First EuroNGI Workshop:
New Trends in Modelling, Quantitative Methods and Measurements

A distributed approach to solve large Markov chains

JAROSŁAW BYLINA[a]

[a]Department of Computer Science, Marie Curie-Skłodowska University, Pl. M. Curie-Skłodowskiej 1, 20-031 Lublin, Poland, e-mail: jmbylina@hektor.umcs.lublin.pl

1. Introduction

Queueing networks have been widely used for modelling and analysis of real computer networks and many other kinds of networks (as comunication networks, for example).

In general it is necessary to adopt a strictly numerical approach to queueing networks. It is always possible to obtain a Markov chain for any queueing network – we can approximate arbitrarily closely any probability distribution with a phase-type representation [5] – like Cox or Erlang distributions. Additionally, we can include features such as priority queueing, blocking etc. in the Markov chain representation.

Although, in general, when a compilcated network behaviour is to be represented by a Markov chain then the size of the state space becomes huge very fast and there are problems with space and time complexity of algorithms for solving such huge linear systems.

A distributed algorithm for solving huge and sparse linear systems that appear in the Markov chain analysis of queueing network models was presented in [2] (and in this paper, in sections 3.–4.). That algorithm requires the linear system coefficients matrix $\mathbf{Q}$ to be distributed (nearly) evenly among machines before the start of the computations.

One solution to that problem is to generate the matrix $\mathbf{Q}$ on one computer and then distribute it among others. However, this approach is rather time-consuming – because we use only one machine of the whole cluster and the distribution after the matrix generation is expensive from the communicational point of view – and space-consuming – because we try to store the whole huge matrix on one machine what can be expensive or even impossible (one of the advantages of the algorithm described in [2] was the starting distribution of the matrix $\mathbf{Q}$ – to use the space on the machines in the best manner).

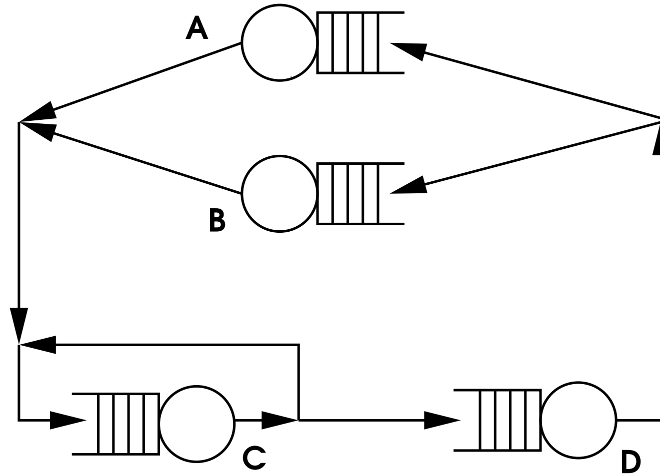

Fig. 1. A scheme of a queueing network example

We achieve better results when we design a distributed algorithm for generating the matrix **Q** after which there will be a respective part of the matrix on each of the machines. In our algorithm each computer is responsible for generating its own part of the matrix. Of course, there is some communication but reduced to the minimum. Such an algorithm is presented in [3] (and in this paper, in sections 5.–6.).

2. Queueing Networks and Markov Chains

To start investigating the behaviour of a queueing network as a Markov chain we have to choose a representation for the set of *states*. The most common manner of this is to represent each system state as a vector which components completely describes states of all elements of a queueing network. For the queueing network from the figure 1 with the constant number N of customers and with exponential (that is 'without memory') distribution of the service times we can describe the system state with a four-element vector (a, b, c, d) where a, b, c and d are numbers of customers waiting and being served in service stations **A**, **B**, **C** and **D**, respectively, and where $a + b + c + d = N$. But – for example – for elements with a Cox or Erlang distribution of service time we need two numbers – one for the number of customers and second for the state (phase) of the server. Further, for example, we may require more paremeters if there may be more classes of customers in the queue and so on.

Next we have to enumerate all potential transitions among states and define for them the *transition rates* q_{ij} (independent of time for homogeneous Markov chain, most in-

teresting for us):

$$q_{ij} = lim_{\Delta t \to 0} \frac{p_{ij}(\Delta t)}{\Delta t} \qquad \text{for} \quad i \neq j,$$

$$q_{ii} = -\sum_{j \neq i} q_{ij},$$

where $p_{ij}(\Delta t)$ is the probability that if the system is in the state i the transition occurs to the state j in the time Δt. This is how the transition rate matrix $\mathbf{Q} = (q_{ij})$ is created.

These steps are realised by the algorithm from [3].

The last step (being realised by the algorithm from [2]) in the analysis is to compute probability vector $\boldsymbol{\pi}(t)$ which components $\pi_i(t)$ are the probabilities that the system is in the state i at the time moment t – or to compute the long-run (independent of time) probability vector $\boldsymbol{\pi} = (\pi_1, \ldots, \pi_n)$ from:

$$\boldsymbol{\pi}\mathbf{Q} = \mathbf{0}^T, \qquad \boldsymbol{\pi} \geq \mathbf{0}^T, \qquad \sum_i \pi_i = 1.$$

We are interested in the latter one. For conviniency we assign $\mathbf{x} = \boldsymbol{\pi}^T$ and our problem is to find $\mathbf{x} = (x_1, \ldots, x_n)$ from:

$$\mathbf{Q}^T\mathbf{x} = \mathbf{0}, \qquad \mathbf{x} \geq \mathbf{0}, \qquad \sum_i x_i = 1.$$

3. The Iterative GMRES Algorithm

For solving such an equation we chose one of projection methods, namely *iterative GMRES algorithm* – see the figure 2. One of the advantages of this method is no fill-in generation (because the matrix $\mathbf{Q}$ is only used in the matrix-vector multiplication), the other is the fast convergency rate. The iterative GMRES algorithm is also convinient to vectorize [4] and parallelize (see the next section and [2]).

If we want to use iterative GMRES algorithm to solve Markov chains representing behaviour of a queueing network we should expect the matrix (and of course vectors) of a huge size. To store such a huge and sparse matrix in an efficient manner we use three one-dimensional arrays (the Harwell-Boeing storage scheme). First of them (the only array with real items) stores only the nonzero elements of the matrix. These elements are stored by rows of $\mathbf{Q}^T$ but the elements do not need to be in any particular order within a row – it is only necessarily for all elements of the row i to be before all elements of the row $(i+1)$. The second array contains – for each nonzero element – its column index. The third array holds only one number for each column of the matrix $\mathbf{Q}^T$: its starting index in the first and second array.

1. choose $\mathbf{x}_0$ and m
2. $\mathbf{r}_0 \leftarrow -\mathbf{Q}^T\mathbf{x}_0$
3. $\beta \leftarrow ||\mathbf{r}_0||_2$
4. $\mathbf{v}_1 \leftarrow \mathbf{r}_0/\beta$
5. for $j \leftarrow 1, \ldots, m$:
   (a) $\mathbf{w} \leftarrow \mathbf{Q}^T\mathbf{v}_j$
   (b) for $i \leftarrow 1, \ldots, j$:
      i. $h_{ij} \leftarrow \mathbf{v}_i^T\mathbf{w}$
      ii. $\mathbf{w} \leftarrow \mathbf{w} - h_{ij}\mathbf{v}_i$
   (c) $h_{j+1,j} \leftarrow ||\mathbf{w}||_2$
   (d) $\mathbf{v}_{j+1} \leftarrow \mathbf{w}/h_{j+1,j}$
6. find $\mathbf{y} = (y_1, \ldots, y_m)$ minimizing $||\beta\mathbf{e}_1 - \mathbf{H}\mathbf{y}||_2$
7. $\mathbf{x}_0 \leftarrow \mathbf{x}_0 + \sum_{i=1}^{m} \mathbf{v}_i y_i$
8. if the result is insufficiently accurate then go to 2

Fig. 2. The iterative GMRES algorithm

4. The Distributed Implementation

In our distributed implementation of iterative GMRES we used MPI (*Message-Passing Interface*, [1, 6]) standard that allows writing distributed programs relatively easy. Program was writen in the language C and compiled with the gcc compiler under Linux.

To achieve the best size performance we decide to divide the matrix $\mathbf{Q}^T$ (n/k consecutive rows for each node, where n is the size of the matrix and k is the number of nodes) and the vectors $\mathbf{r}_0$, $\mathbf{x}_0$, $\mathbf{w}$ and $\mathbf{v}_i$, $i = 1, \ldots, m + 1$ (n/k consecutive items of each vector for each node) evenly among nodes.

All operations on the vectors (as scaling, adding, multiplying, computing their norm) are computed localy by each node on that node's part of the vector(s). Then the partial result is exchanged with others nodes (only if it is necessary – as for computing the norm of a vector or the scalar product of two vectors).

To multiply the matrix $\mathbf{Q}^T$ by a vector we have to gather all components of the vector at the node (so we need one additional auxiliary full-sized vector in each node) *before* multiplying but we do not need to exchange the elements of the product, because each node hold 'its own' part of the vector after multiplying.

To see how much memory each node needs for the algorithm, let us assume that nz is the number of nonzero elements of the matrix $\mathbf{Q}^T$. Each of the machines stores:

- for the matrix $\mathbf{Q}$:
  - nz/k real numbers;
  - $(nz + n)/k$ integer numbers;
- for each of the vectors $\mathbf{r}_0$, $\mathbf{x}_0$, $\mathbf{v}_i$, $\mathbf{w}$:
  - n/k real numbers;
- for the auxiliary vector:
  - n real numbers.

So each node has to store $(\frac{nz+(3+m)\cdot n}{k} + n)$ real numbers and $\frac{nz+n}{k}$ integer numbers.

5. The Generation of the Transition Rate Matrix

However, to exploit well the distributed GMRES algorithm (or any other distributed algorithm in which the matrix $\mathbf{Q}$ is used only for the matrix-vector multiplication) we should efficiently distribute the matrix $\mathbf{Q}$ before computations. The best way to do that is not to distribute it at all but to generate every part of the matrix on its machine.

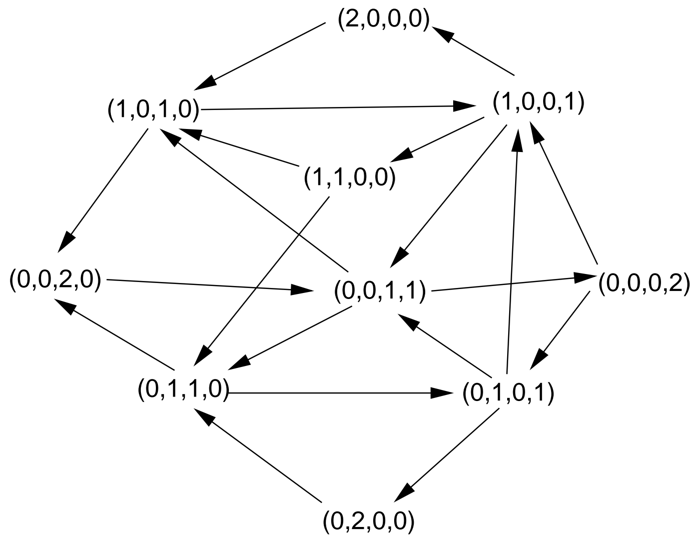

Fig. 3. A transition graph of a queueing network

We are intrested in one of the most general methods – the Breadth-First Search algorithm (BFS), which allows us to enumerate all states, to number them, to enumerate all potential transition among states and to compute the transition rates.

The BFS algorithm is an algorithm to traverse all the edges of a directed graph. We start with a list with a single (arbitrary) vertex (that is: a state vector) and we investigate all edges with the given vertex as a starting point. For each of these edges we add to the list its ending point – but only if it is not in the list yet. Next we go to the next vertex in the list and so on, while there are vertices not yet traversed in the list.

Described above algorithm can be used to generate the transition rate matrix for a queueing model. In this case it does not search the graph but it generates the transition graph (such a graph for the queueing network from the figure 1 for $N = 2$ is presented on the figure 3) and the transition rates. Such a version of the BFS algorithm is presented on the figure 4.

6. The Distributed Generation

To adapt the algorithm presented above to work on a network of computers we have to pick one of the machines as a *master*. The others will be *slaves*. The slaves are the machines which generate their own parts of the matrix $\mathbf{Q}$ (and they will participate in computing the probability vector $\boldsymbol{\pi}$ with the use of the algorithm from sections 3.–4. [2, 3]; the master will not participate in the second part). The master controls the actions

1. Initialize `L` as a numbered list that contains only one, arbitrary state vector `w` (a graph vertex).
2. For each vector `v` representing the state that is allowed as a next state after the state `w`:
   (a) if `v` is not a member of the list `L` then attach `v` to `L`;
   (b) compute the transitin rate from `w` to `v` and remember it as `Q[ind(w, L), ind(v, L)]` (where `ind(a, B)` is an index of `a` in the list `B`).
3. If `w` is not the last element of `L` then assign the next element of `L` to `w` and go to 2.

Fig. 4. The BFS generation algrithm

of the slaves, receives some data from them, maintains the global data and sends the slaves some information. The idea of the algorithm is as follows.

First, the master sends a state vector to each slave, but different vectors to different slaves. Moreover, each slave have to be sent a vector belonging to a different *vector pool* (see below). Next, each slave generates the transition rates as described in section 5. but only for the transitions from the states of the slave's own vector pool. If the slave meets a state not from its vector pool, then the state is attached to an auxiliary list, which is send to the master after the whole process. The master tracks the entire generated state space and while there exists state not being generated by the slave owning the state's vector pool the master sends it to that slave and orders it to start over with this state as a starting state. The slave algorithm is presented on the figure 5.

The vector pool is a key idea for this algorithm. The vector pools are subsets of the state space, their intersections are empty and their union is the whole state space. Moreover, the pools should be of almost the same count beacause they are assigned to slaves 'one-to-one' and the algorithm described in [2] works the best for this condition. The problem is that we have to know the division into the pools *before* our distributed algorithm starts.

The method to solve this problem is to describe each of the vector pools with simple conditions, which can be distributed among slaves before the actual algorithm starts. Exemplary conditions for our graph from the figure 3 could be 'the first element of the vector is zero'/'the first element of the vector is not zero' (one pool of 6 and the other of 4 elements) or 'the 'one of the vector elements is 2/'no vector element is 2 and the

1. Initialize `L` as a numbered list that contains only one state vector `w` (a graph vertex, received from the master).
2. Initialize `X` as an empty list.
3. For each vector `v` representing the state that is allowed as a next state after the state `w`:
   (a) if `v` is not a member of your vector pool and is not a member of the list `X` then attach `v` to `X` else if `v` is a member of your vector pool and is not a member of the list `L` then attach `v` to `L`;
   (b) compute the transitin rate from `w` to `v` and remember it as `QL[ind(w, L), ind(v, L)]` or as `QX[ind(w, L), ind(v, X)]` – it depends on the list in which the `v` is (where `ind(a, B)` is an index of `a` in the list `B`).
4. If `w` is not the last element of `L` then assign the next element of `L` to `w` and go to 3.
5. Send the elements of `X` (that have not been sent yet) to the master.
6. Wait for the answer from the master.
7. If the answer will not be the 'ending signal', then attach received elements (state vectors generated by other slaves and received by the master) to `L` and go to 4.
8. Together with the ending signal the master sends the data (mainly sizes and indices) that the slave uses to compose its own part of the matrix `Q` from `QL` and `QX`.

Fig. 5. The distributed BFS generation algrithm – slave

first one is zero'/'no vector element is 2 and the first one is not zero' (4/3/3). The slaves can easily check if the generated state vectors are in their pools with such conditions, but on the other hand we should choose these conditions very carefully – to minimize differencies between quantities of the vector pools.

7. Conclusion

We presented two algorithms which together can be used to solve large Markov chains (namely to generate the Markov chain's infinitesimal generator $\mathbf{Q}$ and to solve the equation $\boldsymbol{\pi}\mathbf{Q} = \mathbf{0}^T$) exploiting a potential of distributed systems. The algorithms are partially implemented and tested. There is an open (and not simple) question how the vector pools can be automatically chosen by the system when only the queueing model (and not the matrix $\mathbf{Q}$) is known.

References

[1] B. Bylina: Komunikacja w MPI, *Informatyka Stosowana S2/2001, V Lubelskie Akademickie Forum Informatyczne*, Kazimierz Dolny, 17–18 maja 2001, pp. 31–40 (in Polish).

[2] J. Bylina: Distributed solving of Markov chains for computer network models, *Annales UMCS Informatica* 1 (2003), Lublin 2003, pp. 15–20.

[3] J. Bylina: Distributed generation of Markov chains infinitesimal generator matrices for modelling networks (submitted to *Annales UMCS Informatica*).

[4] J. Bylina, B. Bylina: GMRES dla rozwiązywania łańcuchów Markowa na komputerze wektorowym CRAY SV1, *Algorytmy, metody i programy naukowe*, Polskie Towarzystwo Informatyczne, Lublin 2004, pp. 19–24 (in Polish).

[5] M.F. Neuts: *Matrix Geometric Solutions in Stochastic Models – An Algorithmic Approach*, Johns Hopkins University Press, Baltimore, 1981.

[6] P.S. Pacheco: *A User's Guide to MPI*, University of San Francisco 1998.