



Department of Mathematics
Faculty of Mechanical Engineering
Slovak University of Technology in Bratislava

4th International Conference
APLIMAT 2005

DISTRIBUTED GENERATION OF MARKOV CHAINS INFINITESIMAL GENERATORS WITH THE USE OF THE LOW LEVEL NETWORK INTERFACE

BYLINA Jarosław, (PL), BYLINA Beata, (PL)

Abstract. In this paper a distributed algorithm for the Markov chain transition rate matrix (a.k.a. infinitesimal generator) generation is presented. A division of the Markov chain state set among the machines with the use of conditional expressions is considered. An implementation (with the use of the BSD socket layer interface) of this algorithm is described. Problems connected with such an implementation are concerned.

1 Introduction

Modelling with the use of queueing networks is a tool wide used for analysis of real systems. A very useful method for investigating the queueing models is to analyse Markov chains corresponding to the queueing models.

Obviously, there are some problems connected to such an approach. First of all, Markov chains are stochastic processes with no memory of past states (i.e. the whole future of the process depends only on the present state), and if they are to describe real systems the memory must be included in the description of each state – and that is why a great number of states (with included past information) arises from each state (without such information included). The more precision we demand, the more states we need – it is so called ‘states explosion’.

On the other hand, one of the advantages of the homogeneous Markov chains is relative ease of finding the long-run probabilities π_i of the states ($i = 1, \dots, n$, where n is the number of states). Namely, if we enumerate all the states of a Markov chain and calculate all the transition rates between them we can find mentioned above probabilities by solving the equation:

$$\pi \mathbf{Q} = 0, \quad \pi \geq 0, \quad \|\pi\|_1 = 1, \quad (1)$$

where $\pi = (\pi_i)_{1 \leq i \leq n}$, and $\mathbf{Q} = (q_{ij})_{1 \leq i, j \leq n}$ is the transition rates matrix given by:

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

$$q_{ii}(t) = - \sum_{j \neq i} q_{ij}(t)$$

where $p_{ij}(t_1, t_2)$ is the probability that when the model is in the state i in the moment t_1 the transition occurs to the state j before the moment t_2 .

Various method for solving equation (1) are presented in [1, 5, 8] among others. In this article we want to consider the problem of the matrix \mathbf{Q} generation.

2 The traditional algorithm

We are interested in distributed generation of the matrix \mathbf{Q} . Such an approach is convenient because we want to utilize a distributed implementation of solving the equation (1) and if we were generating the matrix \mathbf{Q} on only one machine and then distributing it, the time to distribute (and perhaps also the time to generate) would be very long. Moreover, the memory required to store the whole matrix only on one machine (during the generation) could be simply insufficient. Thus, better is to generate the matrix \mathbf{Q} in parts, on the machines on which we are to conduct next steps of our computations instead of distributing it afterwards – one part of \mathbf{Q} on each machine.

The matrix \mathbf{Q} generation is equivalent to the generation of a transition graph (with weights) for the considered Markov chain. For the basis of our implementation we chose a very general method – the Breadth-First Search (BFS) algorithm [7]. The use of this algorithm allows us to traverse (or generate) all the edges of the transition graph (assumed that the graph is connected – which is true in our case) and generate their weights. Each edge of the transition graph corresponds to exactly one positive element of the matrix \mathbf{Q} which is the transition rate between given states (or the edge's weight).

The traditional (not distributed) BSF algorithm can be used for the generation as following:

1. start with a list L containing an arbitrary vertex w of the graph (i.e.: an arbitrary state of the Markov chain);
2. for each vertex v adjacent to w :
 - (a) if v is not in L then attach v to L ,
 - (b) calculate (and store in \mathbf{Q}) the transition rate from w to v ;
3. if w is not the last vertex in L then take the next vertex from L as w and go to 2.

3 The distributed algorithm

We have been working on a distributed implementation of the algorithm described above. In [2] and [3] we presented an early version of the distributed algorithm but some issues were concerned and improved during the course of the implementation.

The distributed algorithm is based on the *master-slave* idea – one of the processes is chosen as a *master* supervising other processes (*slaves*) which do the actual work. Moreover, the master does not need a lot of processor time nor communication, so it can be run on a machine on which a slave runs. Every machine hosts only one slave and after generation each of them takes part in the second portion of computations – in distributed solving the equation (1) [2, 4].

The slave part of our new distributed algorithm is presented below (we do not present the master part, because the master's work can be obviously seen from the slave part). Some terms (POOLS, to ANSWER, to ASK) are defined later.

1. Receive POOLS' description from the master.
2. Initialize L_i (for $i = 1, \dots, p$, where p is the total of the pools) as an empty lists and \mathbf{Q}_i (for $i = 1, \dots, p$) as empty matrices.
3. Choose an arbitrary vertex (i.e.: state) from your own pool and attach it to L_m (where m is the number of your pool).
4. Let w be the first unprocessed vertex from L_m .
5. For every v adjacent to w :
 - (a) ANSWER;
 - (b) if v is not in $L_{\text{pool}(v)}$ (where $\text{pool}(v)$ is the number of the pool of the vertex v) then attach v to $L_{\text{pool}(v)}$;
 - (c) calculate (and store in $\mathbf{Q}_{\text{pool}(v)}$ ASKING if it is necessary) the transition rate from w to v .
6. ANSWER.
7. If all vertices in your L_m have been processed:
 - (a) let the master know about that;
 - (b) if the master's reply is that the generation is complete – jump to 9.
8. Return to 4.
9. Compose your part of the matrix \mathbf{Q} needed for distributed solving (1) from \mathbf{Q}_i (where $i = 1, \dots, p$).

The POOLS (in the above algorithm) are simply sets of graph vertices and every vertex is in exactly one pool. Each slave owns its own (exactly one) pool.

ANSWERING means that the slave receives data (if there are any at the moment) about its own pool's vertices from other slaves (while they ASK) that generated them, and then answers these slaves by sending them these vertices' indices in matrices \mathbf{Q}_i . The answering slave also attaches received vertices to L_m (if they are not there, of course).

ASKING means sending a generated vertex to another slave (to the owner of the vertex) and waiting for its index. While waiting, the slave ANSWERS other slaves' ASKINGS, because if it slept it could cause a dead-lock.

4 The pools

The pools should be at similar size because thus we achieve the better performance in the second portion of computations – in the distributed solving of the equation (1) [2, 4].

Moreover, the less edges between vertices from distinct pools, the less need to ASK so the less communication needed and – by this – the faster generation. The figure 1 shows two

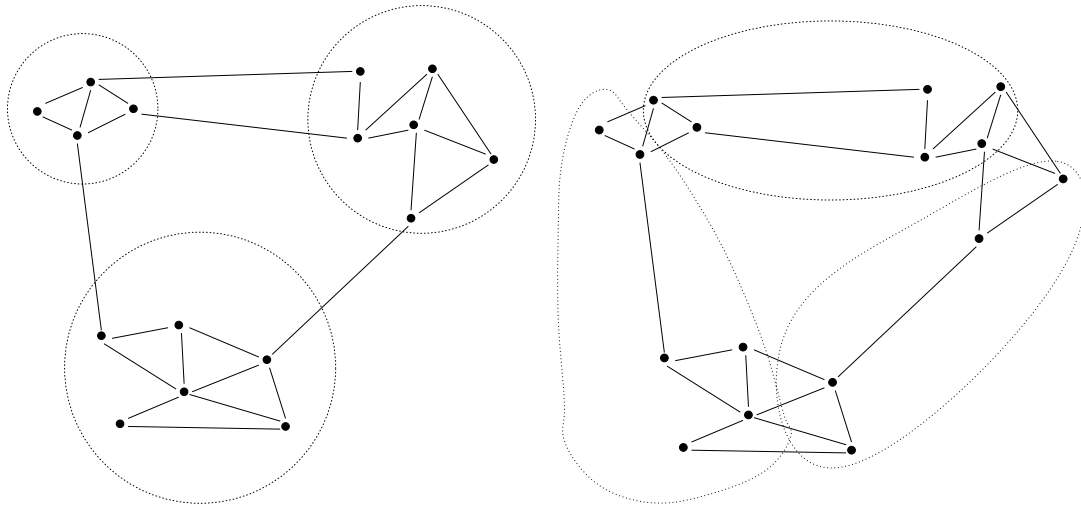


Figure 1: A graph (arrows and weights on the edges omitted for clarity) with two different divisions into pools

different divisions of a graph into three pools. Numbers of vertices in the pools are the same (6/6/4) but in the first (left) division we have much less edges between pools than in the second (right) one and thus the first one is better by generating less communication.

The pools are to be defined before starting the generation. Of course, before the generation we have no vertices generated, so no pool can be defined by enumerating vertices belonging to it. Rather, we describe pools with simple conditions, by which the master and the slaves can easily check which pool a generated/received vertex belongs to.

In our implementation vertices are represented by vectors (because the vertices of the transition graph *are* vectors describing states of the Markov chain). These vectors have a fixed length and integer elements. This is why the pool belonging conditions for vertices (vectors) can be expressed for example as ‘if the first element of a given state vector is zero then the state vector belongs to the pool #1’ or ‘if the first element of a given state vector is not zero and the second element is an even number then the state vector belongs to the pool #2’ etc.

Such conditions can be easily coded and distributed among the slaves by the master before the actual algorithm starts (in its first step, see above). But on the other hand we should choose these conditions very carefully – to minimize differences between quantities of the vector pools. The selection of the conditions should be also carefully attuned to the actual model. We are working on some automation of the choice of the conditions.

5 Details of the implementation

The algorithm was implemented as a set of files written with the use of the language ANSI C [6] and compiled with the compiler `gcc` (but none of the nonstandard `gcc`’s features were utilized) under the operating system *Linux*. All the communication is implemented with the use of the *BSD socket layer* interface what makes the implementation portable to many Unixes.

Moreover, such a low-level approach gives us an opportunity, to efficiently use the network by sending deliberately prepared packages, not sending unnecessary information, and sending it as fast as possible. In distributed algorithms communication is an important issue and that is why we decided for such a solution.

To represent vertices of the transition graph (i.e.: states of the Markov chain) we chose

a vector representation. Each vertex is stored as a vector, components of which describe the components of the analysed system. However, details of the model – to achieve the maximal flexibility – are left to be established by the user of our implementation. It should be done in the file `model.c` by (re)defining functions:

```
struct list_rates * lr_init_adjacent(Tvertex);  
Tvertex vertex_to_start_with(int);
```

First of them generates a list of vertex directly adjacent to a given one, together with the transition rates to them; the second one generates a starting (arbitrary) vertex belonging to a given pool.

The data structures worth mentioning are the lists L_i (for $i = 1, \dots, p$) which are not pure lists, because they need to include a fast searching mechanism (see steps 5a, 5b, 6 in our algorithm above). We decided to use a crossover of one-direction lists and binary search trees.

6 Conclusion and future works

Our implementation is still in the phase of corrections. We are concentrated on achieving the best time performance. The next step is to connect the distributed generation with distributed solving algorithms (GMRES [4, 2], but not only).

The side effect of the use of the low-level socket interface is ability to utilize our generation algorithm implementation with the big sets of machines – connected with not necessarily a very fast (nor homogeneous) network. Thus we are interested in investigating the behaviour of our implementation on clusters and grids.

The last – but not least – question we want to mention is the choice of the pools in advance (before the generation), knowing only some general facts about the analysed Markov chain. It is connected to ease of the transition graph decomposition and we are working hard on it.

References

- [1] Bylina, B., Bylina, J.: The Vectorized and Parallelized Solving of Markovian Models for Optical Networks, In: *Lecture Notes in Computer Science* 3037, Springer-Verlag Berlin Heidelberg 2004, pp. 578–581.
- [2] Bylina, J.: A distributed approach to solve large Markov chains, In: *First EuroNGI Workshop: New Trends in Modelling, Quantitative Methods and Measurements* (to appear).
- [3] Bylina, J.: Distributed generation of Markov chains infinitesimal generator matrices for queueing network models, In: *Annales UMCS Informatica* (to appear).
- [4] Bylina, J.: Distributed solving of Markov chains for computer network models, In: *Annales UMCS Informatica* 1 (2003), Lublin 2003, pp. 15–20.
- [5] Bylina, J., Bylina, B: Solving Markov chains with the WZ factorization for modelling networks, In: *Proceedings of 3rd International Conference Aplimat 2004*, Bratislava 2004, pp. 307–312.
- [6] Kernighan, B.W., Ritchie, D.M.: *The C Programming Language, Second Edition*, Prentice Hall Inc., 1988.
- [7] Knottenbelt, W.: *Parallel Performance Analysis of Large Large Markov Models*, PhD. thesis, University of London, 1999.
- [8] Stewart, W.: *Introduction to the Numerical Solution of Markov Chains*, Princeton University Press, Chichester, West Sussex, 1994.

Current address

Mgr. Beata Bylina, Department of Computer Science, Institute of Mathematics, Maria Curie-Skłodowska University, Pl. M. Curie-Skłodowskiej 1, 20-031 Lublin, Poland

e-mail: beatas@hektor.umcs.lublin.pl

Mgr. Jarosław Bylina, Department of Computer Science, Institute of Mathematics, Maria Curie-Skłodowska University, Pl. M. Curie-Skłodowskiej 1, 20-031 Lublin, Poland

e-mail: jmbylina@hektor.umcs.lublin.pl