Using the usl package

Analyze system scalability in R with the Universal Scalability Law

Stefan Möding

February 15, 2013

The Universal Scalability Law is used to quantify the scalability of hardware or software systems. It uses sparse measurements from an existing system to predict the throughput for different loads and can be used to learn more about the scalability limitations of the system. This document introduces the 'usl' package for R and shows how easily it can be used to perform the relevant calculations.

Contents

1 Version 1

2 Introduction 2

3 Background 2

4 Examples of Scalability Analysis 3

4.1 Case Study: Hardware Scalability . . . 3

4.2 Case Study: Software Scalability . . . 9

References 11

1 Version

This document describes version 1.1.0 of the 'usl' package.

2 Introduction

Every system architect faces the challenge to deliver an application system that meets the business requirements. A critical point during the design is the scalability of the system. Scalability can informally be defined as the ability to support a growing amount of work. This can have two facets with respect to a computer system.

On one hand there is software scalability where the focus is about how the system behaves when the demand increases, i.e., when more users are using it or more requests have to be handled. On the other hand there is hardware scalability where the behavior of an application system running on larger hardware configurations is investigated.

A system is said to scale if it handles the changed demand or hardware environment in a reasonable efficient and practical way.

The Universal Scalability Law (USL) has been developed by Dr. Neil J. Gunther to allow the quantification of scalability for the purpose of capacity planning. It provides an analytic model for the scalability of a computer system. A comprehensive introduction to the Universal Scalability Law including the mathematical grounding has been published in [Gun07].

3 Background

Dr. Gunther shows in [Gun07] how the scalability of every computer system can be described by a common rational function. This function is *universal* in the sense that it does not assume any specific type of software, hardware or system architecture.

Equation 1 has the Universal Scalability Law where $C(N) = X(N)/X(1)$ is the relative capacity given by the ratio of the measured throughput $X(N)$ for load N to the throughput $X(1)$ for load 1.

$$C(N) = \frac{N}{1 + \sigma(N-1) + \kappa N(N-1)} \tag{1}$$

The denominator consists of three terms that all have a specific physical interpretation:

Concurrency: The first term models the linear scalability that would exist if the different parts of the system (processors, threads ...) could work without any interference caused by interaction.

Contention: The second term of the denominator refers to the contention between different parts of the system. Most common are issues caused by serialization or queueing effects.

Coherency: The last term represents the delay induced by keeping the system in a coherent and consistent state. This is necessary when writable data is shared in different parts of the system. Predominant factors for such a delay are caches implemented in software and hardware.

In other words: σ and κ represent two concrete physical issues that limit the achievable speedup for parallel execution. Note that the contention and coherency terms grow linearly respectively quadratically with N and therefore their influence becomes larger with an increasing N.

Due to the quadratic characteristic of the coherency term there will be a point where the throughput of the system will start to go retrograde, i.e., will start to decrease with further increasing load.

In [Gun07] Dr. Gunther also proves that Equation 1 is reduced to Amdahl's Law for $\kappa = 0$. Therefore the Universal Scalability Law can be seen as a generalization of Amdahl's Law for speedup in parallel computing.

Using a set of measurements for the throughput X_i at different loads N_i we can solve this nonlinear equation to estimate the coefficients σ and κ.

The computations used to solve the equation for the measured values are discussed in [Gun07]. The 'usl' package has been created to subsume the steps into one simple function call. This greatly reduces the manual work that previously was needed to perform the scalability analysis.

The function provided by the package also includes some sanity checks to help the analyst with the data quality of the measurements.

4 Examples of Scalability Analysis

The following sections present some examples of how the 'usl' package can be used when performing a scalability analysis. They also explain typical function calls and their arguments.

4.1 Case Study: Hardware Scalability

The 'usl' package contains a demo dataset with benchmark measurements from a raytracer software[1]. The data was gathered on an SGI Origin 2000 with 64 R12000 processors running at 300 MHz.

[1] `http://sourceforge.net/projects/brlcad/`

For the benchmark the software computed a number of reference images with different levels of complexity. The measurements contain the average number of calculated ray-geometry intersections per second for the number of used processors.

It is important to note that with changing hardware configurations the relative number of *homogeneous* application processes per processor is to be held constant. So when k application processes were used for the N processor benchmark then $2k$ processes must be used to get the result for $2N$ processors.

Start the analysis by loading the 'usl' package and look at the supplied dataset.

```
R> library(usl)
R> data(raytracer)
R> raytracer
```

```
   processors throughput
1           1         20
2           4         78
3           8        130
4          12        170
5          16        190
6          20        200
7          24        210
8          28        230
9          32        260
10         48        280
11         64        310
```

The data shows the throughput for different hardware configurations covering the available range from one to 64 processors. We can easily see that the benefit for switching from one processor to four processors is much larger than the gain for upgrading from 48 to 64 processors.

To get a grip on the data we create a simple scatterplot.

```
R> plot(throughput ~ processors, data = raytracer, pch = 16)
```

Figure 1 shows the throughput of the system for the different number of processors. This plot is a typical example for the effects of *diminishing returns* as it clearly shows how the benefit of adding more processors to the system gets smaller for higher numbers of processors.

Our next step builds the USL model from the dataset. The `usl()` function creates an S4 object that encapsulates the computation.

The first argument is a formula with a symbolic description of the model we want to analyze. In this case we would like to analyze how the "throughput" changes with regard to the number of "processors" in the system. The second argument is the dataset with the measured values.

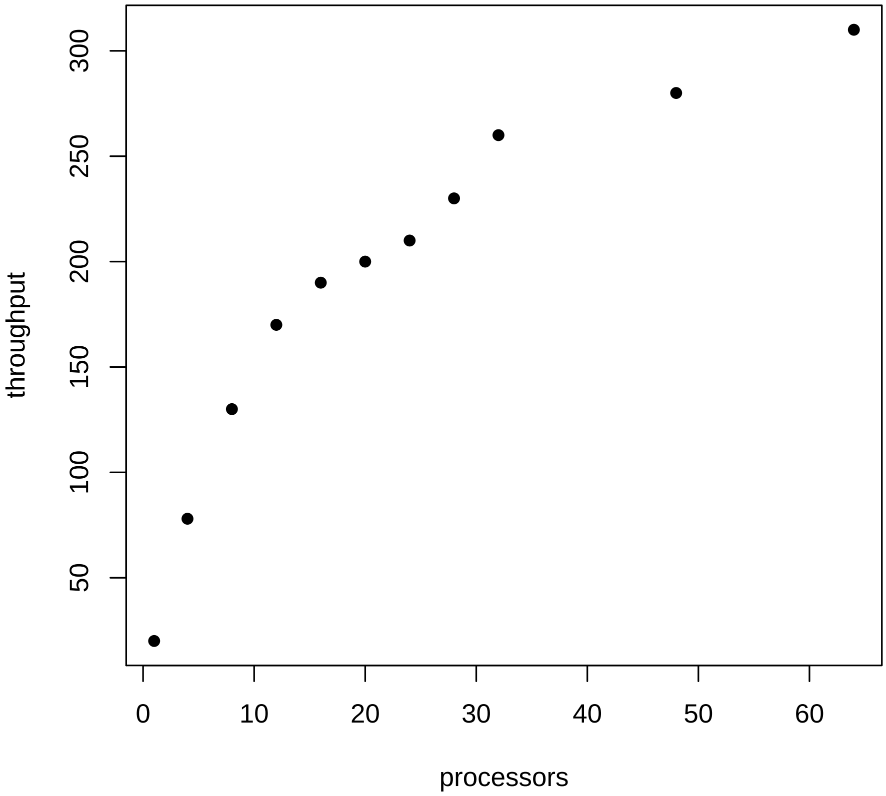

Figure 1: Measured throughput of a raytracing software in relation to the number of available processors

```
R> usl.model <- usl(throughput ~ processors, data = raytracer)
```

The model object can be investigated with the `summary()` function.

```
R> summary(usl.model)

Call:
usl(formula = throughput ~ processors, data = raytracer)

Scale Factor for normalization:   20

Efficiency:
   Min     1Q Median     3Q    Max 
 0.242  0.408  0.500  0.760  1.000 
```

```
Residuals:
   Min     1Q Median     3Q    Max
-12.93  -5.23   3.08   9.00  15.25

Coefficients:
   sigma     kappa
5.00e-02  4.71e-06

Multiple R-squared: 0.988,        Adjusted R-squared: 0.987
```

The output of the `summary()` function shows different types of information.

- First of all it includes the call we used to create the model.
- It also includes the scale factor used for normalization. The scale factor is used internally to adjust the measured values to a common scale. It is equal to the value $X(1)$ of the measurements.
- The efficiency tells us something about the work that is performed per processor. Intuition tells us that two processors could be able to handle double the work of one processor but not more. Calculating the ratio of the workload per processor should therefore always be less or equal to 1. In order to verify this we can use the distribution of the efficiency values in the summary.
- We are performing a regression on the data to calculate the coefficients and therefore we determine the residuals for the fitted values. The distribution of the residuals is also given as part of the summary.
- The coefficients σ and κ are the result that we are essentially interested in. They tell us the magnitude of the contention and coherency effects within the system.
- Finally R^2 estimates how well the model fits the data. We can see that the model is able to explain more than 98 percent of the data.

The function `efficiency()` extracts the specific values so we can have a closer look at the specific efficiencies of the different processor configurations.

```
R> efficiency(usl.model)
```

```
     1      4      8     12     16     20     24     28     32     48
1.0000 0.9750 0.8125 0.7083 0.5938 0.5000 0.4375 0.4107 0.4062 0.2917
    64
0.2422
```

A bar plot is useful to visually compare the decreasing efficiencies for the configurations with an increasing number of processors. Figure 2 shows the output diagram.

```
R> barplot(efficiency(usl.model))
```

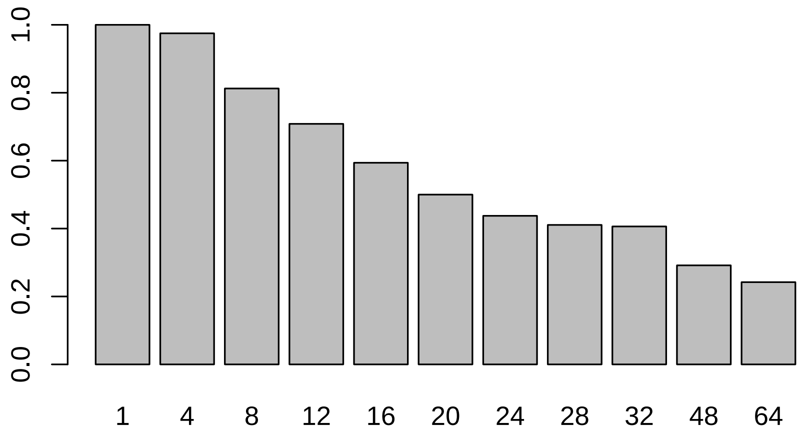

Figure 2: Rate of efficiency for an increasing number of processors running the raytracing software

The values for the model coefficients σ and κ can be retrieved with the `coef()` function.

```
R> coef(usl.model)

    sigma     kappa
5.002e-02 4.708e-06
```

To get an impression of the scalability function we can use the `plot()` function and create a combined graph with the original data as dots and the calculated scalability function as solid line.

```
R> plot(throughput ~ processors, data = raytracer, pch = 16)
R> plot(usl.model, add = TRUE)
```

Figure 3 has the result of that plot.

Imagine that SGI would have built the server with up to 128 processors. We also assume that that system architecture would have no other scalability limitation than the ones already part of the USL model. Then we can use the existing model and predict the system throughput for maybe 96 and 128 and processors.

```
R> predict(usl.model, data.frame(processors = c(96, 128)))

    1     2
331.3 344.6
```

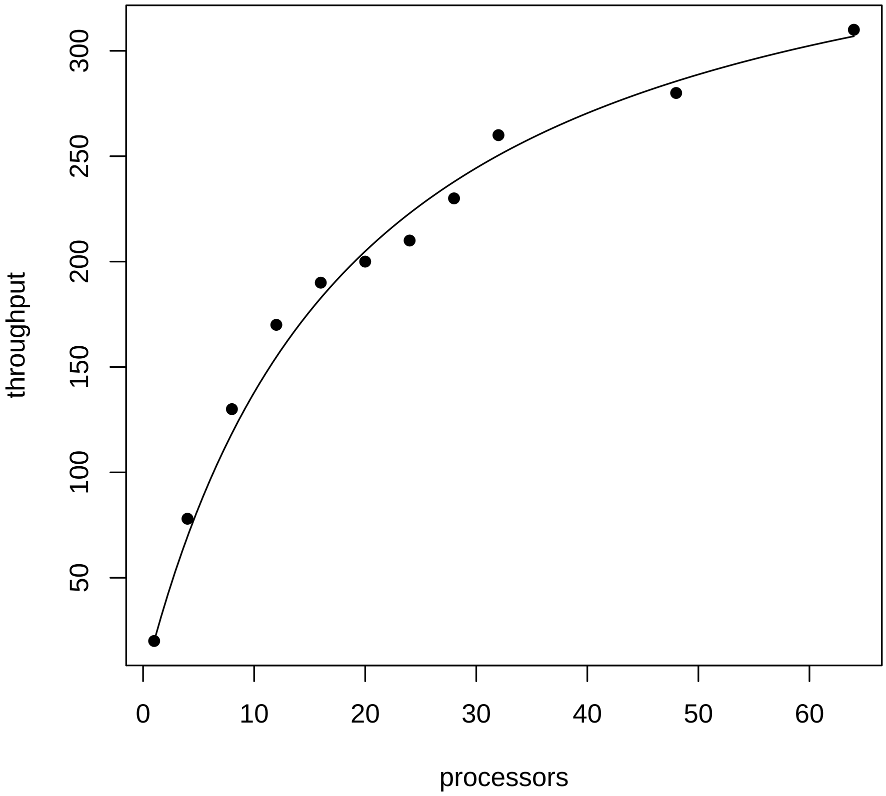

Figure 3: Throughput of a raytracing software using different numbers of processors

We can see from the prediction that there is still an increase in throughput achievable with such a large number of processors. Therefore we now use the `peak.scalability()` function to determine the point where the maximum throughput is reached.

```
R> peak.scalability(usl.model)
```

```
[1] 449.2
```

According to the model the system would achieve its best throughput with 449 processors. This is certainly a result that could not easily be deduced from the original dataset.

4.2 Case Study: Software Scalability

In this section we will perform an analysis of a SPEC benchmark. A Sun SPARCcenter 2000 with 16 CPUs was used in October 1994 for the SDM91 benchmark[2]. The benchmark simulates a number of users working on a UNIX server (editing files, compiling ...) and measures the number of script executions per hour.

First select the demo dataset with the data from the SPEC SDM91 benchmark.

```
R> library(usl)
R> data(specsdm91)
R> specsdm91
```

```
  load throughput
1    1       64.9
2   18      995.9
3   36     1652.4
4   72     1853.2
5  108     1828.9
6  144     1775.0
7  216     1702.2
```

The data provides the measurements made during the benchmark. The column "load" shows the number of virtual users that were simulated by the benchmark and the column "throughput" has the measured number of script executions per hour for that load.

Next we create the USL model for this dataset by calling the `usl()` function. Again we specify a symbolic description of the model and the dataset with the measurements. But this time we choose a different method for the analysis.

```
R> usl.model <- usl(throughput ~ load, specsdm91, method = "nlxb")
```

There are currently three possible values for the `method` parameter:

`default`: The default method uses a transformation into a 2nd degree polynom. It can only be used if the data set contains a value for the normalization where the "throughput" equals 1 for one measurement. This is the algorithm introduced in [Gun07].

`nls`: This method uses then `nls()` function of the R stats package for a nonlinear regression model. It estimates not only the coefficients σ and κ but also the scale factor for the normalization. The nonlinear regression uses constraints for its parameters which means the "port" algorithm is used internally to solve the model. So all restrictions of the "port" algorithm apply.

[2] `http://www.spec.org/osg/sdm91/results/results.html`

`nlxb`: A nonlinear regression model is also used for this case. Instead of the `nls()` function it uses the `nlxb()` function from the nlmrt package (see [Nas12]). This method also estimates both coefficients and the normalization factor. It is expected to be more robust than the `nls` method.

If there is no measurement where "load" equals 1 then the default method does not work and one of the remaining methods must be used.

We also use the `summary()` function to look at the details for the analysis.

```
R> summary(usl.model)

Call:
usl(formula = throughput ~ load, data = specsdm91, method = "nlxb")

Scale Factor for normalization:   90

Efficiency:
   Min     1Q Median     3Q    Max
0.0876 0.1626 0.2860 0.5624 0.7211

Residuals:
   Min     1Q Median     3Q    Max
 -81.7  -48.3  -25.1   29.5  111.1

Coefficients:
   sigma     kappa
0.027728  0.000104

Multiple R-squared: 0.99,        Adjusted R-squared: 0.987
```

Looking at the coefficients we notice that σ is about 2.8 percent and κ is about 0.01 percent. We hypothesize that a proposed change to the system — maybe a redesign of the cache architecture — could reduce κ by half and want to know how the scalability of the system would change.

We can calculate the point of maximum scalability for the current system and for the hypothetical system with the `peak.scalability()` function.

```
R> peak.scalability(usl.model)

[1] 96.52

R> peak.scalability(usl.model, kappa = 5e-05)

[1] 139.4
```

The function accepts two optional arguments `sigma` and `kappa`. They are useful to do a what-if analysis. Setting these parameters override the calculated model parameters and show how the system would behave with a different contention or coherency coefficient.

In this case we learn that the point of peak scalability would move from around 96.5 to about 139 if we would be able to actually build the system with the assumed optimization.

Both calculated scalability functions can be plotted using the `plot()` or `curve()` functions. The following commands create a graph of the original data points and the derived scalability functions. To fully show the scalability of the hypothetical system we have to increase the range of the plotted values with the first command.

```
R> plot(specsdm91, pch = 16, ylim = c(0, 2500))
R> plot(usl.model, add = TRUE)
R> cache.scale <- scalability(usl.model, kappa = 5e-05)
R> curve(cache.scale, lty = 2, add = TRUE)
```

We used the function `scalability()` here. This function is a higher order function that does not return a specific value but another function. This makes it possible to use the `curve()` function to plot the values over the specific range.

Figure 4 shows the measured throughput in scripts per hour for a given load, i.e., the number of simulated users. The solid line indicates the derived USL model while the dashed line resembles our hypothetical system using the optimized cache.

From the figure we can see that the scalability really peaks at one point. Increasing the load beyond that point leads to retrograde behavior, i.e., the throughput decreases again. As we have calculated earlier, the measured system will reach this point sooner than the hypothetical system.

This illustrates how the Universal Scalability Law can help to decide if the system currently is more limited by contention or by coherency issues and also what impact a proposed change would have.

References

[Gun07] Neil J. Gunther. *Guerrilla Capacity Planning: A Tactical Approach to Planning for Highly Scalable Applications and Services.* Springer, Heidelberg, Germany, 1st edition, 2007.

[Nas12] John C. Nash. *nlmrt: Functions for nonlinear least squares solutions*, 2012. R package version 2012-12.16.

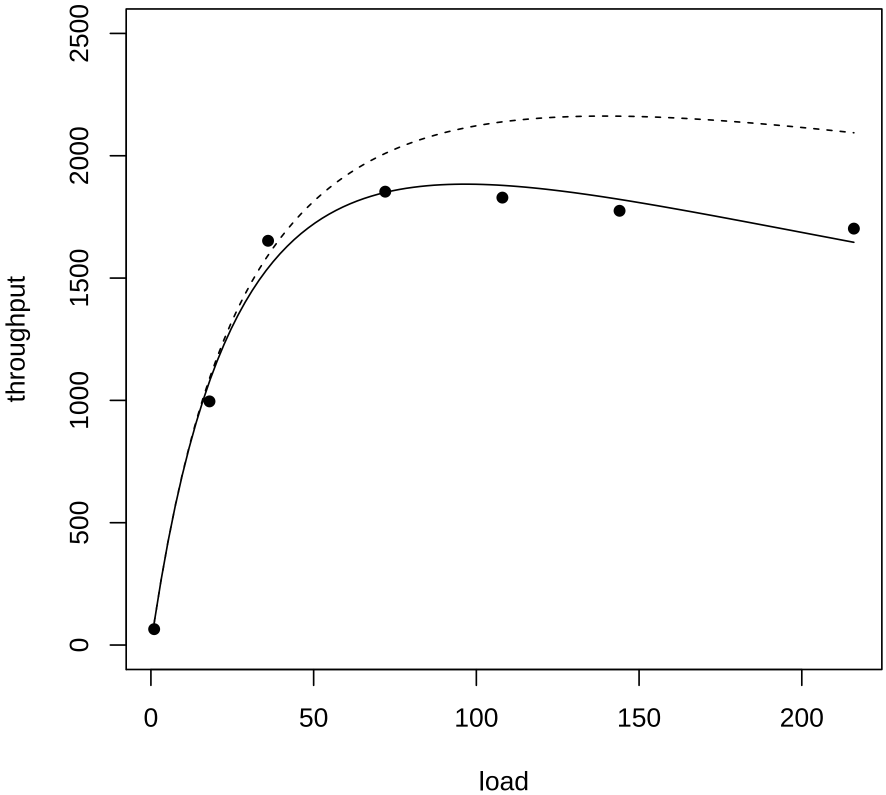

Figure 4: The result of the SPEC SDM91 benchmark for a SPARCcenter 2000 (dots) together with the calculated scalability function (solid line) and a hypothetical scalability function (dashed line)