Computing For Research:
Computational Efficiency and Precision with R

Matthew RP Parker

or:

Let's Improve Our Research Efficiency

Matthew RP Parker

It would be nice if we could improve our **research efficiency** by reducing the amount of time our research takes to do. We can accomplish this using two related computing topics:

- **Computational Efficiency**
  - how to reduce the computing resources required to solve problems
- **Computational Precision**
  - how to increase the precision of computing results

Before we begin:

- We will be framing this around **R** and research computing
- however, most of these ideas can be applied to other programming languages and to computing problems outside of research as well
- Throughout, I will say **efficiency** and **precision** to mean *computational* efficiency/precision, not statistical efficiency/precision
- Efficiency and Precision are often at odds with each other: increasing precision usually comes at the cost of decreased efficiency

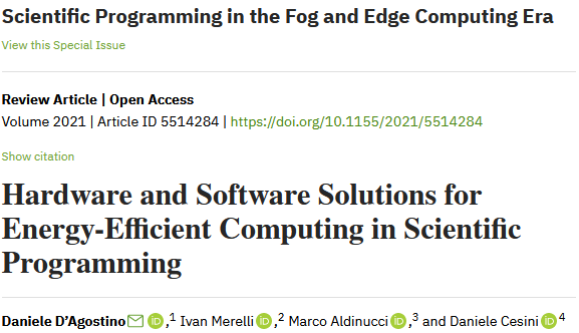
So what is **Computational Efficiency**?

- Efficiency is about reducing resource costs
- Efficiency comes in three flavours:
  - **Time Efficiency** (how long does it take to run?)
  - **Memory/Storage Efficiency** (how expensive is the hardware?)
  - **Energy Efficiency** (economic and environmental footprint?)

Here are a few points regarding **Energy efficiency** before we focus entirely on Memory and Time efficiency:

- Energy efficiency is extremely important for large scale projects such as training very large learning models
- New technologies and engineering provide large increases in efficiency over time
- Without upgrading hardware, there is still something we can do:
  - increasing Time efficiency directly reduces Energy costs
  - recycling energy through reuse of waste heat from compute clusters
  - scheduling heavy computing tasks during off-peak electricity usage hours

If you are interested in Energy efficiency, you can read this primer on the topic[1]:



Scientific Programming in the Fog and Edge Computing Era

View this Special Issue

Review Article | Open Access

Volume 2021 | Article ID 5514284 | https://doi.org/10.1155/2021/5514284

Show citation

Hardware and Software Solutions for Energy-Efficient Computing in Scientific Programming

Daniele D'Agostino[1], Ivan Merelli[2], Marco Aldinucci[3] and Daniele Cesini[4]

[1]Daniele D'Agostino, Ivan Merelli, Marco Aldinucci, Daniele Cesini, "Hardware and Software Solutions for Energy-Efficient Computing in Scientific Programming", Scientific Programming, vol. 2021, Article ID 5514284, 9 pages, 2021. https://doi.org/10.1155/2021/5514284

So how do we measure efficiency?

- Memory efficiency is measured in **number of bytes** of drive space and peak RAM usage.
- Time efficiency can be measured in terms of theoretical **algorithm operations** using Big Oh (*theoretical efficiency*)
- Time efficiency can also be measured in terms of **compute time** (*realized efficiency*)
- Big Oh is useful for understanding how a solution will scale with increasing size (depends only on the algorithm complexity)
- Compute time is the real world time cost of the solution (depends on both the algorithm complexity, the specific hardware used, the specific software used, and other factors such as temperature)

Measure Efficiency Using Tools:

- We can measure how efficient a function is overall using the R packages **bench** and **profvis**
  - `bench::mark()` measures the time, the memory footprint, of one or more functions
  - we will use `bench::mark()` throughout this talk for illustration of different ideas
- We can analyze the efficiency of a function line by line using **profvis**
  - `profvis()` is a wrapper for Rprof
  - `profvis()` creates a flame graph with a breakdown of resource usage for each line of code and each function call
  - it is invaluable for finding bottlenecks and discovering why your code is taking minutes to run instead of seconds

Measure Efficiency Using Tools:

- So which profiling method should you use?
- I think you should use both!
- `bench::mark()` answers **WHICH** function uses more time or memory
- `profvis()` answers **WHY** a function uses more time or memory

For loops are a contentious subject among R programmers, but it boils down to this:

Good for loops are good. Bad for loops are bad.

- **Should you use for loops:**
- for loops in R are not inherently bad
- however, they can reduce efficiency when misused!
- be aware of memory usage, allocation is an expensive operation
- pre-allocate memory outside of loops

Let's compare a bad and a good for loop using the R package **bench**:

```r
library(bench)

size=1e4
bench::mark(
{
  values = NULL
  for(i in 1:size) {
    values = c(values, rnorm(1,0,1))
  }
},
{
  values = numeric(size)
  for(i in 1:size) {
    values[i] = rnorm(1,0,1)
  }
},
iterations = 10)
```

- `size=1e4` is the number of random variables we are creating
- 1st for loop: the variable **values** grows in length by one at each iteration
- 2nd for loop: the variable **values** uses pre-allocated memory

Let's compare a bad and a good for loop using the R package **bench**:

```
library(bench)

size=1e4
bench::mark(
{
  values = NULL
  for(i in 1:size) {
    values = c(values, rnorm(1,0,1))
  }
},
{
  values = numeric(size)
  for(i in 1:size) {
    values[i] = rnorm(1,0,1)
  }
},
iterations = 10)
```

- 1st for loop: `values` grows in size at each iteration
- 2nd for loop: `values` memory is pre-allocated
- 2nd is 12x more time efficient, and 17x more memory efficient

```
# A tibble: 2 × 13
  expression      min median `itr/sec` mem_alloc `gc/sec` n_itr  n_gc total_time
  <bch:expr>   <bch:t> <bch:>     <dbl> <bch:byt>    <dbl> <int> <dbl>   <bch:tm>
1 { values =… 150.8ms  180ms      5.34   406.3MB     10.2    10    19      1.87s
2 { values =…  14.1ms   15ms     54.9     24.4MB     5.49    10     1   182.12ms
```

SFU

Take advantage of vectorized functions where possible.

```
size=1e4
bench::mark(
{
  set.seed(123)
  values = numeric(size)
  for(i in 1:size) {
    values[i] = rnorm(1,0,1)
  }
  values
},
{
  set.seed(123)
  values = rnorm(size,0,1)
},
iterations = 10)
```

- Vectorization is the main reason people say "don't use for loops in R"
- 1st for loop: does **not** take advantage of vectorization
- 2nd no loop: uses vectorized function
- 2nd is 32x more time efficient, and 293x more memory efficient

	expression	min	median	`itr/sec`	mem_alloc	`gc/sec`	n_itr	n_gc	total_time
	<bch:expr>	*<bch:t>*	*<bch:t>*	*<dbl>*	*<bch:byt>*	*<dbl>*	*<int>*	*<dbl>*	*<bch:tm>*
1	{ set.see…	12.7ms	12.8ms	70.5	24.4MB	7.83	9	1	127.67ms
2	{ set.see…	395.5µs	397.1µs	2494.	83.2KB	0	10	0	4.01ms

`lapply`, `sapply`, `mapply`, `map`, etc., can all be used to replace for loops. Sometimes faster, sometimes slower.

```
size=1e4
bench::mark(
{
  set.seed(123)
  values = numeric(size)
  for(i in 1:size) {
    values[i] = rnorm(1,0,1)
  }
  values
},
{
  set.seed(123)
  values = sapply(1:size, function(X) {rnorm(1,0,1)})
},
iterations = 10)
```

- for loop: an explicit loop
- sapply: an implicit loop
- very comparable, but for loop performs marginally better here

	expression	min	median	`itr/sec`	mem_alloc	`gc/sec`	n_itr	n_gc	total_time
	<bch:expr>	*<bch:>*	*<bch:>*	*<dbl>*	*<bch:byt>*	*<dbl>*	*<int>*	*<dbl>*	*<bch:tm>*
1	{ set.seed(…	13ms	13.2ms	68.8	24.4MB	7.64	9	1	131ms
2	{ set.seed(…	19.2ms	19.7ms	50.2	24.7MB	5.58	9	1	179ms

Sometimes code can be run simultaneously to increase time efficiency.
The R package **doParallel** allows you to do this with ease.

- `makeCluster(n)` creates a cluster of `n` compute cores
- `foreach` mimics the structure of a for loop
- `%dopar%` instructs R to split the iterations over available compute cores

```
library(doParallel)
cl <- makeCluster(10)
registerDoParallel(cl)

foreach(i=1:10) %dopar% {
  f()
}
```

SFU

Here we have a contrived example:

```
library(doParallel)
cl <- makeCluster(10)
registerDoParallel(cl)

# a function that takes 0.1 second to compute
f <- function() { Sys.sleep(0.1) }
bench::mark(
{
  for(i in 1:10) {
    f()
  }
},
{
  foreach(i=1:10) %dopar% {
    f()
  }
},
iterations = 10, check = FALSE)
```

	expression	min	median	`itr/sec`	mem_alloc	`gc/sec`	n_itr	n_gc	total_time
	<bch:expr>	<bch:tm>	<bch:tm>	<dbl>	<bch:byt>	<dbl>	<int>	<dbl>	<bch:tm>
1	{ for (i in 1:10…	1.08s	1.09s	0.916	9.56KB	0	10	0	10.91s
2	{ foreach(i = 1:…	108.13ms	109.24ms	9.17	763.61KB	0	10	0	1.09s

- the parallel loop is 10x more time efficient
- the for loop is 80x more memory efficient
- parallelization is usually a trade-off: spend more energy (compute cores) and spend more memory to reduce compute time

Sometimes, R code is just too slow. The truly efficient functions in R are generally written in c/cpp.

- we can use profiling (`profvis()`) to find the bottlenecks in our code
- if we find one function or section of code is slowing everything else down, we can consider writing it in a faster, lower level programming language
- **Rcpp** is an R library which allows you to easily[2] write your own cpp functions in R

[2] ease of use requires some knowledge of cpp

Often we choose modelling frameworks to fit a research problem. Other times we have more latitude to choose...

Eg: parameter estimation in a likelihood setting. Could use frequentist approach (MLE), or Bayesian (probabilistic programming/MCMC).

- if one framework is slow, the other might be comparatively fast!
- **Bayesian is often less efficient than frequentist approach** due to large iterations needed to estimate the posterior distribution
- however, large numbers of latent states can be computationally challenging for MLE (integration from likelihood), while **sampling latent states is relatively inexpensive**
- it can be **worthwhile to test different frameworks** when running into large computation times with your particular application

When the usual solutions aren't enough, sometimes you can transform your compute problem into one which is more efficiently solved.

- solve linear systems with efficient libraries: **RcppEigen**, **RcppArmadillo**
- recognize convolutions which are VERY slow to compute, instead solve using FFT[3]
- use asymptotic statistics to find more efficient solutions
- Finding ways to apply fast algorithms to common problems is a very important field of research! If you find a fast method to solve a common problem, you can publish your new algorithm, and help a lot of researchers to improve their own research efficiency

[3] Parker, M.R.P., Cowen, L.L.E., Cao, J., Elliott, L.T. Computational Efficiency and Precision for Replicated-Count and Batch-Marked Hidden Population Models. JABES 28, 43-58 (2023). https://doi.org/10.1007/s13253-022-00509-y

What is **Precision**?

- **In statistics:**
- accuracy is how close your point estimate is to the true value: $|\bar{X} - \mu|$
- precision is how closely clustered your point estimates are $1/\hat{\sigma}_X^2$
- **In computation:**
- precision is how accurately values are REPRESENTED during computing, and how accurate the RESULTS are at the end of computing
- precision has an upper bound based on the number of bits used to represent information
- precision usually decreases every time a calculation is performed

SFU

In R, numbers can be either Integer or Numeric
(defaulting to numeric in most cases)

- `integer` vs `numeric`

```
# integer 32 bit precision
.Machine$integer.max # [1] 2147483647
object.size(1:10) # 96 bytes

# numeric 64 bit precision
.Machine$double.xmax # [1] 1.797693e+308
object.size(as.double(1:10)) # 176 bytes
```

- `integer` vectors use less memory
- `numeric` numbers have higher precision

Overflow occurs when a value is too large to store using the current precision level:

Integer overflow produces a warning and an `NA`:

```
> as.integer(.Machine$integer.max) * 2L
[1] NA
Warning message:
In as.integer(.Machine$integer.max) * 2L : NAs produced by integer overflow
```

Numeric overflow produces a result of "`Inf`", which stores no numeric information:

```
> .Machine$double.xmax * 2
[1] Inf
```

Underflow means something different for integers and numerics:

- **Integer Underflow:** the integer is too small to represent (R calls this an overflow, but it is often referred to as integer underflow elsewhere)

```
> as.integer(-.Machine$integer.max)
[1] -2147483647
> as.integer(-.Machine$integer.max)-1L
[1] NA
Warning message:
In as.integer(-.Machine$integer.max) - 1L :
  NAs produced by integer overflow
```

- **Numeric Underflow:** the number is too small **in magnitude** to represent, and is thus truncated to zero

```
> .Machine$double.eps
[1] 2.220446e-16
> .Machine$double.eps / 1e307
[1] 1.976263e-323
> .Machine$double.eps / 1e308
[1] 0
```

Numeric underflow is a major issue in statistical research computing: a very small probability is **NOT** a zero probability.

- log transforms usually work well to solve this problem:

```
> log10(.Machine$double.eps)
[1] -15.65356
> log10(.Machine$double.eps) - log10(1e307)
[1] -322.6536
> log10(.Machine$double.eps) - log10(1e308)
[1] -323.6536
```

- however, logs can easily fail when addition is needed:

```
> .Machine$double.eps / 1e308 +
  .Machine$double.eps / 1e308
[1] 0
> log10(10^(log10(.Machine$double.eps) - log10(1e308)) +
  10^(log10(.Machine$double.eps) - log10(1e308)))
[1] -Inf
```

- you can use our R package **quickNmix** with function `logSumExp` to solve this problem
- **LSE (LogSumExp)** takes advantage of the highest precision floating point region to provide computational stability
- Consider two very small numerics x and y, stored as $\ell x = \log(x)$ and $\ell y = \log(y)$
- Then LSE is simply:

$$\text{LSE}(\ell x, \ell y) = v + \log(\exp(\ell x - v) + \exp(\ell y - v)), \qquad v = \max(\ell x, \ell y)$$

- WLOG, suppose that $\ell x > \ell y$, then a little algebra gives:
- $\text{LSE}(\ell x, \ell y) = \ell x + \log[1 + \exp(\ell y - \ell x)]$
- let $z = \exp(\ell y - \ell x)$, then we have $\log(1 + z)$ bounded between 0 and 1 (an interval with highest computational precision according to IEEE 754)

```
> quickNmix:::logSumExp(c(log(.Machine$double.eps) - log(1e308),
+                         log(.Machine$double.eps) - log(1e308)))
[1] -744.5467
```

Sometimes, 64 bit precision is just not enough!

Examples which can require very high precision include:

- **Astrophysics** modelling such as binary star systems and gravitational forces
- **climate modelling** and **weather systems** (loss of precision accumulates with time)
- physical modelling of **interacting particles** (especially at high energies)
- studying **chaotic systems** (even very small inaccuracies can invalidate research results)
- probability systems with **large numbers of discrete states** (when asymptotics and approximations cannot be used)

MPFR is an acronym for "Multiple Precision Floating-Point Reliably"

- **RMPFR:** an R package for high precision computing
- based on GNU MPFR (open source, portable to work on many platforms)
- R stores numerics in physical computer registers
  (precision stuck at 64 bits for standard modern computers)
- RMPFR instead uses RAM to store the high-precision numerics
  (precision limited only by system RAM)
- however, there is an enormous cost to using RMPFR, since it replaces a **hardware** solution for a **software** solution
  (RMPFR is always slower, and can be hundreds of thousands of times slower)
- solutions like RMPFR should be a last resort, because computation times quickly become infeasible for even small sized problems

SFU

We can use RMPFR or LSE to solve the same problems, so lets compare!

```
bench::mark(
{
  x = Rmpfr::mpfr(.Machine$double.eps, precBits = 128)
  y = Rmpfr::mpfr(1e308, precBits = 128)

  log(x/y + x/y)
},
{
  lx = log(.Machine$double.eps)
  ly = - log(1e308)
  quickNmix:::logSumExp(c(lx-ly, lx-ly))
},
iterations = 100, check = FALSE)
```

```
# LSE: -744.5467
# 128 bit precision: -744.546714850723281468799269866591636269 9
```

```
# A tibble: 2 x 13
  expression          min median `itr/sec` mem_alloc `gc/sec` n_itr  n_gc total_time result memory
  <bch:expr>        <bch:> <bch:>     <dbl> <bch:byt>    <dbl> <int> <dbl>   <bch:tm> <list> <list>
1 { x = Rmpfr::mp… 1.16ms 1.23ms       803.     1.6KB     8.11    99     1      123ms <NULL> <Rprofmem>
2 { lx = log(.Mac…    6µs  6.6µs    144113.        0B        0   100     0      694µs <NULL> <Rprofmem>
# ℹ 2 more variables: time <list>, gc <list>
```

- our toy example was $log(x/y + x/y)$, where x/y underflows at 64 bits precision
- now we **compare RMPFR** with 128 bits of precision against **LogSumExp**

LSE is 180x more time efficient here than RMPFR, and RMPFR uses 1.6kb more RAM

Recap:

- Efficiency and Precision go hand in hand:
- Computational Efficiency is a necessary consideration for many scientific computing tasks
- more efficient computing leads to more efficient research
- Computational Precision is a necessary consideration for all scientific computing tasks
- low precision can invalidate research results
- Compute Precision comes at a heavy cost in terms of Computational Efficiency
- higher precision causes lower efficiency
- The difficulty is to **find a trade-off where precision is high enough, but compute time is feasible**