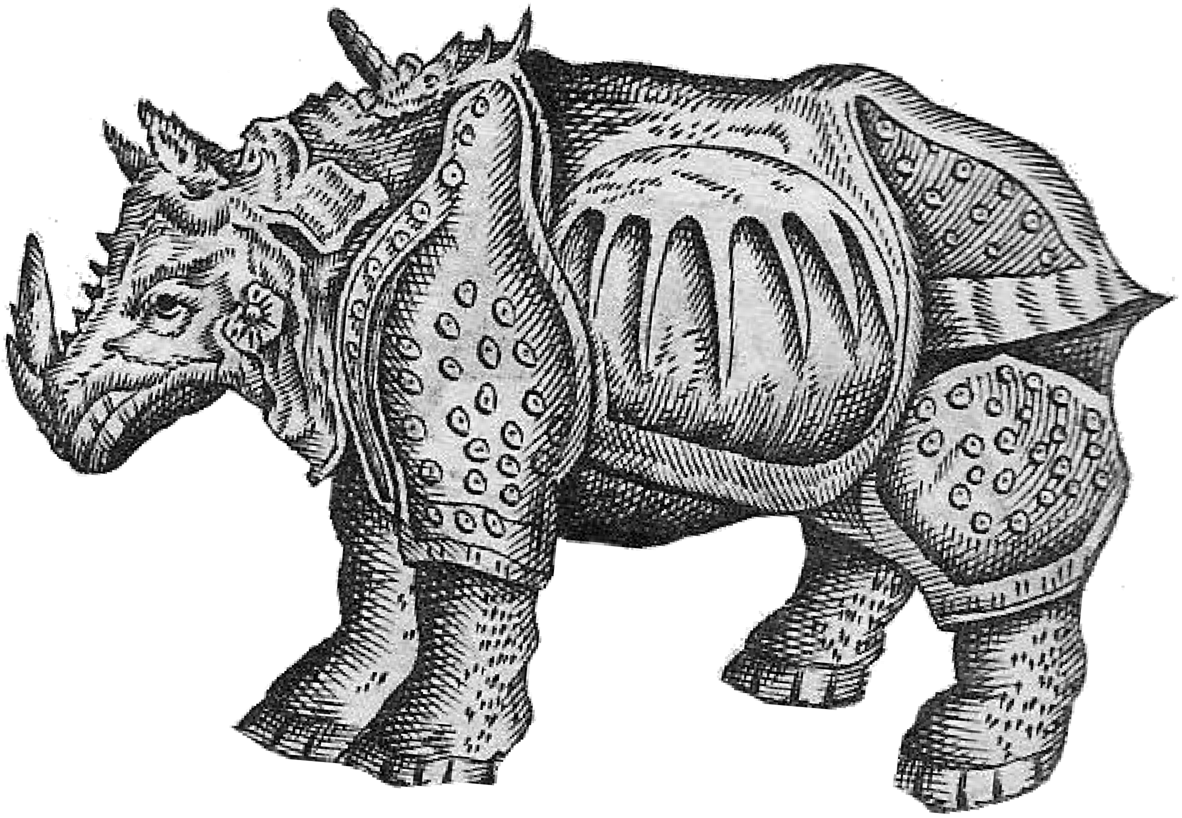
Version
0.1-0



Programming with Big Data in R

Guide to the pbdPAPI Package

Performance Analysis Tools for R

pbdR Core Team

Guide to the
pbdPAPI Package

Performance Analysis Tools for R

June 10, 2014

Drew Schmidt
National Institute for Computational Sciences
University of Tennessee

Christian Heckendorf
Boston University
Boston, Massachusetts

Wei-Chen Chen
Department of Ecology and Evolutionary Biology
University of Tennessee

George Ostrouchov
Computer Science and Mathematics Division,
Oak Ridge National Laboratory

Pragneshkumar Patel
National Institute for Computational Sciences
University of Tennessee

Version 0.1-0

Acknowledgement

Schmidt was supported in part by the National Institute for Mathematical and Biological Synthesis, sponsored by the National Science Foundation, the U.S. Department of Homeland Security, and the U.S. Department of Agriculture through NSF Awards #EF-0832858 and #DBI-1300426, with additional support from The University of Tennessee, Knoxville.

Heckendorf was generously supported by Google for Google Summer of Code 2014.

Chen was supported in part by the Department of Ecology and Evolutionary Biology at the University of Tennessee, Knoxville, and a grant from the National Science Foundation (MCB-1120370.)

Disclaimer

The findings and conclusions in this article have been formally disseminated neither by the U.S. Department of Energy, nor by the University of Tennessee, and as such should not be construed to represent any determination or policy of any University, Agency, and/or National Laboratory.

Cover art is *A Rhinoceros*, Richard Hall, ca. 1740.

This publication was typeset using LaTeX.

Contents

1 Introduction

The value of profiling code is indisputable. R's own `Rprof()` function is extremely useful, but its profiling capabilities are limited to simple timings of R functions. This is a very good starting point in performance analysis and can quickly help the R programmer focus in on bottlenecks. But for more experienced developers (especially those working with compiled code) additional performance information can be invaluable. Access to low-level hardware counter data can have tremendous impact when trying to understand and optimize performance of compiled code.

The **pbdPAPI** (Schmidt *et al.*, 2014) package offers access to this low-level hardware counter information by way of the high-level C library **PAPI** (Mucci *et al.*, 1999). Therefore, an installation of **PAPI** is required in order to use the package. For convenience, we bundle **PAPI** version 5.3.0 with **pbdPAPI**, which will install by default. However, with appropriate configure arguments, one can easily build **pbdPAPI** with an existing system installation of **PAPI**. See Section 2 for details.

The current main features of **pbdPAPI** include:

1. Simple, high-level interfaces that mimic R's own profiling syntax.
2. A low-level interface that mimics **PAPI**'s native calls, with extremely general functionality.

Note that the **pbdPAPI** package is not officially affiliated with the **PAPI** project in any way.

2 Installation

In this section, we will describe the various ways that one can install the **pbdPAPI** package.

2.1 Important Note

It is possible for **PAPI** and/or **pbdPAPI** to build, but still have no access to any counter information. For example, the package will build on a Raspberry Pi, but that hardware does not have any hardware counters. If the package installs without error, you can see how many hardware counters you have available by executing:

```r
library(pbdPAPI)
system.ncounters()
```

For a detailed list of all counters and whether or not your platform supports them, simply call

```r
library(pbdPAPI)
papi.avail()
```

2.2 WITHOUT a System Installation of PAPI

This is the default method of installation. Here, the **PAPI** library will automatically be built first as a static library, and then the **pbdPAPI** package will be built and linked against that static library. All of this is handled completely transparently, and should only go wrong if your system is not supported by **PAPI**. This is the simplest approach, and should cover most users. Simply build the package as you would any other:

Shell Command

```
R CMD INSTALL pbdPAPI_0.1-0.tar.gz
```

and using the **devtools** package:

```
library(devtools)
install_github(username="wrathematics", repo="pbdPAPI")
```

2.3 WITH an Existing System Installation of PAPI

If you already have a system installation of **PAPI** available, it makes more sense to link with that existing library. The one catch is that the static library *must* have been compiled with `-fPIC`, which is non-standard. To build an external **PAPI** library in this way, you should do so by first setting:

Shell Command

```
export CC="${CC} -fPIC"
```

Assuming that `CC` is set; if not, you can use `cc` in the right hand side.

To link with an external installation of **PAPI**, from the command line, execute:

Shell Command

```
R CMD INSTALL pbdPAPI_0.1-0.tar.gz \
    --configure-args="--enable-system-papi \
    --with-papi-home=location/of/PAPI/install"
```

and using the **devtools** package:

```
library(devtools)
install_github(username="wrathematics", repo="pbdPAPI",
    args="--configure-args='--enable-system-papi
          --with-papi-home=location/of/PAPI/install'")
```

3 Performance Measurement

3.1 flips and flops

flips This is intended to measure *instruction* rate through the floating point pipe with no massaging.

flops Perhaps the more well-known measurement is the rate of floating point *operations*.

Calling `system.flops(expr)` on a valid R expression `expr` will produce system and processor timings, the number of floating point operations, and the Mflops.

The `pca` demo shows off this functionality with principal components analysis. Executing:

```
demo("pca", "pbdPAPI")
```

on an Intel Sandy Bridge Core i5 produces the following outputs:

```
      m  n  measured theoretical difference pct.error   mflops
1 10000 50 211875901   202500000    9375901  4.425185 1637.195
```

Theoretical flops

flops = (# cores) ∗ (# of SSE units per core) ∗ (cycles / second) ∗ (# SSE operations per cycle)

single precision (divide by 2 for double precision).

So for this Intel Sandy Bridge Core i5 again as a reference, the

$$\begin{aligned}\text{Mflops} &= (4) * (2) * (3200Mhz) * 2 \\ &= 25600\end{aligned}$$

or about 25 Gflops (25,000,000,000 flops), aka, a hell of a lot of arithmetic.

3.2 Cache Misses and Cache Hits

Memory and Cache Computers operate at *billions* of cycles per second. Of course, those operations occur on data. A useful abstraction we use in thinking about processing data is you load the stuff up into ram and then the processor does things to it. This is usually fine, or at least convenient, but it's not accurate, as you are probably aware.

Another more accurate abstraction is that shown in Figure 1. In a sense, the magic really happens when things get into the CPU registers. But something that's in ram that you want to

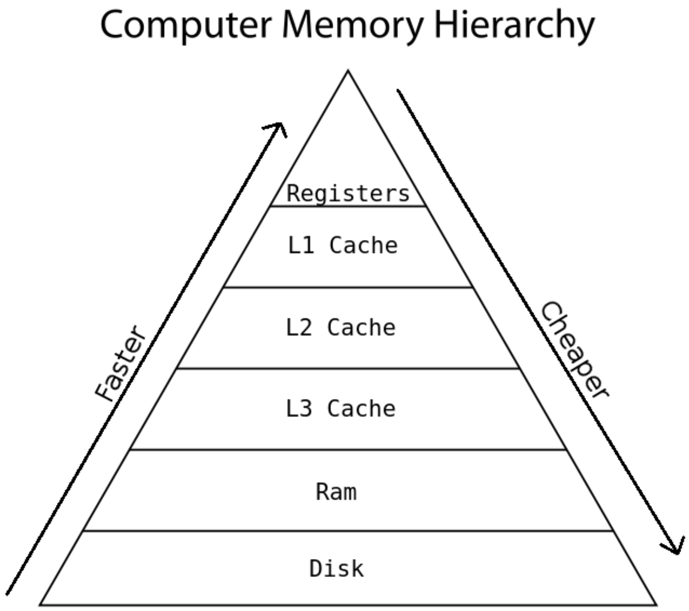


Figure 1: Memory Hierarchy

operate on, as it's headed to the CPU, it gets cached into various levels of (comparatively) fast access storage along the way. Not understanding this (simplified from reality) architecture can have can cause some pretty nasty performance side-effects. on your code. If you're unfamiliar with this, I would strongly encourage you to check out this really cool interactive visualization showing (relative) speeds of cache misses, also shown in Figure **??**. It too involves some hefty simplifications of how modern hardware actually works, so if this is at all confusing, let us all take a moment to pity the tragic life of the computer engineer.

Latency Numbers

Cache Misses Fundamentally, a cache miss occurs when the cache needs some piece of data to pass along to registers, but it isn't immediately available and has to go digging through ram (or god help you, disk) to get it. Cache misses are bad and reduce performance. You can't get rid of them, unless your entire problem fits into cache (which is glorious when it happens), but you can eliminate *unnecessary* cache misses being aware of how your data and algorithms interact with cache. When people tell you things like "R's matrices are column-major" or that you should loop over columns then rows, this is exactly what they are talking about.

4 Using pbdPAPI

The **pbdPAPI** package has 2 distinct interfaces for gathering performance data, one low-level and "one" high-level. The low-level interface is reminiscent of **PAPI**'s own native C interface, and "the" high-level interface is a collection of similar-looking wrappers around this interface.

4.1 High-Level Interface

4.2 Low-Level Interface

The low-level interface is provided for users who more familiar with hardware and may have questions about the operation of a block of code that do not easily fall into the the set of wrappers that make up the high-level interface. Even this comes in two forms. The first is the `system.event()` interface, which looks like the high-level interface functions, except that you must explicitly supply a **PAPI** events vector, with event names stored as strings. For a list of all possible events, and how to determine which are supported on your platform, see the R help `?papi.avail`.

As an example, suppose you wanted to measure the level 1 data cache misses and L1 data cache hits. After looking up the names for these events in the help file mentioned just above, you would simply call

```
events <- c("PAPI_L1_DCM", "PAPI_L1_DCH")
system.event(1+1, events=events)
```

Here, replace `1+1` with your preferred computation.

The final form of the low-level interface is essentially a deconstruction of the code making up the body of the `system.event()` function. This would allow you to skip several forms of error checking (if you want to do that for some reason). Primarily its purpose is to resemble the native C-level **PAPI** interface. We really don't recommend you use this, but it's there if you really believe you need it.

Revisiting the above example, you would call:

```
events <- c("PAPI_L1_DCM", "PAPI_L1_DCH")

papi.start(events=events)
1+1
papi.stop(events=events)
```

5 Examples

5.1 flops

Principal Components Analysis is a common statistical analysis technique. The implementation of PCA typically involves computing the singular value decomposition (SVD) of the input data and then projecting the data onto the right singular vectors. It is known that an SVD requires $6mn^2 + 20n^3$ floating point operations and that the projection onto the right singular vectors requires an additional $2mn^2$ operations (Golub and Van Loan, 1996). Finally, as PCA is usually performed on centered (and often scaled) data, we require an additional $2 * mn + 1$ operations for centering the data.

The **pbdPAPI** package contains the example `pca.r` as a package demo, which will perform a PCA on random data consisting of 10,000 observations and 50 predictors. The analysis uses R's ordinary `prcomp()` routine, and the performance is measured by **pbdPAPI**'s `system.flops()`. You can run this demo by calling

```
demo("pca", package="pbdPAPI")
```

An example output from this machine is:

```
      m  n  measured theoretical difference pct.error   mflops
1 10000 50 212538720   203500001    9038719   4.25274 2284.257
```

The **pbdPAPI** package has several other demonstrations that, like this, compare a theoretical floating point operations count against a measured count, and then display the Mflops the operation achieved. These demos include an inner product calculation `inner.r`, a matrix-matrix product `matprod.r`, and the fitting of a linear model in `regression.r`.

5.2 Cache Misses

To see the full source code described in this example, see the `cache_access` demo in **pbdPAPI**.

Consider the following example, where we will fill a matrix with 1's, first by looping over rows then columns, and then by looping over columns then rows. For maximum effect, we will be dropping to C by way of **Rcpp**. If you do not have **Rcpp** installed on your system, you can still follow along (even if you don't know C++), but you will not be able to recreate the timings locally.

You are probably aware that R matrices are stored in column-major fashion. What this means is that the data, as it is laid out in physical memory, is easier to go from the entry with index (i, j) to index $(i+1, j)$ than it is to go to index $(i, j+1)$. Accessing in the correct way minimizes the amount of work the computer needs to do in searching for the data your program needs. And in particular, this will help minimize cache misses, because if you routinely access data

```
bad_cache_access <- "
  int i, j;
  const int n = INTEGER(n_)[0];
  Rcpp::NumericMatrix x(n, n);


  for (i=0; i<n; i++)
    for (j=0; j<n; j++)
      x(i, j) = 1.;

  return x;
"
```

```
good_cache_access <- "
  int i, j;
  const int n = INTEGER(n_)[0];
  Rcpp::NumericMatrix x(n, n);


  for (j=0; j<n; j++)
    for (i=0; i<n; i++)
      x(i, j) = 1.;

  return x;
"
```

```
library(inline)

bad <- cxxfunction(signature(n_="integer"),
                   body=bad_cache_access, plugin="Rcpp")
good <- cxxfunction(signature(n_="integer"),
                    body=good_cache_access, plugin="Rcpp")
```

A quick check of run times shows something drastically different happening:

```
n <- 10000L

system.time(bad(n))
#   user  system elapsed
#  1.016   0.232   1.259

system.time(good(n))
#   user  system elapsed
#  0.201   0.155   0.357
```

So even though we (mathematically) are doing the exact same thing, the run times differ by a factor of 3.5. **pbdPAPI** allows us to more thoroughly see what's happening. If we use `system.cache()` to check the L1, L2, and L3 cache misses for each of these functions:

```
library(pbdPAPI)
n <- 10000L

system.cache(bad(n))
#$L1.total
#[1] 193580295
#
#$L2.total
#[1] 159442230
```

```
#
#$L3.total
#[1] 16895275

system.cache(good(n))
#$L1.total
#[1] 15552007
#
#$L2.total
#[1] 11580023
#
#$L3.total
#[1] 801150
```

it should be readily apparent what is going on now. The L1 cache misses differ by more than an order of magnitude, 194 million to 16 million!

```
system.cache(bad(n), events="l2.ratio")
# L2 cache miss ratio
#            0.815597

system.cache(good(n), events="l2.ratio")
# L2 cache miss ratio
#           0.7156862
```

References

Golub GH, Van Loan CF (1996). *Matrix Computations (Johns Hopkins Studies in Mathematical Sciences)(3rd Edition)*. 3rd edition. The Johns Hopkins University Press.

Mucci PJ, Browne S, Deane C, Ho G (1999). "PAPI: A portable interface to hardware performance counters." In *Proceedings of the Department of Defense HPCMP Users Group Conference*, pp. 7–10.

Schmidt D, Heckendorf C, Chen WC, Ostrouchov G, Patel P (2014). "pbdPAPI: Programming with Big Data – Performance Analysis Tools for R." R Package, URL http://cran.r-project.org/package=pbdPAPI.