of this comes at a cost. Not only do these system calls take time, they also cause page faults. These occur when a program requests data that is not currently in real memory. The operating system then fetches the data from virtual memory and loads it into RAM. Therefore, it is much faster for the application to not call the kernel and simply manage memory itself. The downside of this is that the operating system will generally run out of memory more quickly. The threshold at which different memory allocators turn memory allocation over to the kernel varies greatly. A key performance issue arises because OS X makes system calls for allocations of 15 kilobytes (KB) and larger while, for example, the current version of dmalloc makes system calls for allocations of 256KB and larger.[8]

To make matters worse, unlike on Linux, this threshold is not changeable by the user. On Linux, it is common practice in high performance computing to completely avoid calls to the kernel when allocating memory for the computationally intensive process.[9] On Linux, which uses the GNU malloc, calls to the kernel (via the mmap function) can be avoided completely by setting two runtime environmental variables: MALLOC_TRIM_THRESHOLD to -1 and MALLOC_MMAP_MAX to 0. It is unfortunate that it is not possible to do something similar with OS X's default malloc because it would help alleviate the performance issue. Therefore, one has to use an alternative memory allocator on OS X.

Picking what the threshold should be for a memory allocator to allow the kernel to allocate memory is an art. There is no generally optimal solution. An issue which arose with OS X was that professional video and graphics users were running out of allocatable memory, thus the current OS X memory allocator minimizes the amount of memory used—i.e., it needs to aggressively recover freed memory and optimally allocate memory pages. This was an issue

[8]One KB is equal to $2^{10} = 1024$ bytes. Dmalloc's threshold used to be 128KB but it was increased to 256KB as computers have changed—compare version 2.6.6 with 2.8.3.

[9]For example, see the following documentation from Lawrence Livermore National Laboratory `http://www.llnl.gov/LCdocs/linux/index.jsp?show=s7`.

because OS X only assigns 2GB of memory space to processes regardless of the amount of physical memory in a computer. Linux does not have this limitation. Therefore the current OS X memory allocator was written to minimize memory usage even if that means that much smaller memory allocations are handled by the OS X kernel than by the Linux kernel.

The Benchmarks

As noted above, the benchmarks are based on my Matching package for R (Sekhon 2007). The benchmark scripts only vary by the sample size of the dataset being examined. The data sizes are: 445, 890, 1780 and 5340 observations.[10] Each script runs the benchmark three times and the best runtime of the three is recorded. Each script is executed 1000 times and the average times are reported below. The setup is outlined in Table 1.

Figure 1 presents the results for the first benchmark. For the first benchmark, there is only a small difference between Linux and OS X with the default malloc and no difference when dmalloc is used. This is the benchmark which was used to optimize the software on OS X by Apple's OS X Performance group. The difference between Linux's and OS X's default malloc is small but statistically significant—the p-value based on the empirical distribution over 1000 simulations is 0.09.

The results for this benchmark with the new code are in sharp contrast to the original results (which were obtained using exactly this script), but which used an algorithm which was not as optimized—e.g., it did not make use of the BLAS libraries. In the original benchmark, Linux took 8.84 seconds, Windows XP 9.38 seconds and OS X (with the default malloc) 22.78 seconds! Clearly the code improvements have sped up the code on all operating systems (on Linux from 8.84 seconds to 4.4), but the improvement for OS X has been tremendous: from 22.78 seconds to 4.6!

[10] Each script runs the benchmark three times and the best runtime of the three is recorded. Each script is executed 1000 times and the average times are reported below. The scripts are available on my website. 445: `http://sekhon.berkeley.edu/macosx/GenMatch.R`, 890: `http://sekhon.berkeley.edu/macosx/GenMatch2.R`, 1780: `http://sekhon.berkeley.edu/macosx/GenMatch3.R`, and 5340: `http://sekhon.berkeley.edu/macosx/GenMatch5.R`.

However, even with the new algorithm, as we increase the sample sizes, differences begin to become more pronounced. With 890 observations, Linux is now 20% faster than default OS X (p-value=0.00) and 10% faster than OS X with dmalloc (p-value=0.00).

For 1780 observations, there is some evidence that the difference between Linux and OS X with dmalloc is either asymptoting at about 10% or possibly even shrinking—from 1.11 times as slow as Linux in the previous benchmark to 1.08 times as slow now. The next simulation will help to nail this down. In any case, the gap between Linux and the default OS X malloc version has doubled and OS X is now about 1.4 times slower than Linux.

In an attempt to answer the asymptoting vs shrinking gap question, a benchmark was run with 5340 observations (12 times the original dataset). The Linux advantage over OS X using dmalloc was only present for a given range of dataset size and with 5340 observations it has disappeared once again. But the gap between default OS X and Linux continues to grow.

Why exactly there remains a gap for some dataset sizes between the OS X dmalloc and the Linux results is not clear. One way to try to answer the question is to use Shark which allows one to see what functions an application is spending its time in.[11] With the default OS X malloc, Shark is able to quickly show that the process, in this case the 5240 observations benchmarks, is spending a lot of time calling a system library.[12] However, with dmalloc, it is not clear why OS X is slower than Linux for some sample sizes. No system call jumps out, so a mystery remains which further analysis would no doubt clear up.[13]

[11] For information about Shark see `http://developer.apple.com/tools/sharkoptimize.html`.

[12] About 12% of the runtime is spent in 'mach_msg_trap' which is a symbol in the libSystem.B.dylib library. The only other libSystem calls taking more than 1% are '_isnand' (1.4%) and 'dyld_stub_isnand' (1.3%). 'malloc' itself is reported to take up (directly) 0.0% of the runtime—so calls to it are being accounted elsewhere.

[13] With dmalloc the only libSystem calls which take up greater than 1% of the time are '_inand' (2.5%), 'dyld_stub__isnand' (2.5%) and 'syscall' (1.2%). '_mmap' takes up 0.3% of the runtime and 'malloc' 0.4%. Does the large amount of time spent in 'mach_msg_trap' indicate that the XNU kernel is spending a lot of time message passing as it is often accused of or is this simply how Shark is reporting the kernel's default virtual memory manager?

Conclusion

This brief report gives a flavor of how to conduct software benchmarks. It is striking how even though everything a computer does is deterministic and observable, the experimental method is still essential for making inferences. Methods like those proposed by qualitative researchers for making inferences with deterministic systems are not used in the literature. When benchmarking, it is common to match on (and hence eliminate) as many confounders as possible and to report measures of uncertainty. Since computers are deterministic, the remaining uncertainty must come from confounders. For example, when conducting these benchmarks, I tried to stop as many daemons as possible.[14] But background tasks, such as those related to the graphics system whether it be X11 or Quartz, will still take up CPU resources at times the analyst does not predict.

References

Bratley, P., B.L. Fox, and L.E. Scrage. 1983. *A Guide to Simulation*. New York: Springer-Verlag.

Diamond, Alexis and Jasjeet S. Sekhon. 2005. "Genetic Matching for Estimating Causal Effects: A General Multivariate Matching Method for Achieving Balance in Observational Studies." See `http://sekhon.berkeley.edu/papers/GenMatch.pdf`.

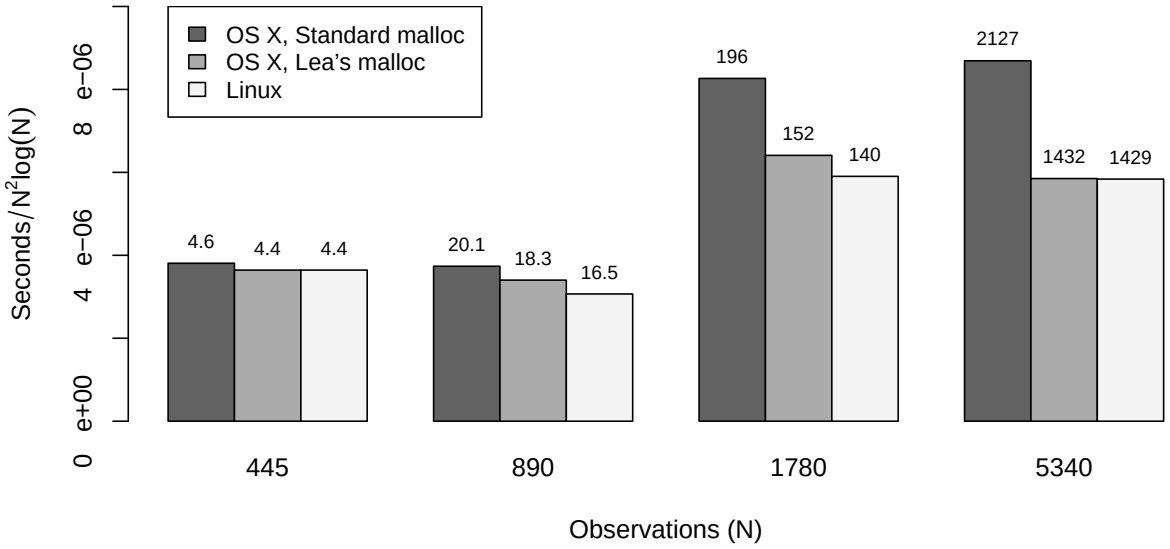
Sekhon, Jasjeet S. 2007. "Matching: Algorithms and Software for Multivariate and Propensity Score Matching with Balance Optimization via Genetic Search." *Journal of Statistical Software*. In press.

[14] A daemon is a background process which handles services such as automatic disk mounting and printing.

Table 1: Operating System and Computer

Label	OS and Chip
OS X Standard malloc	Tiger on MacBookpro, Intel 2.16GHz Dual Core 2GB RAM
OS X Lea's malloc	Same as above but with dmalloc
Linux	Ubuntu Linux (Drapper Drake) with i686-SMP kernel on MacBookpro, Intel 2.16GHz Dual Core 2GB RAM Note: Xorg server running with GNOME

Figure 1: GenMatch Benchmark Comparison



Entries above bars are gross processing seconds. Scaling factor for the $y-$axis is the average asymptotic order of the algorithm which is $O(N^2log(N))$. Asymptotically, the dominating factor is quicksort which is applied N times, and the average order of quicksort is $O(Nlog(N))$.