Modeling calculation kernels with Stan

Hoël Jalmin

Tutored by Arnaud Legrand and Tom Cornebize

The 21th of June, 2019

Context

With the current need for high performance computing, and the hardware complexity:

- How to predict the duration of calculations?
- How to check if the performance is normal?

For this talk:

1. Brief presentation of the context
2. Introduction to Bayesian sampling
3. Examples of application

Background on HPC and Polaris research

Modern context

- HPC systems use thousands of nodes, cache, hyperthreading, etc -> makes it difficult to predict performance
- Some functions are used everywhere, and called thousands of times

Polaris research

- Simulating HPL on smaller supercomputers to optimize it at a lesser cost
- Elaborated complex models but needed to evaluate and confirm the models

HPL simulation

Examples

- The blas library (especially matrix per matrix multiplication) is used by thousands of programs and constitutes most of HPL calculations.
- Performance variability is also caused by network communications
- Needs to check the models with bayesian sampling.

Bayes model

Model Let's say $y \sim \mathcal{N}(\mu, \sigma)$

- μ: Model parameters
- y: Dependent data (posterior)
- σ: Independent data (prior)

We observe some data and need to find model parameters

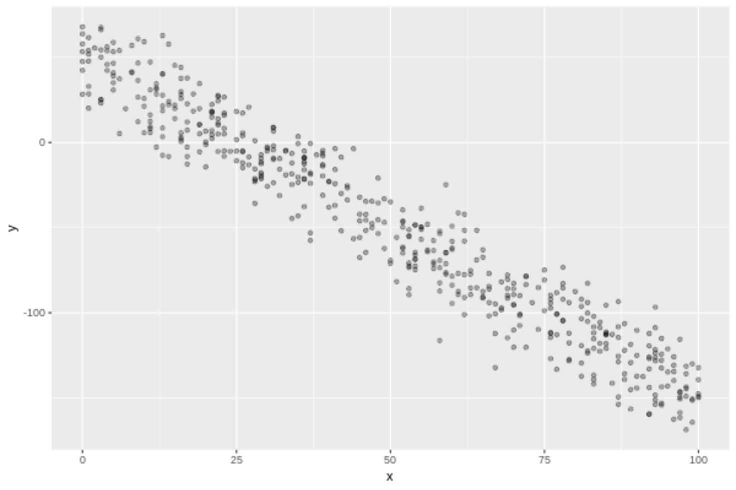
The vocabulary

- Posterior: The distribution of the parameters
- Likelihood: A function of the parameters, the model
- Prior: Existing knowledge of the system, guesses on the parameters values

$$\boxed{\underbrace{p(\mu|y,\sigma)}_{\text{Posterior}} \propto \underbrace{p(y|\mu,\sigma)}_{\text{Likelihood}}\underbrace{p(\mu,\sigma)}_{\text{Prior}}} \text{ assuming } y \sim \mathcal{M}(\mu, \sigma)$$

A Bayesian Sampler, Stan

With a simple example



Using this data, we'll try to find the parameters that were used to generate it.

The Stan model

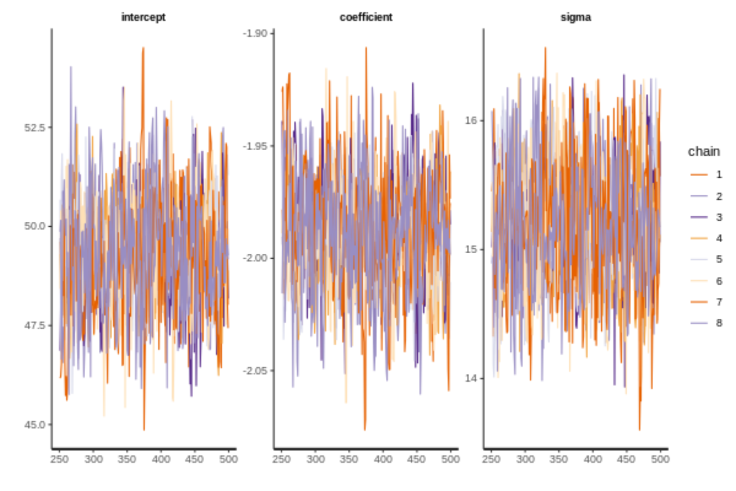
```
library(rstan)

modelString = "data { // the observations
    int<lower=1> N; // number of points
    vector[N] x;
    vector[N] y;
}
parameters { // what we want to find
    real intercept;
    real coefficient;
    real<lower=0> sigma; // indication: sigma cannot be negative
}
model {
    // We define our priors
    intercept   ~ normal(0, 10); // We know that all the parameters follow a nor
    coefficient ~ normal(0, 10);
    sigma       ~ normal(0, 10);

    // Then, our likelihood function
    y ~ normal(coefficient*x + intercept, sigma);
}
"
sm = stan_model(model_code = modelString)
```

Making the fit and checking the results

```
data = list(N=nrow(df),x=df$x,y=df$y)
fit = sampling(sm,data=data, iter=500, chains=8)

print(fit)

Inference for Stan model: ea4b5a288cf5f1d87215860103a9026e.
8 chains, each with iter=500; warmup=250; thin=1;
post-warmup draws per chain=250, total post-warmup draws=2000.

                mean se_mean   sd     2.5%      25%      50%      75%    97.5%
intercept      49.86    0.04 1.36    47.14    48.94    49.87    50.82    52.40
coefficient    -2.00    0.00 0.02    -2.04    -2.01    -2.00    -1.98    -
1.95
sigma          15.03    0.01 0.47    14.18    14.70    15.02    15.35    15.99
lp__        -1615.90    0.04 1.12 -1618.80 -1616.45 -1615.62 -1615.05 -1614.58
            n_eff Rhat
intercept    1070 1.00
coefficient  1042 1.00
sigma        1042 1.01
lp__          871 1.00

Samples were drawn using NUTS(diag_e) at Wed Jun 19 17:07:18 2019.
For each parameter, n_eff is a crude measure of effective sample size,
and Rhat is the potential scale reduction factor on split chains (at
convergence, Rhat=1).
```

Checking the convergence

```
stan_trace(fit)
```



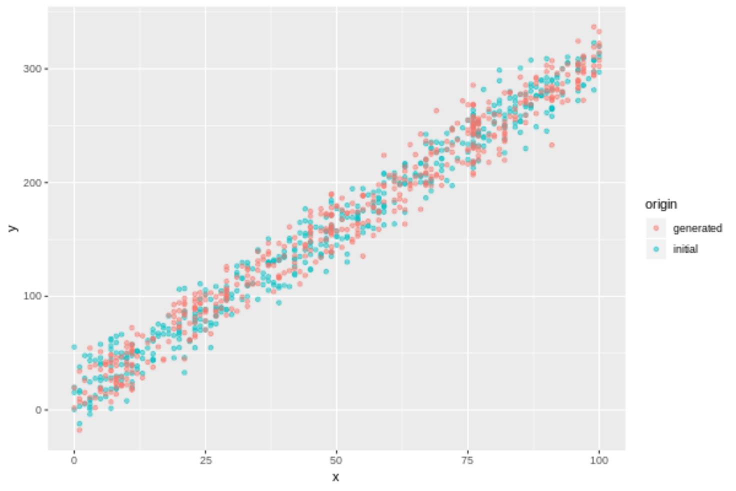
Generating new data

- Generating new data to check the results and model's accuracy.

```
modelString = "data {
    int<lower=1> N;
    vector[N] x;
    vector[N] y;
}
parameters { // what we want to find
    real intercept;
    real coefficient;
    real<lower=0> sigma;
}
model {
    intercept   ~ normal(0, 10);
    coefficient ~ normal(0, 10);
    sigma       ~ normal(0, 10);
    y ~ normal(coefficient*x + intercept, sigma);
}
generated quantities {
   real x_pos = x[categorical_rng(rep_vector(1,N) / N)];
   real y_pos; // posterior predictions
   y_pos = normal_rng(coefficient*x_pos+intercept, sigma);
}
"
sm = stan_model(model_code = modelString)
```

Plotting the new data

```
extracted=rstan::extract(fit)
df_generated = data.frame(x=extracted$x_pos, y=extracted$y_pos, origin='generate
ggplot(tmp, aes(x=x, y=y, color=origin)) + geom_point(alpha=0.5)
```



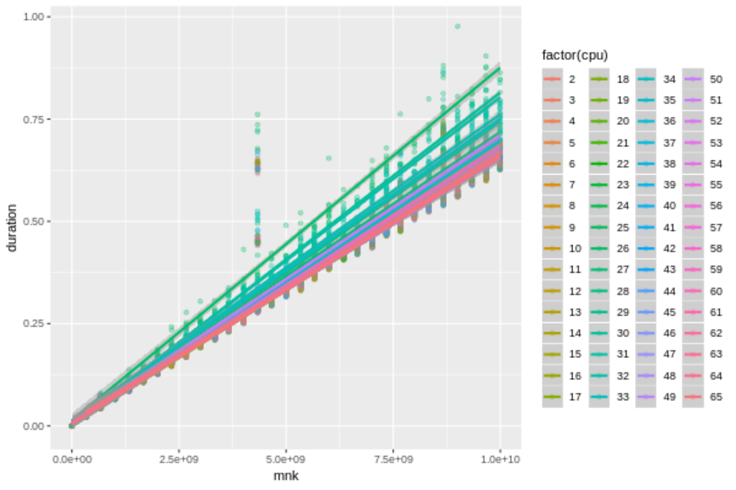
The importance of the priors

- The priors are necessary to have convergence in the fit
- Non-informative prior vs informative (careful not to have a falsely informative one and introduce bias)
- A little bit of precision is better, but initialisation values can make the trick

The different models for dgemm

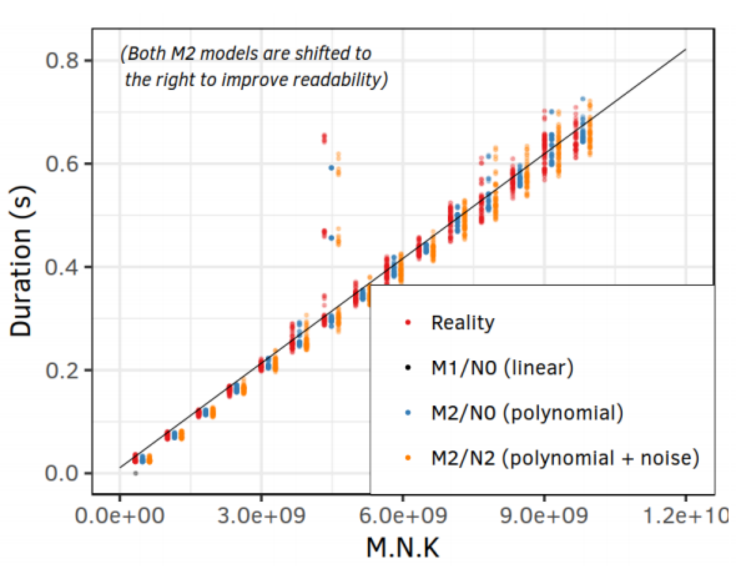
The theory behind it

- Dgemm's duration depends on the matrix size, but also on the CPU used to run it



The possible models

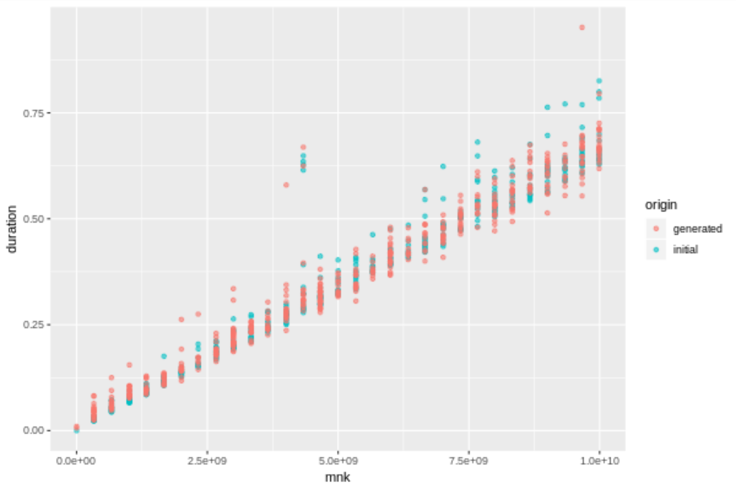
Different possible models, some more accurate than others:



A M-2 N-2 MODEL (POLYNOMIAL WITH NOISE DEPENDING ON X)

```
modelString = "data {
    int<lower=0> N;
    vector[N] mnk;
    vector[N] duration;
    vector[5] mu_mean; //the priors for mu and sigma
    vector[5] mu_sd;
    vector[5] sigma_mean;
    vector[5] sigma_sd;
}
parameters {
    vector[5] mu_raw; vector[5] sigma_raw;
}
transformed parameters {
    vector[5] mu; vector[5] sigma;
    for(j in 1:5){
      mu[j] = mu_mean[j] + mu_raw[j] * mu_sd[j];
      sigma[j] = sigma_mean[j] + sigma_raw[j] * sigma_sd[j];}
}
model {
    mu_raw ~ normal(0,1); sigma_raw ~ normal(0,1);
    duration ~ normal(mu[1]*mnk + mu[2]*mn + mu[3]*mk + mu[4]*nk + mu[5],
    sigma[1]*mnk + sigma[2]*mn + sigma[3]*mk + sigma[4]*nk + sigma[5]);
}
"
```

The generated data



A M_H-2 N_H-2 MODEL (DEPENDING ON THE HOST)

- Much like the previous model, but with different observations for each host
- Added a variable for the number of hosts, and used matrixes instead of vectors for all the parameters.

```
modelString = "data {
    int hosts;
}
parameters {
    matrix[hosts,5] mu_raw;
    matrix[hosts,5] sigma_raw;
}
transformed parameters {
    matrix[hosts,5] mu;
    matrix[hosts,5] sigma;
}
model {
    for(i in 1:hosts){
      duration[i] ~ normal(mu[i,1]*mnk[i]+mu[i,2]*mn[i]+mu[i,3]*mk[i]+
      mu[i,4]*nk[i]+mu[i,5], sigma[i,1]*mnk[i]+sigma[i,2]*mn[i]+
      sigma[i,3]*mk[i]+sigma[i,4]*nk[i]+sigma[i,5]);}
}
"
```

A hierarchical M_H-1 N_H-2 model

- Useful to find the value of hyperparameters from which we get the parameters
- From this we could calculate new parameters for new CPUs
- Here μ-alpha and σ-alpha are the hyperparameters for alpha, and the same goes for the other parameters

$\mu - alpha \sim \mathcal{N}(alpha_{\mu}, alpha_{\sigma})$ with alpha_μ and alpha_σ the priors

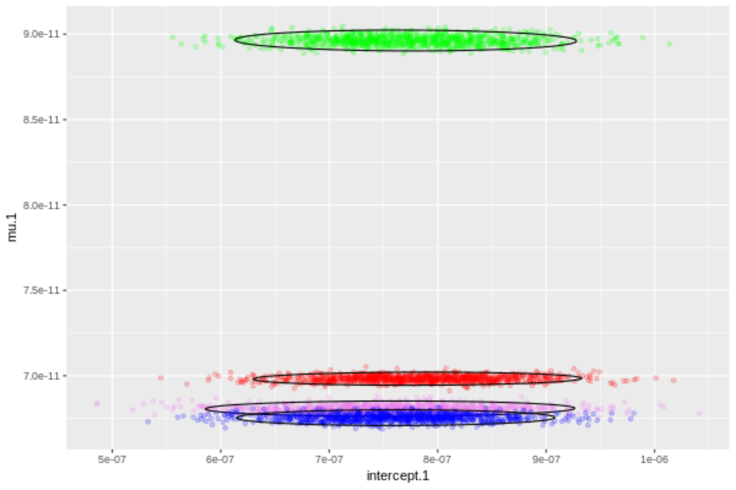
$\sigma - alpha \sim \mathcal{N}(0, 1)$

$alpha[i] \sim \mathcal{N}(\mu - alpha, \sigma - alpha)$

$duration[i] \sim \mathcal{N}(alpha[i] * mnk + beta[i], teta[i] * mnk + gamma[i])$

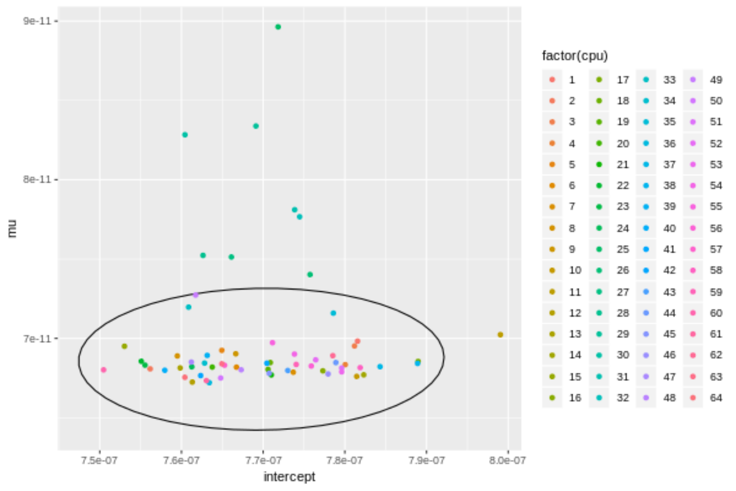
Posterior visualisation

The posterior with models depending on the host shows a lot of difference between hosts (here we have 3 ”average” CPU and a slow one):



Posterior visualisation

If we look at the means of the parameters' values for each host, we get a range of values in which most hosts are.



The follow up

- Modeling other calculation kernels
- Modeling the network communications
- Parsing and converting Stan code to C, to generate new data more efficiently
- Anomaly detection