aprof: Amdahl's profiler, directed optimization made easy

Marco D. Visser

September 4, 2014

Abstract

Amdahl's profiler or *aprof* is meant to assists the evaluation of whether and where to focus code optimization, using Amdahl's law and visual aids based on line profiling. The main goal is to help to balance development vs. execution time. Amdahl's profiler does this by enabling rapid identification of bottnecks within sections of code while simultaniously projecting potential gains. This document explains some of the basic ideas behind aprof: how it applies Amdahl's law to project optimization gains and how it organises profiling output files. It also provides a simple guide towards profiling and code optimization, while identifying somes common inefficiencies in R.

1 Profiling

"We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil"

- Donald Knuth (1974). "Structured Programming with go to Statements". ACM Journal Computing Surveys 6 (4): 268.

Empirical studies in computer science show that inefficiency is often rooted in a small proportion of code (**?**). Therefore, identifying which part of the code takes the most time can allow effective and targeted optimization efforts. Code profilers help towards this end; these are software engineering tools that measure the performance of different parts of a program as it executes (**?**). In this section we will describe how to use a statistical profiler (called *Rprof* in R), that uses operating system interrupts to probe which routines are active at regular intervals and counts which R expressions are consuming the most resources.

In cases where random-access memory (RAM) storage, rather than processor time, poses the most pressing limit, one can use a memory profiler (e.g. Rprofmem) to obtain similar statistics for memory efficiency. Simpler, though less informative, tools to time and track resource use include the R functions system.time() and object.size(). The former returns the time used by both R and the operating system (for communication between devices, file writing, etc.), while the latter gives an approximation of the memory usage of objects.

Throughout the tutorial we will make heavy use of R's profiling capabilities. R uses a so-called statistical profiler (Rprof), which probes at predefined intervals which routines are active and so we can count which R expressions are consuming most of the system's resources. For example, profiling a program consisting of a two commands will tell you exactly how much time is spent in each command, and therefore which of the two you should take a closer look at. We have also developed software in the form of an R package (aprof: Amdahl's profiler) to organize the output from the R profiler using visual tools to identify bottlenecks (as illustrated in Figure **??**).

In this document we will use the package aprof to organise the output from R's standard profiler and makes it much easier to rapidly identify bottlenecks in your program code. An example of using *aprof* is given below. The package is available at CRAN.

To use this profiler, we first need to make a simple program which we want to profile. Here is an example of a program (InterpreterQuirks) that executes the calculation $N/(1+N)$ many times with either parentheses or brackets, and different amounts of each. The function highlights some of the quirks of an interpreted language as explained earlier (section **??**).

```
## Rcode7.R

InterpreterQuirks<-function(N){
  for (i in 1:N) { N/(1+N) }
  for (i in 1:N) { (((N/(1+N))))}
  for (i in 1:N) { N/{1+N} }
  for (i in 1:N) { {{{N/{1+N}}}} }
}

## Save the function to a source code file
dump("InterpreterQuirks",file="InterpreterQuirks.R")
```

Next we use Rprof to start profiling *InterpreterQuirks*. We will first reload our saved file so the code lines in the R environment, as this ensure that the version in R and the saved file on the disk match up exactly.

```
## Rcode8.R

source("InterpreterQuirks.R")
```

Then we switch on R's profiler *Rprof*, and because we want to know which lines of InterpreterQuirks are the slowest we need to make sure the option *line.profiling* is set to TRUE. Here we use a time interval between samples of 0.02.

```
## Rcode9.R

Rprof(file="InterpreterQuirks.out",interval = 0.02,
          line.profiling =TRUE)
```

```
## Rcode12.R

plot(IntQuirksAprof)
          ## Error:  object 'IntQuirksAprof' not found
```

Figure 1: Output from *aprof*'s PlotExcDens function. It shows the source code for our function *InterpreterQuirks* in the left panel with the execution time per line of code in the right panel. We see a classical interpreted language problem, where more parentheses and brackets mean that the R interpreter has to evaluate more often, while some symbols (here "{" and "}") are interpreted faster (i.e. the lookup speed for these symbols is likely faster). Use *profile.plot* for larger pieces of code.

```
InterpreterQuirks(N=1e5) # run 100 thousand times

Rprof(append=F) # stop profiling
```

Next we load our package *aprof* ("Amdahl's profiler"). We want to visualize the time spent in each line of code using *aprof*'s standard plot function on our program *InterpreterQuirks* (see ?*plot.aprof* for details). However before we start we need to use the function *aprof* to make an "*aprof* object". This object will contain the profiling information and the source file "*InterpreterQuirks.R*".

```
## Rcode10.R

require(aprof)

## Loading required package:  aprof

##make an object of the aprof class
IntQuirksAprof <- aprof("InterpreterQuirks.R","InterpreterQuirks.out")

## Warning:  cannot open file 'InterpreterQuirks.out':  No such
file or directory
## Error:  cannot open the connection
```

Now that we have a standard "*aprofobject*" we can display some basic information about the profiling exercise by simply typing the name of the "*aprofobject*" and hitting return.

```
## Rcode11.R

IntQuirksAprof

## Error:  object 'IntQuirksAprof' not found
```

Next we can use the standard *plot* function on this object to display the execution time per line. The following code should return figure 1.

Plotting an *aprof* object is useful when your program is relatively small however, when your code consists of hundreds of lines, a better function would be *profile.plot*. Go ahead and use it on our *aprof* object "IntQuirksAprof" to see what it does (type ?*profile.plot* in the command line for details). Another useful feature is to summarize our *aprof* object, which gives us the theoretical maximum attainable speed-up for each line of code (see ?*summary.aprof* for details).

```
###Rcode13.R

summary(IntQuirksAprof)

## Error:  object 'IntQuirksAprof' not found
```

Using this information we can easily decide where to focus our efforts and, maybe more importantly, decide whether it is worth the effort to optimize the code. As we can see in the uppermost table, line 4 would be the most promising to work on, as it shows the greatest improvement for each of the sets of speed-up factors (1 - 16×). These numbers effectively tell us what the predicted overall speed-up of the program would be when we focus on a single line. That is, if we improve the execution time of a given line by a factor S (S times faster), the table predicts how much this improvement will affect the overall run-time of the entire program. In the above example we see, however, that the gain is minimal and even when the speed-up factor goes to infinity (effectively when the run time of that line becomes 0; $\lim S \to \infty$) we can only achieve a maximum speed-up of 1.39. Or in other words, if we were to infinitely improve the code in line number 4 we would only improve the overall program by 39%. Infinitely faster is not something we are likely to achieve, but also for the more practical speed-up factors we also see that a factor of 16 improvement is hardly an improvement over a factor of 4. In such cases it may not be worthwhile to either purchase computing resources (parallel machines or execution time on a cluster) or spend time optimizing code. Naturally, in this simple example the overall execution time is so small that we won't spend time optimizing it. However, as we usually would profile a simplified version of a larger program where the execution time may be considerably larger, even a 36% improvement in execution time may be worthwhile.

2 Guidelines for optimization

One should consider optimization only after the code works correctly and produces trustworthy results (**?**). Once this has been achieved, ask yourself whether cruder code would in the end be more time-efficient?

It is important to recall a fact that is recognized by programmers:

"everyone knows that debugging is twice as hard as writing a program in the first place. So if you're as clever as you can be when you write it, how will you ever debug it?" (**?**).

Often, only a portion of the code can be optimized. Applying Amdahls law (**?**) can help make the decision of whether to pursue the optimization. Consider a computation that requires time T_0 to complete and that we are interested in optimizing a portion of that code. If the portion to optimize requires a fraction α of the total time (T_0) to complete, and can be improved by a factor I we can calculate the overall improved execution time (T_i) of the entire computation, as:

$$T_i = (1 - \alpha)T_0 + (\alpha T_0)/I \tag{1}$$

The first term on the right hand side of (1) describes the amount of time required to process the portion of the code that was not optimized. The second term describes the amount of time to process the optimized portion. It follows that the realized speedup factor (S) can be expressed as:

$$S = 1/((1 - \alpha) + \alpha/t) \tag{2}$$

Amdahl's law reveals that the effect of optimization on the overall program performance will depend on both the improvement caused by the optimization (I), and the fraction of the program that can be improved (α) (Fig. 2). For example, suppose a specific operation within R's program code originally took 50% of the execution time ($\alpha = 0.5$), and this operation can be rewritten in a compiled language causing an expected factor 9 speed up (I = 9). Despite the substantial improvement in the operation, the overall improvement is far less (S = 1.8 times faster). In computer science this is seen as the major insight of Amdahls Law (**?**): unless one improves a very large fraction of the overall system, the realized improvement in execution time will be considerably less.

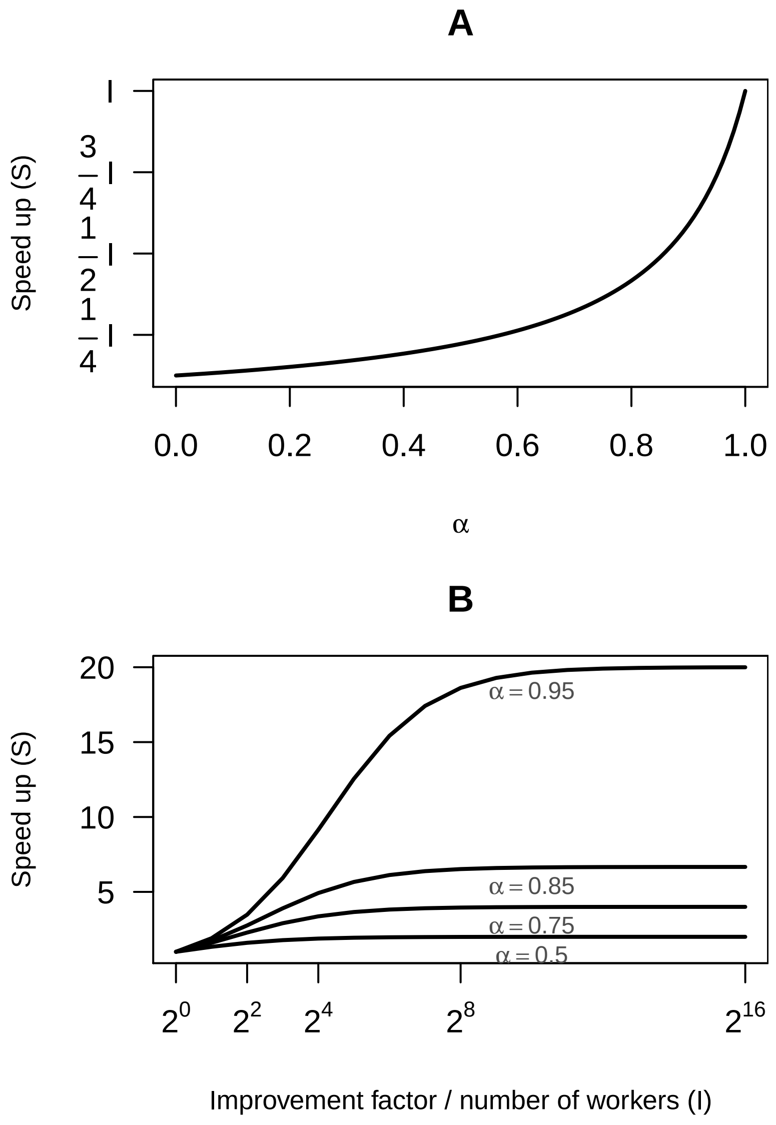

Figure 2: Maximal improvement in program execution speed, or "speed up factor" (S) as predicted by Amdahl's law, either through optimizing code by a factor I or running computations in parallel on I workers. (a) Speed up as function of α, the proportion of the total execution time taken up by the section of code, which is improved by factor I. We see that only when a very large fraction (α) of the execution time is consumed by the section of code to be optimized, will the overall speed gain come close to I. (b) Total expected speed-up gain for different levels of α as a function of I, which can either represent the improvement factor or number of parallel calculations conducted. We see that there are theoretical limits to the maximal improvement in speed and that it is crucially and asymptotically dependent on α. This is also know as the "law of diminishing returns". Predictions based on Amdahl's law are subject to the scaling of the problem.

Here is an example of some basic *aprof* operations, showing the difference in system resources usage between pre-allocating objects in the memory of growing them

```
require(aprof)
# create function to profile
     foo <- function(N){
             preallocate<-numeric(N)
             grow<-NULL
              for(i in 1:N){
                  preallocate[i]<-N/(i+1)
                  grow<-c(grow,N/(i+1))
                 }
             }

     #save function to a source file and reload
     dump("foo",file="foo.R")
     source("foo.R")


     # Profile the function
     Rprof("foo.out",line.profiling=TRUE)
     foo(1e2)
     Rprof(append=FALSE)

     # Create a aprof object
     fooaprof<-aprof("foo.R","foo.out")

## Error: Rprof outputs appears to be empty, were enough samples
made by the profiler?

     fooaprof

## Error: object 'fooaprof' not found
```

The table above gives some basic output, and shows the lines that were sampled by the profiler, and the amount of time spent in each line. In this simple example it's clear that line 7, takes the most time and should be the target of optimization. This is also shown in the next figure:

```
plot(fooaprof)

## Error: object 'fooaprof' not found
```