Efficient implementations of the EM-algorithm for transient Markovian arrival processes*

Mindaugas Bražėnas[1], Gábor Horváth[2], and Miklós Telek[2,3]

[1] Kaunas University of Technology
Department of Applied Mathematics
mindaugas.brazenas@yahoo.com
[2] Budapest University of Technology and Economics
Department of Networked Systems and Services
ghorvath@hit.bme.hu
[3] MTA-BME Information Systems Research Group
telek@hit.bme.hu

Abstract. There are real life applications (e.g., requests of http sessions in web browsing) with a finite number of events and correlated inter-arrival times. Terminating point processes can be used to model such behavior. Transient Markov arrival processes (TMAPs) are computationally appealing terminating point processes which are terminating versions of Markov arrival processes.
In this work we propose algorithms for creating a TMAP based on empirical measurement data and compare various (series/parallel, CPU/GPU) implementations of the EM method for TMAP fitting.

1 Introduction

Stochastic models with background continuous time Markov chain (CTMC) are widely used in stochastic modeling. Phase type (PH) distributions and Markov arrival processes (MAP) exemplify the flexibility and the ease of application of such models. In this work we cope with terminating stochastic processes [1]. Indeed, Phase type distributions are defined by a terminating (also referred to as transient) background Markov chain, but it generates exactly one event. A transient Markovian arrival process (TMAPs) is a point processes with a finite number of possibly correlated inter event times which is governed by a terminating background Markov chain [8]. Basic properties of TMAPs, such as the distribution of the number of generated arrivals or the time until the last arrival, are presented in [8], further properties and moments based characterization are discussed in [6]. TMAPs can be used in a wide range of application fields from traffic modeling of computer systems to risk analysis, including also population dynamics in biological systems. For instance TMAPs are applied to women's lifetime modeling in several countries in [5].

In this work we consider the parameter estimation of TMAPs to experimental data sets based on the EM method. The EM method has been used successfully

* This work was supported by the Hungarian research project OTKA K119750

for parameter estimation of several models with background Markov chains, e.g., for PH distributions [3], for PH distributions with structural restriction [10], for MAPs [4], for MAPs with structural restrictions [9, 7]. The experiences from these previous research results indicate that the inherent redundancy of the stochastic models with background Markov chains makes the parameter estimation of the general models inefficient. In this work we avoid the implementation of the EM based estimation of general TMAPs and immediately apply a similar structural restriction as the one which turned out to be efficient in case of PH distributions [10] and MAPs [9, 7]. The formulas of the EM method for TMAP fitting show similarities with the ones for MAP fitting in [7], but there are introcate details associated with the handling of background process termination which require a non-trivial reconsideration of the expectation and the maximization steps of the method.

Apart of the algorithmic description of the EM method for TMAP fitting we pay attention to efficient implementation for both traditional computing devices (CPU) and graphics processing unit (GPU). Both platforms required various implementation optimizations for efficient computing of the steps of the fitting method. Together with the fitting results and the related computation times we present the applied implementation optimization methods and the related considerations.

The rest of the paper is organized as follows. The next section summarizes the basic properties of TMAPs. Section 3 presents the theoretical foundation of the EM method for TMAP fitting and the high level procedural description of the method. Section 4 discusses several implementation versions for CPU as well as for GPU-based computation. Numerical results are provided in Section 5 and the paper is concluded in Section 6

2 Transient Markovian Arrival Processes

Transient Markovian Arrival Processes (TMAPs) are continuous time terminating point processes where the inter-arrival times depend on a background Markov chain, hence they can be dependent.

TMAPs can be characterized by an initial probability vector, α, holding the initial state distribution of the background Markov chain at time 0 ($\alpha\mathbb{1} = 1$, where $\mathbb{1}$ is the column vector of ones), and two matrices, $\boldsymbol{D_0}$ and $\boldsymbol{D_1}$. Matrix $\boldsymbol{D_0}$ contains the rates of the internal transitions that are not accompanied by an arrival, and matrix $\boldsymbol{D_1}$ consists of the rates of those transitions that generate an arrival. However, contrary to non-terminating MAPs, the generator matrix of the background Markov chain of TMAPs, $\boldsymbol{D} = \boldsymbol{D_0} + \boldsymbol{D_1}$, is transient, that is $\boldsymbol{D}\mathbb{1} \neq 0$ and the non-negative vector $d = -\boldsymbol{D}\mathbb{1}$ describes the termination rates of the background Markov chain. Based on practical considerations we assume that the termination is an observed event (an arrival), which means that a TMAP generates at least one arrival. (If only the "arrival events" are known, which is commonly the case in practice, the TMAPs which do not generate any arrival are not observed. Without knowing how many TMAPs terminated

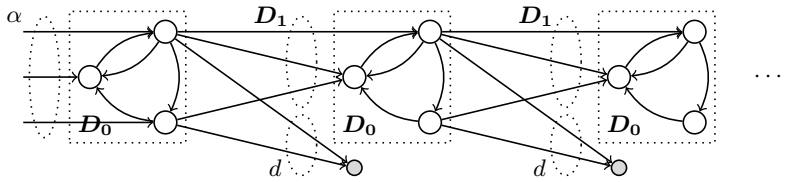

Fig. 1. The structure of the Markov chain representing the number of arrivals and the state of the background process.

without generating any arrival event there is no information to estimate the parameters of those invisible cases.) It also means that the TMAPs considered here are special cases of the ones defined in [8], since we assume that $d_0 = 0$ and our vector d equals to vector d_1 in [8]. The fact that the background Markov chain is transient ensures that the number of events generated by the process is finite. The Markov chain representing the number of arrivals and the state of the background process is depicted in Figure 1.

Matrix $\boldsymbol{P} = (-\boldsymbol{D_0})^{-1}\boldsymbol{D_1}$ describes the state transition probabilities embedded at arrival instants. $\boldsymbol{P}$ holds the state transition probabilities of a transient discrete time Markov chain (DTMC) with termination vector $p = \mathbb{1} - \boldsymbol{P}\mathbb{1}$. Note that $\boldsymbol{P}$ is sub-stochastic matrix (it has non-negative elements and $\boldsymbol{P}\mathbb{1} \leq \mathbb{1}$), and $(\boldsymbol{I} - \boldsymbol{P})^{-1}p = \mathbb{1}$ holds.

In case of TMAPs not only the statistical quantities related to the inter-arrival times are of interest, but also the ones related to the number of generated arrivals.

The number of arrivals $\mathcal{K}$ is characterized by a discrete phase-type (DPH) distribution with initial vector α and transition probability matrix $\boldsymbol{P}$. Hence, the mean number of arrivals is given by

$$E(\mathcal{K}) = \sum_{k=1}^{\infty} \alpha k \boldsymbol{P}^{k-1} p = \alpha(\boldsymbol{I} - \boldsymbol{P})^{-2} p = \alpha(I - \boldsymbol{P})^{-1}\mathbb{1}. \tag{1}$$

If the inter-arrival times are denoted by $\mathcal{X}_1, \mathcal{X}_2, \ldots$, then the joint density function of the inter-arrival times is

$$\begin{aligned} f(x_1, x_2, \ldots, x_k) &= \lim_{\Delta \to 0} \frac{1}{\Delta} P(\mathcal{X}_1 \in (x_1, x_1 + \Delta), \ldots, \mathcal{X}_k \in (x_k, x_k + \Delta)) \\ &= \alpha e^{\boldsymbol{D_0} x_1} \boldsymbol{D_1} e^{\boldsymbol{D_0} x_2} \boldsymbol{D_1} \cdots e^{\boldsymbol{D_0} x_k} (\boldsymbol{D_1}\mathbb{1} + d). \end{aligned} \tag{2}$$

If it exists, the nth moment of $\mathcal{X}_{k+1}$ is

$$E\big(\mathcal{X}_{k+1}^n | \mathcal{X}_{k+1} < \infty\big) = \frac{E\big(\mathcal{X}_{k+1}^n \mathcal{I}_{\{\mathcal{X}_{k+1} < \infty\}}\big)}{P(\mathcal{X}_{k+1} < \infty)} = \frac{n! \alpha \boldsymbol{P}^k (-\boldsymbol{D_0})^{-n} \mathbb{1}}{\alpha \boldsymbol{P}^k \mathbb{1}}. \tag{3}$$

The mean of the inter-arrival times $E(\mathcal{X})$ is not as easy to express as for ordinary MAPs, it is obtained from $E(\mathcal{X}) = E\left(\sum_{k=1}^{\mathcal{K}} \mathcal{X}_k\right) / E(\mathcal{K})$, where the

numerator is derived as

$$E\left(\sum_{k=1}^{\mathcal{K}} \mathcal{X}_k\right) = \sum_{\kappa=1}^{\infty} E\left(\mathcal{I}_{\{\mathcal{K}=\kappa\}} \sum_{k=1}^{\mathcal{K}} \mathcal{X}_k\right) = \sum_{\kappa=1}^{\infty}\sum_{i=0}^{\kappa-1} \alpha \boldsymbol{P}^i \boldsymbol{U} \boldsymbol{P}^{\kappa-1-i} p$$
$$= \sum_{i=0}^{\infty}\sum_{\kappa=0}^{\infty} \alpha \boldsymbol{P}^i \boldsymbol{U} \boldsymbol{P}^{\kappa} p = \alpha(\boldsymbol{I}-\boldsymbol{P})^{-1}\boldsymbol{U}(\boldsymbol{I}-\boldsymbol{P})^{-1}p, \tag{4}$$

where $\boldsymbol{U} = (-\boldsymbol{D_0})^{-1}$, and the denominator is given by (1). As a result the mean inter-arrival time is

$$E(\mathcal{X}) = \frac{E\left(\sum_{k=1}^{\mathcal{K}} \mathcal{X}_k\right)}{E(\mathcal{K})} = \frac{\alpha(\boldsymbol{I}-\boldsymbol{P})^{-1}\boldsymbol{U}(\boldsymbol{I}-\boldsymbol{P})^{-1}p}{\alpha(\boldsymbol{I}-\boldsymbol{P})^{-2}p} = \frac{\alpha(\boldsymbol{I}-\boldsymbol{P})^{-1}\boldsymbol{U}\mathbb{1}}{\alpha(\boldsymbol{I}-\boldsymbol{P})^{-1}\mathbb{1}}. \tag{5}$$

To discuss the correlation of the inter-arrival times we introduce the notation $\hat{\mathcal{X}}_k = \mathcal{X}_k \mid \mathcal{X}_k < \infty$. Note that $\hat{\mathcal{X}}_1 = \mathcal{X}_1$ due to the modeling assumption of at least one arrival. By this notation from (3) we have

$$E\left(\hat{\mathcal{X}}_{k+1}^n\right) = \frac{n!\alpha \boldsymbol{P}^k \boldsymbol{U}^n \mathbb{1}}{\alpha \boldsymbol{P}^k \mathbb{1}}.$$

The expectation of the product of two subsequent inter-arrival times is

$$E\left(\mathcal{X}_1 \hat{\mathcal{X}}_{k+1}\right) = \frac{E\left(\mathcal{X}_1 \mathcal{X}_{k+1} \mathcal{I}_{\{\mathcal{X}_{k+1}<\infty\}}\right)}{P(\mathcal{X}_{k+1} < \infty)}$$
$$= \frac{\alpha(-\boldsymbol{D_0})^{-2}\boldsymbol{D_1}\boldsymbol{P}^{k-1}(-\boldsymbol{D_0})^{-2}(\boldsymbol{D_1}\mathbb{1}+d)}{\alpha(-\boldsymbol{D_0})^{-1}\boldsymbol{D_1}\boldsymbol{P}^{k-1}(-\boldsymbol{D_0})^{-1}(\boldsymbol{D_1}\mathbb{1}+d)} = \frac{\alpha \boldsymbol{U}\boldsymbol{P}^k\boldsymbol{U}\mathbb{1}}{\alpha \boldsymbol{P}^k \mathbb{1}}, \tag{6}$$

where we used that $(-\boldsymbol{D_0})^{-1}\boldsymbol{D_1}\mathbb{1} + d = \mathbb{1}$, due to $\boldsymbol{D_0}\mathbb{1} + \boldsymbol{D_1}\mathbb{1} + d = 0$. Based on the joint expectation the correlation is

$$Corr(\mathcal{X}_1, \hat{\mathcal{X}}_k + 1) = \frac{E\left(\mathcal{X}_1 \hat{\mathcal{X}}_{k+1}\right) - E(\mathcal{X}_1)\, E\left(\hat{\mathcal{X}}_{k+1}\right)}{\sqrt{E(\mathcal{X}_1^2) - E^2(\mathcal{X}_1)}\sqrt{E\left(\hat{\mathcal{X}}_{k+1}^2\right) - E^2\left(\hat{\mathcal{X}}_{k+1}\right)}}. \tag{7}$$

3 An EM Algorithm for TMAPs

In this section an EM algorithm is presented to create a TMAP from measurement data. The measurement data is given by samples $X = (x_k^{(\ell)},\ k = 1, \ldots, K_\ell,\ \ell = 1, \ldots, L)$. We refer the set of dependent samples for a given ℓ as the ℓth run, where the ℓth run is composed by K_ℓ samples. The aim of the EM algorithm is to find $\Theta = (\alpha, \boldsymbol{D_0}, \boldsymbol{D_1})$ by which the likelihood of the observations,

$$\mathcal{L}(\Theta|X) = \prod_{\ell=1}^{L} \alpha e^{\boldsymbol{D_0} x_1^{(\ell)}} \boldsymbol{D_1} \cdots e^{\boldsymbol{D_0} x_{K_\ell}^{(\ell)}} d, \tag{8}$$

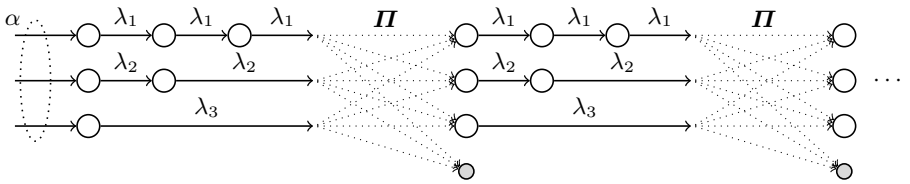

Fig. 2. The special TMAP structure used for fitting.

is maximized. Introducing the run-dependent forward likelihood (row) vectors recursively as

$$a^{(\ell)}[k] = \begin{cases} \alpha, & k = 0, \\ a^{(\ell)}[k-1] e^{\boldsymbol{D_0} x_k^{(\ell)}} \boldsymbol{D_1}, & k > 0, \end{cases} \tag{9}$$

for $\ell = 1, \ldots, L$ and $k = 0, \ldots, K_\ell - 1$, and backward likelihood (column) vectors as

$$b^{(\ell)}[k] = \begin{cases} e^{\boldsymbol{D_0} x_k^{(\ell)}} \boldsymbol{D_1} b^{(\ell)}[k+1], & k < K_\ell, \\ d, & k = K_\ell, \end{cases} \tag{10}$$

for $\ell = 1, \ldots, L$ and $k = K_\ell, \ldots, 1$, the likelihood can be obtained as

$$\mathcal{L}(\Theta|X) = \prod_{\ell=1}^{L} a^{(\ell)}[k_\ell] \cdot b^{(\ell)}[k_\ell + 1], \tag{11}$$

for every $k_\ell = 0, \ldots, K_\ell - 1$.

The forward and backward likelihood vectors play an important role in the presented EM algorithm. However, computing the matrix exponential terms is numerically demanding. To reduce the computational complexity we apply the same structural restriction as in [10], [9] and [7], thus we introduce a special TMAP structure composed of a number of Erlang distributed branches. When a given branch is selected the inter-arrival time is Erlang distributed defined by the parameters (rate and order) of the selected Erlang branch, and after each arrival event a sub-stochastic transition probability matrix determines which Erlang branch to choose for the next inter-arrival given the branch generating the current arrival (see Figure 2). Due to applied structural restriction the computations of matrix exponential terms, e.g. in (8), are replaced by the computations of scalar exponential terms in the form of (12).

In the proposed special structure the inter-arrival times are generated by one of the R Erlang branches. The order and the intensity parameters of the branches are denoted by r_i, λ_i, for $i \in \{1, \ldots, R\}$, respectively. The density of the inter-arrival times generated by branch i is

$$f_i(x) = \frac{(\lambda_i x)^{r_i - 1}}{(r_i - 1)!} \lambda_i e^{-\lambda_i x}. \tag{12}$$

After branch i generates an arrival event, the next one will be generated by branch j with probability $\pi_{i,j}$. The matrix of size $R \times R$ holding these branch switching probabilities is denoted by $\boldsymbol{\Pi} = [\pi_{i,j}]$. Since TMAPs generate a finite number of events we have $\boldsymbol{\Pi}\mathbb{1} < \mathbb{1}$. Observe that the TMAP with the applied structural restriction is uniquely characterized by parameters $\Theta = \{\alpha_i, r_i, \lambda_i, \pi_{i,j},\ \text{for } i,j \in \{1,\ldots,R\}\}$.

By this special TMAP structure the forward and backward likelihood vectors can be obtained without computing matrix exponentials, since

$$a_i^{(\ell)}[k] = \sum_{j=1}^{M} a_j^{(\ell)}[k-1] f_j(x_k^{(\ell)}) \pi_{j,i}, \tag{13}$$

$$b_j^{(\ell)}[k] = \sum_{i=1}^{M} f_j(x_k^{(\ell)}) \pi_{j,i} b_i^{(\ell)}[k+1]. \tag{14}$$

The EM algorithm assumes that the data X available for fitting is incomplete, and there is a hidden data Y. In our case, the hidden data $y_k^{(\ell)} \in Y$ is and integer number representing the Erlang branch that generated the kth inter-arrival time of run ℓ, thus $x_k^{(\ell)}$. If the hidden data was known, the logarithm of the likelihood would be easy to express, as

$$\log \mathcal{L}(\Theta|X,Y) = \sum_{\ell=1}^{L} \sum_{k=1}^{K_\ell} \log\left(f_{y_k^{(\ell)}}(x_k^{(\ell)})\right). \tag{15}$$

Maximizing (15) with respect to λ_i gives

$$\hat{\lambda}_i = \frac{\sum_{\ell=1}^{L} \sum_{k=1}^{K_\ell} r_i \cdot I_{\{y_k^{(\ell)}=i\}}}{\sum_{\ell=1}^{L} \sum_{k=1}^{K_\ell} x_k^{(\ell)} I_{\{y_k^{(\ell)}=i\}}}, \tag{16}$$

where $\hat{\lambda}_i$, and the similar subsequent notations, denotes the optimum assuming Y is known. From [2] we have

$$\hat{\pi}_{i,j} = \frac{\sum_{\ell=1}^{L} \sum_{k=1}^{K_\ell - 1} I_{\{y_k^{(\ell)}=i, y_{k+1}^{(\ell)}=j\}}}{\sum_{\ell=1}^{L} \sum_{k=1}^{K_\ell} I_{\{y_k^{(\ell)}=i\}}}. \tag{17}$$

Note that the summation over k in the denominator runs up to K_ℓ, while the one in the numerator runs up to $K_\ell - 1$, thus matrix $\hat{\boldsymbol{\Pi}}$ is sub-stochastic, reflecting the terminating behavior of TMAPs. The maximum likelihood estimation for the initial vector is

$$\hat{\alpha}_i = \frac{1}{L} \sum_{\ell=1}^{L} I_{\{y_1^{(\ell)}=i\}}. \tag{18}$$

The hidden data is, however, unknown. The marginal distribution of the hidden data $y_k^{(\ell)}$ can be derived from the forward and backward likelihood vectors leading to

$$\begin{aligned} q_i^{(\ell)}[k] = P(y_k^{(\ell)} = i|X,\Theta) &= \frac{P(y_k^{(\ell)} = i, X|\Theta)}{P(X|\Theta)} \\ &= \frac{\left(a_i^{(\ell)}[k-1]\cdot b_i^{(\ell)}[k]\right)\prod_{m\neq\ell} a^{(m)}[0]\cdot b^{(m)}[1]}{\prod_{m=1}^{L} a^{(m)}[0]\cdot b^{(m)}[1]} \\ &= \frac{a_i^{(\ell)}[k-1]\cdot b_i^{(\ell)}[k]}{\alpha\cdot b^{(\ell)}[1]}, \quad k=1,\ldots,K_\ell, \end{aligned} \tag{19}$$

where we used $a^{(\ell)}[0] = \alpha$.

To characterize the joint distribution of the branches generating two consecutive inter-arrival times we also need the probabilities

$$\begin{aligned} q_{i,j}^{(\ell)}[k] = P(y_k^{(\ell)} = i, y_{k+1}^{(\ell)} = j|X,\Theta) &= \frac{P(y_k^{(\ell)} = i, y_{k+1}^{(\ell)} = j, X|\Theta)}{P(X|\Theta)} \\ &= \frac{a_i^{(\ell)}[k-1]\cdot f_i(x_k^{(\ell)})\cdot\pi_{i,j}\cdot b_j^{(\ell)}[k+1]}{\alpha\cdot b^{(\ell)}[1]}. \end{aligned} \tag{20}$$

The calculation of $q_i^{(\ell)}[k]$ and $q_{i,j}^{(\ell)}[k]$ form the E-step of the algorithm.

In the M-step new estimates for Θ are obtained based on the distributions of the hidden data. For λ_i from (16) and (19) we get

$$\lambda_i = \frac{\sum_{\ell=1}^{L}\sum_{k=1}^{K_\ell} r_i\cdot q_i^{(\ell)}[k]}{\sum_{\ell=1}^{L}\sum_{k=1}^{K_\ell} x_k^{(\ell)} q_i^{(\ell)}[k]} = \frac{\sum_{\ell=1}^{L}\frac{\sum_{k=1}^{K_\ell} r_i a_i^{(\ell)}[k-1]b_i^{(\ell)}[k]}{\alpha\cdot b^{(\ell)}[1]}}{\sum_{\ell=1}^{L}\frac{\sum_{k=1}^{K_\ell} x_k^{(\ell)} a_i^{(\ell)}[k-1]b_i^{(\ell)}[k]}{\alpha\cdot b^{(\ell)}[1]}}. \tag{21}$$

Similarly, the new estimates for the branch switching probabilities are obtained from (17) and (20) as

$$\pi_{i,j} = \frac{\sum_{\ell=1}^{L}\sum_{k=1}^{K_\ell-1} q_{i,j}^{(\ell)}[k]}{\sum_{\ell=1}^{L}\sum_{k=1}^{K_\ell} q_i^{(\ell)}[k]} = \frac{\sum_{\ell=1}^{L}\frac{\sum_{k=1}^{K_\ell-1} a_i^{(\ell)}[k-1]f_i(x_k^{(\ell)})\pi_{i,j}b_j^{(\ell)}[k+1]}{\alpha\cdot b^{(\ell)}[1]}}{\sum_{\ell=1}^{L}\frac{\sum_{k=1}^{K_\ell} a_i^{(\ell)}[k-1]b_i^{(\ell)}[k]}{\alpha\cdot b^{(\ell)}[1]}}. \tag{22}$$

Finally, probabilities α_i are derived from (18) and (19), yielding

$$\alpha_i = \frac{1}{L}\sum_{\ell=1}^{L} q_i^{(\ell)}[1] = \frac{1}{L}\sum_{\ell=1}^{L}\frac{\alpha_i b_i^{(\ell)}[1]}{\alpha\cdot b^{(\ell)}[1]}. \tag{23}$$

4 Details of the Numerical Algorithm

The EM algorithm presented in Section 3 is not straight forward to implement in an efficient way. While the special structure proposed for fitting does reduce the

Algorithm 1 Pseudo-code of the proposed EM algorithm

procedure EM-Fit($x_k^{(\ell)}, \lambda_i, \pi_{i,j}, \alpha_i, r_i$)
  $LogLi \leftarrow -\infty$
  for $iter = 1$ **to** $maxIter$ **do**
    Compute and store conditional densities $f_i(x_k^{(\ell)})$ by (12)
    for $\ell = 1$ **to** L **do**
      for $k = 1$ **to** K_ℓ **do**
        Compute and store forward likelihood vectors $a^{(\ell)}[k]$ by (13)
      end for
      Compute and store backward likelihood vector $b^{(\ell)}[K_\ell]$ by (10)
      for $k = K_\ell - 1$ **down to** l **do**
        Compute and store backward likelihood vector $b^{(\ell)}[k]$ by (14)
      end for
    end for
    $oLogLi \leftarrow LogLi$
    $LogLi \leftarrow \sum_{\ell=1}^{L} \alpha \cdot b^{(\ell)}[1]$
    if $iter > 1$ **and** $(LogLi - LogLi) < ln\,(1 + \epsilon)$ **then**
      return $(\lambda_i, \pi_{i,j}, \alpha_i)$
    end if
    Compute new estimates for λ_i by (21)
    Compute new estimate for $\pi_{i,j}$ by (22)
    Compute new estimate for α_i by (23)
  end for
  return $(\lambda_i, \pi_{i,j}, \alpha_i)$
end procedure

computational demand of the procedure significantly, the naive implementation (shown in Figure 1) still contains many numerical pitfalls.

Our aim is to develop an implementation that enables the practical application of the algorithm, thus

- the execution time must be reasonable with large data sets (containing millions of samples),
- the implementation must be insensitive to the order of magnitude of the input data,
- the implementation should exploit the parallel processing capabilities of modern hardware.

These items are addressed in the subsections below.

4.1 Initial guess for $\boldsymbol{\alpha}$, $\boldsymbol{\lambda_i}$ and $\boldsymbol{\Pi}$

We use the following randomly generated initial parameters. α is a random probability vector (composed of R uniform pseudo-random numbers in $(0, 1)$ divided by the sum of the R numbers). The mean run length of the data set is computed as $\bar{K} = \sum_{\ell=1}^{L} K_\ell / L$, and based on that each row of matrix $\boldsymbol{\Pi}$ is a random probability vector multiplied by $1 - 1/\bar{K}$ (that is, initially the

exit probability is the same, $1/\bar{K}$, in each Erlang branch). The initial values for λ_i are computed based on the mean inter-arrival time $\bar{T} = \frac{\sum_{\ell=1}^{L}\sum_{k=1}^{K_\ell} x_k^{(\ell)}}{\sum_{\ell=1}^{L} K_\ell}$, and it is $\lambda_i = r_i/\bar{T}$. Let $x_{max} = \max_{\ell,k} x_k^{(\ell)}$ and $\lambda_{max} = \max_i \lambda_i$. In order to avoid underflow during the computation of $e^{-x_k^{(\ell)}\lambda_i}$ in (12) we re-scale this initial guess according to the representation limits of the single precision floating point numbers with $8+16$ bits, where one of the 8 bits of the mantissa indicates the sign. That is $2^{2^7} \sim e^{88}$ is the representation limit. Accordingly, if $x_{max}\lambda_{max} > 60$ then we re-scale the initial intensity values to $\lambda_i = \frac{60\lambda_i}{x_{max}\lambda_{max}}$, where 60 is a heuristic choice to be far enough from the representation limit (which is 88).

4.2 Improving numerical stability of the forward and backward likelihood vectors computation

Computing vectors $a^{(\ell)}[k]$ and $b^{(\ell)}[k]$ by applying recursions (13) and (14) directly can lead to numerical overflow. To overcome this difficulty we express these vectors in the normal form

$$\begin{aligned} a_i^{(\ell)}[k] &= \dot{a}_i^{(\ell)}[k] \cdot 2^{\ddot{a}^{(\ell)}[k]}, \\ b_i^{(\ell)}[k] &= \dot{b}_i^{(\ell)}[k] \cdot 2^{\ddot{b}^{(\ell)}[k]}, \end{aligned} \tag{24}$$

where $\ddot{a}^{(\ell)}[k]$ and $\ddot{b}^{(\ell)}[k]$ are integer numbers and the values $\dot{a}_i^{(\ell)}[k]$, $\dot{b}_i^{(\ell)}[k]$ are such that $0.5 \leq \dot{a}^{(\ell)}[k]\mathbb{1} < 1$ and $0.5 \leq \mathbb{1}^T\dot{b}^{(\ell)}[k] < 1$. For a given vector $a_i^{(\ell)}[k]$, $\dot{a}_i^{(\ell)}[k]$ and $\ddot{a}^{(\ell)}[k]$ can be obtained from

$$\dot{a}_i^{(\ell)}[k] = \frac{a_i^{(\ell)}[k]}{2^{\left\lceil \log_2\left(a^{(\ell)}[k]\mathbb{1}\right)\right\rceil}}, \quad \ddot{a}^{(\ell)}[k] = \left\lceil \log_2(a^{(\ell)}[k]\mathbb{1}) \right\rceil. \tag{25}$$

To avoid the calculation of $a_i^{(\ell)}[k]$ (that can under- or overflow), it is possible to modify the recursion (13) to work with $\dot{a}_i^{(\ell)}[k]$ and $\ddot{a}^{(\ell)}[k]$ directly, leading to

$$\begin{aligned} \tilde{a}_i^{(\ell)}[k] &= \sum_{j=0}^{M} \dot{a}_i^{(\ell)}[k-1] f_j\left(x_k^{(\ell)}\right)\pi_{j,i}, \\ \dot{a}_i^{(\ell)}[k] &= \frac{\tilde{a}_i^{(\ell)}[k]}{2^{\left\lceil \log_2(\tilde{a}^{(\ell)}[k]\mathbb{1})\right\rceil}}, \quad \ddot{a}^{(\ell)}[k] = \ddot{a}^{(\ell)}[k-1] + \left\lceil \log_2(\tilde{a}^{(\ell)}[k]\mathbb{1}) \right\rceil. \end{aligned} \tag{26}$$

Hence, in the first step $\tilde{a}_i^{(\ell)}[k]$ is computed, from which in the second step the normalized quantity is derived and the exponent is incremented by the appropriate magnitude. To obtain the normal form of $\dot{a}_i^{(\ell)}[0]$ and $\ddot{a}^{(\ell)}[0]$, we can apply (25). The treatment of the normal form of the backward likelihood vectors $\dot{b}_i^{(\ell)}[k]$ follow the same pattern.

The parameter estimation formulas using the normal form of the forward and backward likelihood vectors are

$$\lambda_i = \frac{\sum_{\ell=1}^{L} \frac{1}{\alpha\dot{b}^{(\ell)}[1]} \sum_{k=1}^{K_\ell} r_i \dot{a}_i^{(\ell)}[k-1]\dot{b}_i^{(\ell)}[k] 2^{\ddot{a}^{(\ell)}[k-1]+\ddot{b}^{(\ell)}[k]-\ddot{b}^{(\ell)}[1]}}{\sum_{\ell=1}^{L} \frac{1}{\alpha\dot{b}^{(\ell)}[1]} \sum_{k=1}^{K_\ell} x_k^{(\ell)} \dot{a}_i^{(\ell)}[k-1]\dot{b}_i^{(\ell)}[k] 2^{\ddot{a}^{(\ell)}[k-1]+\ddot{b}^{(\ell)}[k]-\ddot{b}^{(\ell)}[1]}}, \tag{27}$$

$$\pi_{i,j} = \frac{\sum_{\ell=1}^{L} \frac{1}{\alpha\dot{b}^{(\ell)}[1]} \sum_{k=1}^{K_\ell} \dot{a}_i^{(\ell)}[k-1] f_i\left(x_k^{(\ell)}\right) \pi_{i,j} \dot{b}_i^{(\ell)}[k+1] 2^{\ddot{a}^{(\ell)}[k-1]+\ddot{b}^{(\ell)}[k+1]-\ddot{b}^{(\ell)}[1]}}{\sum_{\ell=1}^{L} \frac{1}{\alpha\dot{b}^{(\ell)}[1]} \sum_{k=1}^{K_\ell} \dot{a}_i^{(\ell)}[k-1]\dot{b}_i^{(\ell)}[k] 2^{\ddot{a}^{(\ell)}[k-1]+\ddot{b}^{(\ell)}[k]-\ddot{b}^{(\ell)}[1]}}, \tag{28}$$

$$\alpha_i = \frac{1}{L} \sum_{\ell=1}^{L} \frac{\alpha_i \dot{b}_i^{(\ell)}[1]}{\alpha\dot{b}^{(\ell)}[1]}. \tag{29}$$

Observe that the exponent of 2 depends only on the difference of $\ddot{a}^{(\ell)}[k]$ and $\ddot{b}^{(\ell)}[k]$ for consecutive k values according to (26), thus the multiplication and the division with large numbers has been avoided.

Finally, the log-likelihood of the whole trace data can be computed as

$$\mathcal{L}(\Theta|X) = \log\left(\prod_{\ell=1}^{L} \alpha b^{(\ell)}[1]\right) = \sum_{\ell=1}^{L} \log\left(\alpha\dot{b}^{(\ell)}[1]\right) + \ddot{b}^{(\ell)}[1]\log(2). \tag{30}$$

4.3 Serial implementations

For accuracy and performance comparison we have implemented three versions of the algorithm shown in Figure 1 (with the discussed modifications for numerical stability) :

- Java implementation using double precision floating point numbers,
- C++ implementation using double precision floating point numbers,
- C++ implementation using single precision floating point numbers.

4.4 Parallel implementation

We have adapted the presented algorithm to be executed on GPUs (graphics processing units) by using CUDA library. GPUs are cheap in the sense of computing power, however, their computing cores are much simpler compared to the ones of CPU. Therefore to fully utilize the hardware low level technical details have to be considered such as the thread grouping, the multi-level memory hierarchy, reducing the number of conditional jumps, memory operations, etc.

The entry part of the algorithm (shown in Figure 2) is executed on the host environment (i.e. processed by CPU) from which the so called kernels (shown in Figures 3, 4) are invoked to be executed on GPU device. Upon kernel launching the number of threads in block, the number of blocks in grid and amount of shared memory (in bytes) to be allocated for every block has to be specified.

After kernel launch host process waits until all the threads are processed by the kernel, and then resumes.

The KERNEL-A, shown in Figure 3, computes the normalized likelihood vectors $\dot{a}^{(\ell)}[k]$, $\dot{b}^{(\ell)}[k]$ and their respective exponents $\ddot{a}^{(\ell)}[k]$, $\ddot{b}^{(\ell)}[k]$. The number of threads in block and grid size can be chosen freely, so that to utilize the specific capabilities of the GPU. However, the threads should be assigned with similar amount of work in order not to waste computing resources.

The KERNEL-B, shown in Figure 4, computes new parameter estimates $\Theta = (\lambda_i, \pi_{i,j}, \alpha_i)$. Synchronization between threads is necessary before computing the actual parameter estimate after numerator and denominator values are computed. Since thread synchronization is possible only within a block, the number of blocks is determined by the number of $\pi_{i,j}$ estimates, thus R^2. Thread count in block can by chosen freely.

Note that work load for the kernels are different. The data runs are allocated to threads in grid for KERNEL-A. While for KERNEL-B all the runs are allocated among threads for every block.

Even run allocation to threads is a complex problem. A simple greedy solution is to assign runs in descending order (of number of inter-arrival time samples) to the thread, which has been assigned with the smallest number of inter-arrivals.

Global GPU memory accessing operations are slower compared to shared memory. It is a common practice to load frequently used data from global memory into shared one and after calculations write results back into global memory. In our case parameter estimates as well as structure parameters are uploaded in shared memory.

Additionally previously computed likelihood vector values are cached for computing the next ones. Also Erlang branch densities are computed and stored in shared memory just before to be used in subsequent calculations.

Shared memory can be used for communication, since it is visible for all the threads within block. In KERNEL-B threads perform summation across the assigned runs and the intermediate results are written to shared memory to be loaded by one designated thread to compute the final estimate value.

5 Numerical Experiments

We start the section with a general note on the applied special structure. In spite of the natural expectation that the result of the fitting (in terms of likelihood) with the special structure is worse than the one with the general TMAP class of the same size, however, similar to related results in the literature [9, 7] our numerical experience is just the opposite. The general TMAP class is redundant [6], and the EM algorithm goes back and forth between different representations of almost equivalent TMAPs. Our special TMAP class has much less parameters, and the benefit of optimizing according to less parameters dominates the drawback coming from reduced flexibility of the special structure.

Hereafter we compare the behavior of the four implementations (three serial ones (Java(double), C++(double), C++(single)) and the one for GPU) of the

Algorithm 2 Pseudo-code of the proposed EM algorithm (CUDA)

procedure EM-FIT-CUDA($x_k^{(\ell)}, \lambda_i, \pi_{i,j}, \alpha_i, r_i$, {run allocation to threads})
  Allocate device memory.
  Copy data from host to device memory.
  $LogLi \leftarrow -\infty$
  for $iter = 1$ **to** $maxIter$ **do**
    Invoke KERNEL-A for computing likelihood vectors and run likelihoods.
    $oLogLi \leftarrow LogLi$
    Copy run likelihoods from device to host memory.
    Compute trace data log-likelihood $LogLi$.
    if $iter > 1$ **and** $(LogLi - LogLi) < ln\,(1 + \epsilon)$ **then**
      Copy parameter estimates from device to host memory.
      return $(\lambda_i, \pi_{i,j}, \alpha_i)$
    end if
    Invoke KERNEL-B for computing new parameter estimates.
  end for
  Copy parameter estimates from device to host memory.
  Deallocate device memory.
  return $(\lambda_i, \pi_{i,j}, \alpha_i)$
end procedure

presented EM algorithm. All numerical experiments were made on an average PC with an Intel Core 2 CPU clocked at 2112 MHz with 32KB L1 cache and 4096KB L2 cache, and an ASUS GeForce GTX 560 Ti graphics card with a GPU clocked at 900 MHz having 1 GB of RAM and 384 CUDA cores. For the GPU implementation the first kernel is launched with 64 blocks of 32 threads each, the second one is launched with 9 (R^2) blocks of 192 threads each.

Two data sets are considered, in the first one there are 1000000 runs and 8824586 inter-arrival times in total, and in the second one there are 2000000 runs and 14503248 inter-arrival times in total.

Based on the experiences in [7] we adopt three Erlang branches ($R = 3$) with 1, 2 and 3 states ($r_1 = 1, r_2 = 2, r_3 = 3$). For fair comparison we have run 30 iterations of the EM algorithm in all cases (after which the algorithm seemed to converge) and the compared results are always initiated with the same initial guesses. The run times and log-likelihoods are compared in Table 1.

	With 1 million samples		With 2 million samples	
Implementation	Log-likelihood	Execution time	Log-likelihood	Execution time
Java (double)	$-3.65998 \cdot 10^7$	14m 22s	$-5.73999 \cdot 10^7$	22m 15s
C++ (double)	$-3.65998 \cdot 10^7$	06m 21s	$-5.73999 \cdot 10^7$	10m 28s
C++ (single)	$-3.65902 \cdot 10^7$	06m 44s	$-5.73752 \cdot 10^7$	11m 04s
CUDA (single)	$-3.65997 \cdot 10^7$	46s	$-5.73997 \cdot 10^7$	01m 13s

Table 1. Execution times and log likelihoods of different implementations

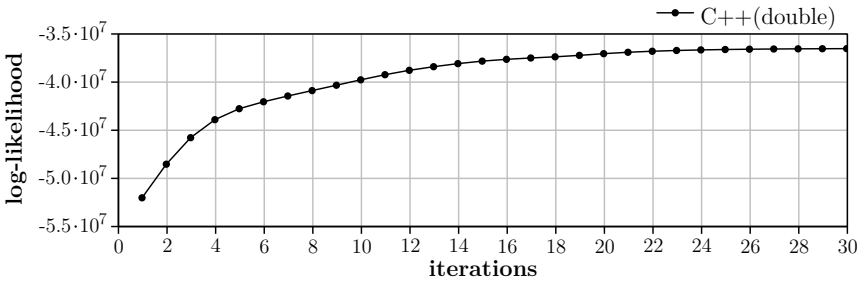

Fig. 3. Log-likelihood of 1000000 sample trace data obtained by C++(double) procedure.

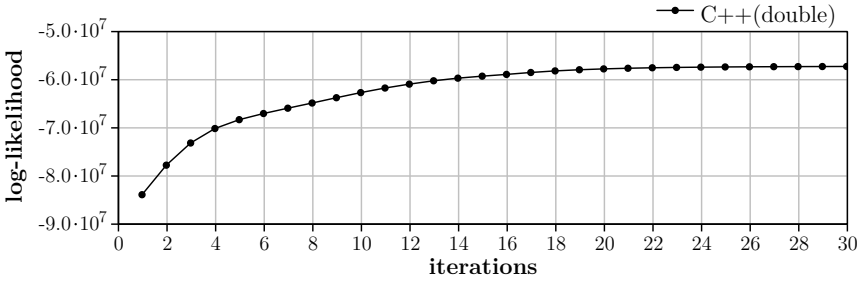

Fig. 4. Log-likelihood of 2000000 sample trace data obtained by C++(double) procedure.

Time necessary to allocate/deallocate arrays is not included in run time, because of different C++ and Java memory management policies. However the time for data allocation/deallocation on GPU device is included. The trace data sample distribution among threads is done in advance, thus not included in run time.

After every iteration log-likelihood was computed using double precision floating point variables. Java (double) and C++(double) implementations gave identical log-likelihoods and are shown in Figures 3, 4. Log-likelihoods obtained by running C++(single) are relatively similar to ones acquired using C++ (double) implementation. Therefore, it is more convenient to plot the difference of log-likelihood obtained from C++(single) minus C++(double). The same applies for results obtained by CUDA (single) implementation.

6 Conclusions

An EM procedure to estimate special structure TMAP parameters was developed and four of its implementations were tested by fitting reasonable large data sets. The C++ implementations of the fitting procedure indicated that both the single and the double precision floating points versions are stable, and converged

to similar limits. Due to the fact that the log-likelihood vectors of independent runs can be computed independently parallel implementation on GPU can speed up the procedure significantly. Log-likelihoods obtained by using CUDA implementation are close to ones obtained computing with serial implementation on CPU.

References

1. *Finite Point Processes*, pages 111–156. Springer New York, New York, NY, 2003.
2. Theodore W. Anderson and Leo A. Goodman. Statistical inference about Markov chains. *The Annals of Mathematical Statistics*, pages 89–110, 1957.
3. S. Asmussen, O. Nerman, and M. Olsson. Fitting phase-type distributions via the EM algorithm. *Scandinavian Journal of Statistics*, 23:419–441, 1996.
4. Peter Buchholz. An EM-algorithm for MAP fitting from real traffic data. In *Computer Performance Evaluation. Modelling Techniques and Tools*, pages 218–236. Springer, 2003.
5. S. Hautphenne and G. Latouche. The Markovian binary tree applied to demography. *Journal of Mathematical Biology*, 64:1109–1135, 2012.
6. Sophie Hautphenne and Miklós Telek. Extension of some MAP results to transient MAPs and Markovian binary trees. *Performance Evaluation*, 70(9):607–622, 2013.
7. Gábor Horváth and Hiroyuki Okamura. A fast EM algorithm for fitting marked Markovian arrival processes with a new special structure. In *Computer Performance Engineering*, pages 119–133. Springer, 2013.
8. G. Latouche, M.-A. Remiche, and P. Taylor. Transient Markov arrival processes. *Annals of Applied Probability*, 13:628–640, 2003.
9. Hiroyuki Okamura and Tadashi Dohi. Faster maximum likelihood estimation algorithms for Markovian arrival processes. In *Sixth International Conference on the Quantitative Evaluation of Systems, QEST'09*, pages 73–82. IEEE, 2009.
10. Axel Thümmler, Peter Buchholz, and Miklos Telek. A novel approach for phase-type fitting with the EM algorithm. *IEEE Transactions on Dependable and Secure Computing*, 3(3):245–258, 2006.

Algorithm 3 Pseudo-code of the KERNEL-A

procedure KERNEL-A($x_k^{(\ell)}, \lambda_i, \pi_{i,j}, \alpha_i, r_i$, {run allocation to threads})
  Identify index ti of thread within the block.
  Identify index si of thread within all the threads in grid.
  Reference shared memory.
  if $ti = 0$ **then**
    Copy estimates for λ_i, $\pi_{i,j}$ and α_i from global memory to shared memory.
    Compute absorbtion probabilities $\pi_i = 1 - \sum_{j=1}^{R} \pi_{i,j}$ and store them in shared memory.
  end if
  Synchronize threads within block.
  for every ℓ **assigned to** si **thread do**
    for $j = 1$ **to** R **do**
      Compute densities $f_j\left(x_1^{(\ell)}\right)$, $f_j\left(x_{K_\ell}^{(\ell)}\right)$ by (12) and store them in shared memory.
    end for
    for $i = 1$ **to** R **do**
      Compute likelihoods $\tilde{a}_i^{(\ell)}[1]$, $\tilde{b}_i^{(\ell)}[K_\ell]$ according to (26) and store them in shared memory.
    end for
    Compute exponents $\ddot{a}^{(\ell)}[1]$, $\ddot{b}^{(\ell)}[K_\ell]$ according to (26) and write them to global memory (also keep values in registers for later use).
    for $i = 1$ **to** R **do**
      Compute normalized likelihoods $\dot{a}_i^{(\ell)}[1]$, $\dot{b}_i^{(\ell)}[K_\ell]$ according to (26), cache them in shared memory and write to global memory.
    end for
    for $k = 2$ **to** $K_\ell - 1$ **do**
      for $j = 1$ **to** R **do**
        Compute densities $f_j\left(x_k^{(\ell)}\right)$, $f_j\left(x_{K_\ell - k}^{(\ell)}\right)$ and store them in shared memory.
      end for
      for $i = 1$ **to** R **do**
        Compute likelihoods $\tilde{a}_i^{(\ell)}[k]$, $\tilde{b}_i^{(\ell)}[K_\ell - k]$ according to (26) and store them in shared memory.
      end for
      Compute exponents $\ddot{a}^{(\ell)}[k]$, $\ddot{b}^{(\ell)}[K_\ell - k]$ according to (26) and write them to global memory (also keep values in registers for later use).
      for $i = 1$ **to** R **do**
        Compute normalized likelihoods $\dot{a}_i^{(\ell)}[k]$, $\dot{b}_i^{(\ell)}[K_\ell - k]$ according to (26), cache them in shared memory and write to global memory.
      end for
    end for
    Compute ℓth run likelihood and write to global memory.
    Compute ℓth run log-likelihood and sum up to register.
  end for
  Write sum of sample log-likelihoods into shared memory.
  Synchronize threads within block.
  Sum up all the run log-likelihoods within block and write to global memory.
end procedure

Algorithm 4 Pseudo-code of KERNEL-B

procedure KERNEL-B($x_k^{(\ell)}, \lambda_i, \pi_{i,j}, \alpha_i, r_i$, {run allocation to threads})
    Identify index (i, j) of the block within grid.
    Identify index si of thread within the current block.
    Reference shared memory.
    for every ℓ **assigned to** si **thread do**
        Read ℓth run's likelihood value from global memory and store in register.
        Compute part of (28) denominator for (ℓ) and sum up in shared memory.
        if $K_\ell > 1$ **then**
            Compute part of (28) numerator for (ℓ) and sum up in shared memory.
        end if
    end for
    Synchronize threads within block.
    if $si = 1$ **then**
        Read summed up values for denominator and numerator, compute new estimate $\pi_{i,j}$ by (28) and store in global memory.
    end if
    Synchronize threads within block.
    if $j = 1$ **then**
        for every ℓ **assigned to** si **thread do**
            Compute part of (27) numerator and denominator for (ℓ) and sum up in shared memory.
        end for
    end if
    Synchronize threads within block.
    if $si = 1$ **then**
        Read summed up values for denominator and numerator, compute new estimate λ_i by (27) and store in global memory.
    end if
    Synchronize threads within block.
    if $j = 1$ **then**
        for every ℓ **assigned to** si **thread do**
            Compute part of (29) numerator for (ℓ) and sum up in shared memory.
        end for
    end if
    Synchronize threads within block.
    if $si = 1$ **then**
        Read summed up values for numerator, compute new estimate α_i by (29) and store in global memory.
    end if
end procedure