Aggregation and Numerical Techniques for Passage Time Calculations in Large semi-Markov Models

Marcel Christoph Günther
mcg05@doc.ic.ac.uk
June 18, 2009

Marker: Dr. Jeremy Bradley
Second marker: Dr. William Knottenbelt

Department of Computing
Imperial College London

Abstract

First-passage time densities and quantiles are important metrics in performance analysis. They are used in the analysis of mobile communication systems, web servers, manufacturing systems as well as for the analysis of the quality of service of hospitals and government organisations. In this report we look at computational techniques for the first-passage time analysis on high-level models that translate to Markov and semi-Markov processes. In particular we study exact first-passage time analysis on semi-Markov processes. Previous studies have shown that it is possible to analytically determine passage times by solving a large set of linear equations in Laplace space. The set of linear equations arises from the state transition graph of the Markov or semi-Markov process, which is usually derived from high-level models such as process algebras or stochastic Petri nets. The difficulty in passage time analysis is that even simple high-level models can produce large state transition graphs with several million states and transitions. These are difficult to analyse on modern hardware, because of limitations in the size of main memory. Whilst for Markov processes there exist several efficient techniques that allow the analysis of large chains with more than 100 million states, in the semi-Markov domain such techniques are still less developed. Consequently parallel passage time analyser tools currently only work on semi-Markov models with fewer than 50 million states. This study extends existing techniques and presents new approaches for state space reduction and faster first-passage time computation on large semi-Markov processes. We show that intelligent state space partitioning methods can reduce the amount of main memory needed for the evaluation of first-passage time distributions in large semi-Markov processes by up to 99% and decrease the runtime by a factor of up to 5 compared to existing semi-Markov passage time analyser tools. Finally we outline a new passage time analysis tool chain that has the potential to solve semi-Markov processes with more than 1 billion states on contemporary computer hardware.

Acknowledgements

I would like to thank my supervisor Jeremy Bradley for all the support and guidance he has given me throughout the project as well as for his enthusiasm about my research which always motivated me to carry on.

I would also like to thank Nicholas Dingle for giving me feedback on my experiments, providing SMARTA and helping me to overcome various technical problems I encountered during the project. Likewise I would like to thank William Knottenbelt for his support and his feedback on my written work.

Finally I would like to thank my friends and family, especially my parents, Netta, Marco, Steve and Daniel whose birthday I forgot because of the write-up.

Computers process what they are being fed. When rubbish goes in, rubbish comes out.

Trans.: EDV-Systeme verarbeiten, womit sie gefüttert werden. Kommt Mist rein, kommt Mist raus.

—

André Kostolany

Contents

CHAPTER 1

Introduction

1.1 Motivation

Whenever we time processes we would like to know the worst-case time to complete the job. This notion of time until completion is captured by response time distributions. In particular the cumulative density function of response time distributions are of interest since they allow us to make statements such as: "In 90% of all cases the job is completed after x seconds". Such intervals are also known as response time quantiles or percentiles. This performance metric is preferable to average response times, as these fail to give an intuition of the worst-case scenario. Response time quantiles are widely used in the analysis of network latencies, web servers, manufacturing systems as well as for the analysis of the quality of service of hospitals and government organisations to name a few areas of application. Response time analysis can also be performed on models such as Markov and semi-Markov processes. In this case we talk about first-passage time distributions, as response time analysis in the Markovian domain corresponds to evaluating the distribution over the time it takes to reach a set of target states from a set of source states in the transition graph of the chain.

1.1.1 Application of passage times in performance analysis

In this section we give two brief examples of applications of response time quantiles, one real-world example and one example that illustrates the passage time analysis on a semi-Markov model that has been generated from a semi-Markov stochastic Petri net (see sect. 2.2.3).

The first example was drawn from a report of the U.S. department of homeland security[31]. The report investigates the performance of the national fire services. The measure of interest is the distribution of the time it takes from the point a call is received by the emergency call center until a fire-engine arrives at the scene. The 90^{th} percentile in this case is less than 11 minutes (see fig. 1.1). The report further investigates regional and seasonal differences in response time.

Clearly such investigations are useful especially when introducing new regulation or procedures to public services or in industry, as they provide an objective measure on how the quality of service compares to earlier years.

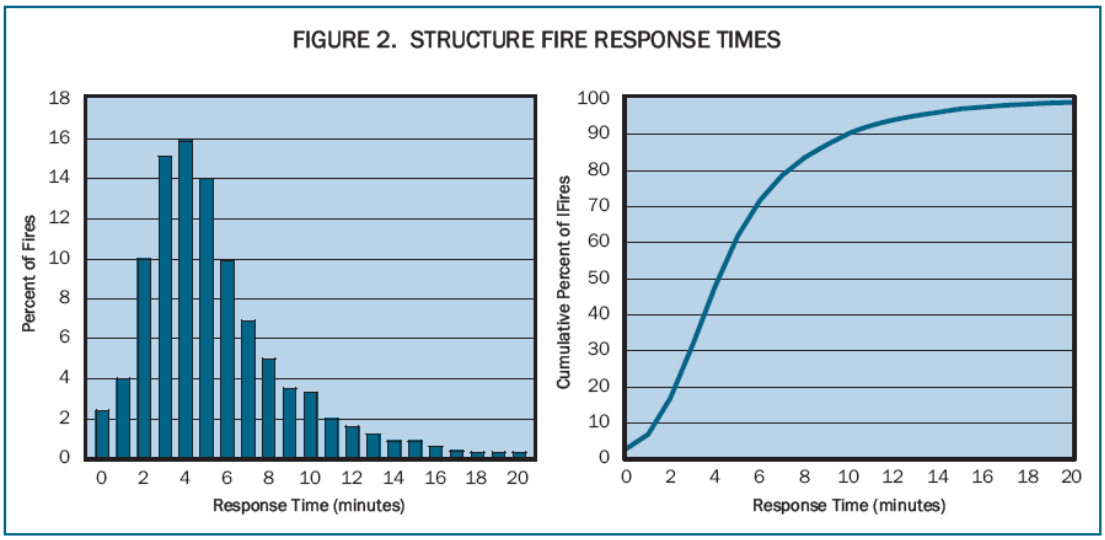


Figure 1.1: This diagram shows the general fire emergency response time distribution as described in [31]. The data originates from the National Fire Incident Reporting System (NFIRS) 5.0 data for 2001 and 2002.

The second example is a response time estimation for a large semi-Markov model. The voting model is described in detail in sect. A.1. For our experiment we computed the response time in the case where we have 60 voters, 25 voting booths and 4 central vote collection servers. The response time corresponds to the time elapsed from the point the first voter casts their vote until the last voter has completed the voting process. The 90^{th} response time percentile in this case is less than 151 seconds.

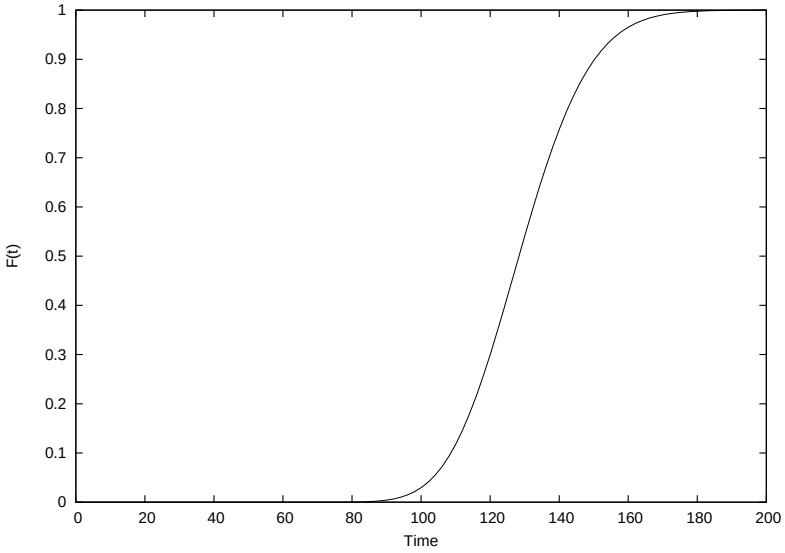


Figure 1.2: Response time cdf of voting model with 60 voters, 25 voting booths and 4 central vote collection servers. The semi-Markov model generated from the model has 106540 states.

The advantage of having a model such as the voting model is that we can simulate how changes in the setup of the e-voting system affect the quality of the voting service. Provided that the voting model approximates the real-world scenario well, studying the model can potentially save a lot of time and money when it comes to putting the system into practice.

1.2 Current state of research

As discrete state continuous time Markov and semi-Markov chains representing models of real-world systems can have several million states, it is infeasible to describe them in terms of their transition graph (see defn. 2.2). Consequently we use high-level modelling approaches such as process algebras[22] and Petri nets (see sect. 2.2) in order to describe our models. However, even though the models may be relatively compact, it is generally hard to infer response time measures directly from the model. One way of performing passage time analysis on the high-level model is to use discrete event simulation, which requires us to average the results of various simulations in order to reduce the variance of the resulting distributions. If a high degree in accuracy is needed then discrete event simulation may not be feasible. In this study we therefore concentrate on exact analytical passage time analysis in semi-Markov models using an iterative passage time algorithm (c.f. sect. 2.6).

To do first-passage time computation in semi-Markov processes (SMPs), we first have to translate the high-level model into a low-level semi-Markov representation (see sect. 2.1 and sect. 2.2). This mapping yields a graph with each vertex being a state of the semi-Markov model. There is a directed edge from one vertex to another if the transition is possible in the high-level model. Each transition encapsulates a transition latency in form of a distribution and each state has a probability distribution over all outgoing transitions, which is used to determine which transition is chosen on leaving the state. The HYDRA and SMARTA tool chains described in [22, 33] each provide a program that generates transition matrices for Markov chains and semi-Markov chains respectively. Our passage time analyser extends the SMARTA tool chain. [23] also discusses how the transition matrix generation for extremely large models can be done in parallel.

In [8] Harrison and Knottenbelt introduce an iterative passage time analysis algorithm (see sect. 2.6) for Markov and semi-Markov chains. Computing passage time densities from the low-level representation of SMPs with the iterative algorithm involves repeated sparse-matrix vector multiplication using the transition matrix of the SMP. Recall that the sparse-matrix represents the transition graph of the SMP and therefore grows with the number of states in the SMP. It is thus computationally challenging to use the iterative passage time algorithm to compute passage times for SMPs with large states spaces. Whilst there exist techniques for Markov chains [8, 22] that make exact passage time calculation on Markov chains with more than 100 million states technically feasible, similar techniques have not been developed for SMPs yet. Due to this, current parallel SMP passage time analysers such as SMARTA are limited to semi-Markov models that have fewer than 50 million states.

In [1] an innovative technique for exact aggregation of states in the low-level representation of semi-Markov processes is presented. Although exact state-by-state aggregation reduces the dimension of the transition matrix, the technique suffers from the problem that the aggregation of states causes a dramatic fill-in of the transition graph as many new transitions (see sect. 2.7) are generated by state-by-state aggregation. As the pace at which the matrix fills in during state-by-state aggregation is causing an even greater memory overhead than storing the initial transition matrix, this aggregation approach is impractical for large semi-Markov models.

1.3 Project aim

The overall aim of the project is to produce an improved first-passage time evaluation method that allows us to extend existing algorithms used in SMARTA in order to be able to evaluate semi-Markov processes with more than 50 million states. Based on the results of [1, 8, 9, 11, 15, 22, 23] our main approach for finding an improved evaluation method is to use state space partitioning strategies for aggregation of states in large semi-Markov models. Partitioning the state space entails dividing it into a number of non-intersecting subsets. The idea was proposed by Bradley, in hope that performing exact state-by-state aggregation on a partition of states

rather than on the entire flat (unpartitioned) state space would limit the explosion in the number of newly created transitions that was observed in [1]. In this report we also briefly discuss the application of our techniques for the computation of other performance metrics (see sect. 2.5). In addition to aggregation techniques based on state space partitioning we also investigate numerical techniques for faster first-passage time computation as a means to improve the speed of the passage time computation.

1.4 Contributions

The list below contains the most important results of the research conducted for this project:

1. State space partitioning for state aggregation in SMPs (chapter 3):
   (a) We define desirable properties of state space partitionings for state aggregation
   (b) We test the application of well established sparse-matrix partitioners for state space partitioning and show that they are only useful when used on small semi-Markov models (see sect. 3.2.3)
2. State-by-state aggregation on state space partitionings (chapter 4):
   (a) We introduce a state-by-state aggregation algorithm called Exact-Fewest-Paths-First (see sect. 4.1.2) which improves the Fewest-Paths-First method described in [1]
   (b) We show that for SMPs that have a small number of states, the generation of partitionings using partitioners such as PaToH and MeTiS (see sect. 3.1) and the consequent partition-by-partition aggregation using exact state aggregation drastically decreases the amount of memory and computation needed for aggregation
   (c) We introduce the concept of cheap state aggregation (see sect. 4.3.1) which is an exact state aggregation technique that finds and aggregates states in a manner such that the number of transitions in the transition matrix does not increase and show that it can be applied efficiently even when the state space becomes large
3. Atomic aggregation of partitionings and Barrier partitioning (chapter 5):
   (a) As exact state-by-state aggregation is still an expensive operation compared to the cost of the actual passage time analysis, we show that entire partitions of states can be aggregated in one go by performing a restricted passage time analysis from the predecessor to the successor states of a partition (see sect. 5.1.1)
   (b) We show that aggregation of partitions can always be done approximately at low extra cost by introducing an extra state that separates the predecessor states from the partition internal states of the partition we are aggregating (see sect. 5.1.3)
   (c) We introduce a new partitioning method called k-way *barrier* partitioning (see sect. 5.3), which reduces the amount of memory needed to perform passage time analysis on the large versions of the voting and the web-server model by up to 99%. We also show that the modified passage time algorithm for k-way barrier partitioned SMP transition matrices is exact. Furthermore our implementation of the passage time analyser using the k-way barrier partitioning is faster than the current SMARTA analyser
   (d) We describe an algorithm for finding k-way barrier partitionings in large SMPs and show that in practice the partitioner has linear complexity in the number of transitions in the semi-Markov model
   (e) We show that the 2-way barrier is well-suited for parallelisation and subsequently extend the concept to show we can improve the current parallel passage time analysis algorithm [15, 22] for the computation of a single s-point using k-way barrier partitioning

4. Path truncation (chapter 6):

   (a) We show that the iterative passage time algorithm can be improved by regularly setting small complex values in ν_r, i.e. the vector that we multiply the sparse transition matrix with, to zero. Our error analysis also shows that truncation does not induce a significant loss of accuracy

   (b) We combine the truncation technique with the k-way barrier technique to obtain a new exact passage time evaluation algorithm, which in our implementation is up to 5 times faster than the passage time analyser of SMARTA. Moreover it is possible to implement the algorithm in a manner so that it only requires a fraction of the memory needed for the same passage time computation by SMARTA (c.f. item 3c)

1.5 Publications

The following publications arose from the research conducted for this project:

- **Aggregation Strategies for Large Semi-Markov Processes**, III International Symposium on Semi-Markov Models[27]. This conference paper presents new state aggregation techniques for semi-Markov processes based on state space partitioning strategies. The paper covers large parts of chapters 3, 4 and 5.

- **Truncation of Passage Time Calculations in Large Semi-Markov models**, 25^{th} UK Performance Engineering Workshop[28]. This paper discusses the use of truncation for faster iterative passage time analysis on semi-Markov models. The paper covers the truncation section in chapter 6.

CHAPTER 2

Background

This chapter provides background information on semi-Markov processes, high-level modelling formalisms, Laplace transforms, performance analysis measures, exact state-by-state aggregation and graph partitioning. We assume that the reader of this report is familiar with basic concepts of random variables, probability distributions, stochastic processes and Markov processes.

2.1 Semi-Markov Processes (SMPs)

Semi-Markov processes are a generalisation of Markov processes. In contrast to Markov processes, where state holding times are exponentially distributed, semi-Markov processes allow any type of probability density. In the following we also refer to state holding times as *sojourn times*. Each transition from state i to j in a SMP is associated with a sojourn time distribution. The distribution represents the holding time in state i given that the transition is the next one to fire. It is possible for the transition from state i to j to have a different state holding time distribution than say the transition from state i to k if $j \neq k$. The holding time of state i is always dependent on the choice of the next outgoing transition. To reflect this in the model each state has a discrete probability distribution over its outgoing transitions. In a SMP the next state transition is always a probabilistic choice with respect to this distribution of the current state i. Having determined the transition that is to fire next, the state holding time can be sampled from its sojourn time distribution.

Semi-Markov process (SMP)

Sojourn time

Definition 2.1. Let $S = \{1, 2, \ldots, n\}$ be the state space of a SMP. Let $\{(X_n, T_n) \mid n \geq 0\}$ define a Markov renewal process, where $X_n \in S$ is the state after the n^{th} state transition has occurred and T_n, $(T_0 = 0)$ the time at which the n^{th} transition occurred. Suppose $X_n = i$. We then denote the weighted cumulative sojourn time density function for state i given that the $(n+1)^{\text{st}}$ state is j as the kernel of the SMP:

$$R(n, i, j, t) = P(X_{n+1} = j \wedge T_{n+1} - T_n \leq t \mid X_n = i)$$

This is the *kernel* of a continuous time semi-Markov chain (CTSMC). This study mainly focuses on time-homogeneous SMPs which are independent of n as the kernel does not vary with time. For time-homogeneous SMPs we can rewrite the kernel as

Kernel

$$R(i, j, t) = p_{ij} H_{ij}(t)$$

where $p_{ij} = P(X_{n+1} = j \mid X_n = i)$ for all $n \geq 0$ is the transition probability from state i to j and $H_{ij} = P(T_{n+1} - T_n \leq t \mid X_n = i,\ X_{n+1} = j)$, the cdf of the sojourn time distribution in state i given that the next state is j [1, 15, 22].

Transition graph

Transition matrix

Definition 2.2. Throughout this report we refer to the reachability graph of SMPs, as the *transition graph* of the SMP, where each state is a vertex and each transition between two states an edge between to vertices. Moreover we do not distinguish between the transition graph of a SMP and its *adjacency matrix* which we term *transition matrix*.

2.2 High-level modelling formalism for SMPs

Despite the fact that semi-Markov processes can be defined by specifying every state and transition explicitly, this approach becomes very tedious if not impossible to do by hand as the underlying model of a SMP becomes complex and large. Some of the models that we analyse in this study for instance have an underlying SMP with more than a million states and transitions. Hence, instead of describing models in terms of their low-level SMP graph representation we should rather aim at using high-level modelling formalisms that translate to finite state SMPs. That way we can specify models in a human readable format and consequently use computers to do the actual SMP generation. The actual translation from a high-level model to its underlying low-level SMP involves generating all possible states, transitions and the kernel from the high-level description of the model. In this section we introduce a Petri net modelling approach for SMPs. Information on other high-level modelling formalisms can be found in [22, 23].

2.2.1 Petri nets

Petri nets exist in various forms and are used for a wide range of models, such as models for parallel processes, queuing networks and communication protocols. The basic idea behind Petri nets is that we describe a model in terms of tokens which can move between places. We then analyse the model by observing the likeliness of certain *markings*. A marking is a vector of integers that describes how many tokens each place contains. When translating a Petri net into a SMP, markings become states and there is a transition between any two states i, j if the corresponding marking j can be reached from marking i via one transition firing. This mapping produces a SMP reachability graph for the simplest form of Petri nets, the Place-Transition nets.

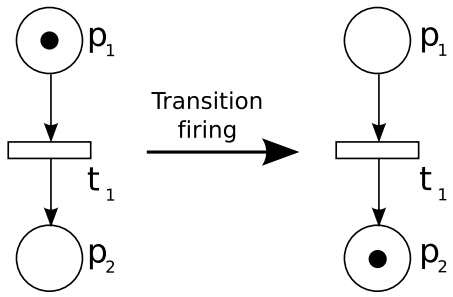


Figure 2.1: A Place-Transition net with 2 places, 1 transition and 1 token. In Petri net diagrams large empty circles represent places, empty rectangles transitions and tokens are represented as small black dots. The arrows describe the direction of a transition.

Place-Transition net

Definition 2.3. A *Place-Transition net* is a 5-tuple $PN = (P, T, I^-, I^+, M_0)$ s.t.

- $P = p_1, \ldots, p_n$ with $n \in \mathbb{N}^+$
- $T = t_1, \ldots, t_m$ with $m \in \mathbb{N}^+$
- $P \cap T = \emptyset$
- $I^-, I^+ : P \times T \mapsto \mathbb{N}_0$ are describing backward and forward incidence of places and transitions respectively. $I^-(p, t) > 0$ iff place p can fire tokens through transition t. In other

words $I^-(p,t) > 0$ iff t is an outgoing transition of p. Similarly $I^+(p,t) > 0$ iff place p can receive tokens through transition t, i.e. t is an incoming transition of p.

- $M_0 : P \mapsto \mathbb{N}_0$ is the initial marking of the model.

Instead of translating every possible marking into a state in the underlying SMP we simply say that the set of all markings reachable from M_0 is the *state-space* of the underlying SMP. *State-space*

Definition 2.4. In a Place-Transition net $PN = (P, T, I^-, I^+, M_0)$ we have the following firing rules

- The *marking* is a function $M : P \mapsto \mathbb{N}_0$, such that $M(p)$ is the number of tokens on place p. *Marking*
- $M[t >$ implies that transition $t \in T$ is enabled in marking M. We have $M[t >$ iff $M(p) \geq I^-(p,t)$ for all $p \in P$. A function that takes a marking and a transition and decides whether the transition is enabled or not on the basis of abundance of tokens on preceding places is a *net-enabling function*. $M[t >$ *Net-enabling function*
- If a transition $t \in T$ is enabled in marking M and fires we have $M'(p) = M(p) - I^-(p,t) + I^+(p,t)$ for all $p \in P$, where M' is the new marking. We say M' is directly reachable from M and write $M[t > M'$ or simply $M \rightarrow M'$. $M \rightarrow M'$

For a transition t to be enabled there have to be $I^-(p,t)$ tokens on each of its input places p. When the transition fires, $I^-(p,t)$ tokens are removed from each input place $p \in P$ and $I^+(p,t)$ tokens are added to every output place $p \in P$. A system represented by a Petri net can deadlock if there exists a marking which has no enabled outgoing transitions.

2.2.2 Generalised stochastic Petri nets

From a Place-Transition net we can derive the reachability graph of a SMP. However, in order to use Petri nets as a high-level formalism for SMPs, we also have to define the notion of sojourn time distribution and transition probability in our Petri net model.

Definition 2.5. A *Generalised stochastic Petri net (GSPN)* is a 4-tuple $GSPN = (PN, T_1, T_2, C)$ where *Generalised stochastic Petri net (GSPN)*

- $PN = (P, T, I^-, I^+, M_0)$ is the underlying Place-Transition net
- $T_1 \subseteq T$ is the set of timed transitions, $T_1 \neq \emptyset$
- $T_2 \subseteq T$ is the set of immediate transitions, with $T_1 \cap T_2 = \emptyset,\ T = T_1 \cup T_2$
- $C = (c_1, \ldots, c_{|T|})$ where

$$c_i = \begin{cases} \text{a rate} \in \mathbb{R}^+ \text{of an exponential probability distribution modelling the firing delay of transition if } t_i \in T_1 \\ \\ \text{a weight} \in \mathbb{R}^+ \text{specifying the relative firing frequency of transition if } t_i \in T_2 \end{cases}$$

  where both types of c_i may be marking dependent.

In GSPNs the transition probability depends on the marking under which the transition is enabled as some transitions are more likely to fire when some of the outgoing transitions are not enabled in a certain place. Timed transitions are dominated by immediate transitions which fire in time zero, whilst timed transitions in GSPNs have exponentially distributed firing delays (see [22] for further information). These delays may depend on individual markings. Hence it is generally hard to reduce or aggregate places in the high-level model or to spot certain sets of markings in a Petri net that can be simplified in the resulting SMP. It also makes sense to distinguish between markings in which immediate transitions are enabled and those in which they are disabled.

Vanishing marking
Tangible marking

Definition 2.6. A *vanishing marking* is a marking in which an immediate transition is enabled. Clearly the sojourn time in such a marking is zero. A *tangible marking* is one where no immediate transition but at least one timed transition is enabled. We denote the set of vanishing markings by $\mathcal{V}$ and the set of tangible markings by $\mathcal{T}$.

Note. In later sections we also refer to vanishing and tangible states in SMPs, which are semantically equivalent to vanishing and tangible markings as markings in a Petri net are interpreted as states in the low-level reachability graph of a SMP.

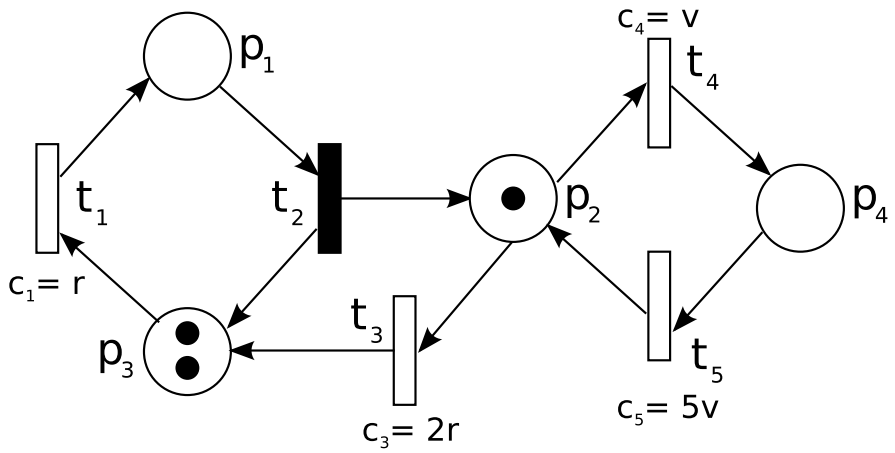


Figure 2.2: Immediate transitions have black rectangles. Note that the timed transitions have exponential sojourn time distributions with rates dependent on some values v and r.

Stochastic Petri nets

GSPNs are far more expressive than their Place-Transform counterparts. In fact it can be shown that the reachability graph of a GSPN which has $\mathcal{V} = \emptyset$ is isomorphic to some continuous time Markov-Chain. These special GSPNs are called *stochastic Petri nets*. Furthermore it is possible to transform a GSPN with $\mathcal{V} \neq \emptyset$ to one with $\mathcal{V} = \emptyset$ without corrupting measures such as steady-state probabilities or passage times in the underlying model[22].

2.2.3 Semi-Markov stochastic Petri nets

The final generalisation of Petri nets that we introduce in this section is the semi-Markov stochastic Petri net (SM-SPN). In a SM-SPN we can choose any probability distribution for the firing delay of timed transitions. Furthermore timed transitions in this model have weights and priorities, such that we can sample transitions according to a probability distribution over all enabled transition with high priorities when more than one transition is enabled in a given marking. This is in accordance with the semi-Markov definition in sect. 2.1.

Definition 2.7. A SM-SPN is a 4-tuple $(PN, \mathcal{P}, \mathcal{W}, \mathcal{D})$ such that

- $PN = (P, T, I^-, I^+, M_0)$ is the underlying Place-Transition net
- $\mathcal{P} : T \times \mathcal{M} \mapsto \mathbb{N}_0$, denoted $p_t(m)$ is a marking dependent priority function for a transition.
- $\mathcal{W} : T \times \mathcal{M} \mapsto \mathbb{R}_+$, denoted $w_t(m)$ is a marking dependent weight function for a transition that is used to model probabilistic choice.
- $\mathcal{D} : T \times \mathcal{M} \mapsto [0, 1]$, denoted $d_t(m)$ is a marking dependent cumulative distribution function for the firing-delay of a transition.

where $\mathcal{M}$ is the set of all markings for a given SM-SPN.

Clearly these information allow us to derive the kernel of a semi-Markov process (see defn. 2.1). Finally we need to redefine the transition enabling function to take the priority levels of transitions into account.

Definition 2.8. In a SM-SPN $(PN, \mathcal{P}, \mathcal{W}, \mathcal{D})$ we have the following functions

- $\mathcal{E}_N : \mathcal{M} \mapsto P(T)$ is a net-enabling function with the same properties as $M[t >$ in defn. 2.4.
- $\mathcal{E}_P : \mathcal{M} \mapsto P(T)$ is a function which specifies priority-enabled transitions from a given marking.

Given a marking m the function $\mathcal{E}_P(m)$ selects only those net-enabled transitions that have the highest priority, i.e. the largest value $p_t(m)$ among all $\mathcal{E}_N(m)$ enabled transitions of m. Each of the priority-enabled transitions is fired with probability

$$P(t \in \mathcal{E}_P(m) \text{ fires}) = \frac{w_t(m)}{\sum_{t' \in \mathcal{E}_P(m)} w_{t'}(m)}$$

just as we described in sect. 2.1. Having made the probabilistic choice of which enabled transition fires next, the sojourn time, i.e. the delay before the firing occurs, has the cumulative distribution $d_t(m)$.

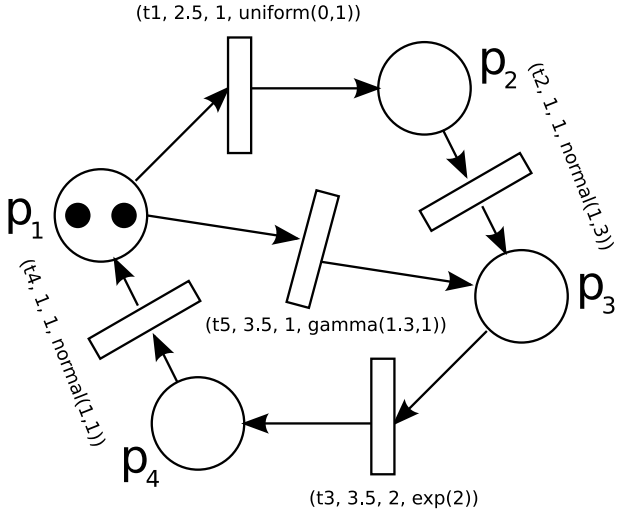


Figure 2.3: Transitions now take parameters (name, weight, priority, sojourn time distn).

2.2.4 SM-SPN models used in this study

It is generally possible to transform GSPNs and SPNs into SM-SPNs (for details see [22]). The models we analyse in our study are all generated from GSPNs and SM-SPNs. Therefore it is feasible to measure their steady-state distribution, transient distribution, passage time distributions and other common semi-Markov measures (see sect. 2.5). The Petri nets for the voting, web-server and courier model are explained in detail in appendix A. As the original courier model is a GSPN we define the smcourier model to be the SM-SPN version of the courier model.

2.3 Laplace transforms

A Laplace transformation is a mapping from a real-valued function $f(t)$ to a complex-valued function. The mapping is invertible, hence Laplace functions can be mapped back to a r.v. function. In the following we define the Laplace transform and show the benefits of representing the kernel of a SMP in Laplace space rather than in real space.

Laplace transform

Definition 2.9. The *Laplace transform* $L\{f(t)\}(s)$ with $t \in \mathbb{R}_0^+,\ s \in \mathbb{C}$ of r.v. function $f(t)$ is defined as

$$L\{f(t)\}(s) = \int_0^\infty e^{-st} f(t)\ dt$$

Exponential order

where $f(t)$ must be of *exponential order*, i.e. $|f(t)| < e^{\alpha t},\ \alpha > 0$ for each t in the domain of $f(t)$. Furthermore $f(t)$ is only allowed to have a finite number of finite discontinuities.

Note. The most commonly used probability density functions, e.g. uniform, normal, exponential, etc., are all of exponential order and it can be shown that they all have unique Laplace transforms. The uniqueness of the Laplace transforms allows us to recover the original r.v. function $f(t)$ from $L\{f(t)\}(s)$. The condition $t \in \mathbb{R}_0^+$ is not overly restrictive in our case as the probability distributions in the kernel represent time delays.

Theorem 2.1. Let $f(t)$ be a real-valued probability density function on $[0, \infty]$ and $F(t)$ be the corresponding cumulative density function, i.e.

$$F(t_1) = \int_0^{t_1} f(t)\ dt$$

then

$$L\{F(t)\}(s) = L\{f(t)\}(s)/s$$

Proof. see [20] ∎

Note. By thm. 2.1 we can represent the weighted cumulative sojourn time density functions in the kernel (see defn. 2.1) in terms of their underlying pdf Laplace transforms and later recover the Laplace transforms of the cumulative density functions by dividing the pdf Laplace transforms by s. In practice we represent the kernel in terms of its pdf Laplace transforms.

Definition 2.10. Let $f(t)$, $g(t)$ be two r.v. functions with $t \in \mathbb{R}_0^+$ then

$$h(t) = f(t) * g(t) = \int_0^\infty f(\tau) * g(t - \tau)\ d\tau$$

Convolution

is the *convolution* of f and g.

Theorem 2.2. The Laplace transform of the convolution of two r.v. functions $f(x)$, $f(y)$ with $x, y \in \mathbb{R}_0^+$ is the product of the Laplace transforms of $f(x)$ and $f(y)$, i.e.

$$L\{f(x) * g(x)\}(s) = L\{f(x)\}(s)\ L\{g(x)\}(s)$$

Proof. see [20] ∎

Convolutions occur whenever we want to write a random variable Z as the sum of other random variables. It is not hard to see that in real space these integrals are difficult to compute in general. When doing passage time analysis (see sect. 2.5.2) on SMPs we need to perform many convolutions of pdfs of the sojourn time distributions from the kernel. Thus it is hard to do passage time analysis in real space. In Laplace space on the other hand it is straightforward to compute the Laplace transform of a convolution of many pdfs, as we merely have to multiply their individual Laplace transforms.

Theorem 2.3. The Laplace transform is a linear transformation. Let $f(t)$, $g(t)$ be two r.v. functions with $t \in \mathbb{R}_0^+$ and $a, b \in \mathbb{R}$ two constants then

$$L\{af(t) + bg(t)\}(s) = aL\{f(t)\}(s) + bL\{g(t)\}(s)$$

Proof. see [20] ∎

2.4 Laplace transform inversion

As mentioned before it is possible to recover $f(t)$ from its Laplace transform $L\{f(t)\}$ as the Laplace transform of $f(t)$ is unique.

Definition 2.11. The *inverse of the Laplace transform of* $f(t)$ is

Inverse Laplace transform

$$L^{-1}\{L\{f(t)\}(s)\} = f(t) = \frac{1}{2\pi i}\int_{a-i\infty}^{a+i\infty} e^{st}L\{f(t)\}(s)\ ds \tag{2.1}$$

where a is a real number which lies to the right of all singularities of $L\{f(t)\}(s)$.

Equation 2.1 is known as the *Bromwich contour inversion integral*. Because of the many convolutions that need to be computed during passage time analysis it is impossible to keep an exact representation of the Laplace transforms of all distributions in the kernel of a SMP. Instead we only keep those samples of the transforms $L\{f(t)\}(s)$ in memory that are required to retrieve $f(t)$ for the values of t we are interested in. We denote the points for which we want to calculate $f(t)$ as *t-points*. Similarly we refer to Laplace transform points of $L\{f(t)\}(s)$, which we need to recover $f(t)$ for all required t-points, as *s-points*. The choice of s-points depends on the type of numerical Laplace inversion method we use to recover $f(t)$ for a given set of t-points. As a consequence of thms. 2.2 and 2.3 we can limit passage time analysis to those samples needed for numerical inversion. This is highly beneficial as it simplifies the way we can represent Laplace transforms in practice.

t-point
s-point

2.4.1 Numerical Laplace transform inversion

In [15] Bradley, Dingle, Harrison and Knottenbelt show how selected samples from a Laplace transform of a pdf $f(t)$ can be used to retrieve $f(t)$ and $F(t)$ using numerical Laplace inversion. In practice this has the advantage that each Laplace transform of a pdf can be represented as a set of complex numbers which has constant memory requirements no matter how complex the underlying functions of the Laplace transforms become. In this section we present the *Euler Laplace inversion* and the *Laguerre Laplace inversion* method, which are two methods that are well-suited for Laplace inversion after performing passage time analysis on SMPs. The following description of the two methods is a summary of the description in [22].

Euler inversion
Laguerre inversion

2.4.1.1 Euler method

Suppose we want to recover $f(t)$ for a given t-point. First note that we can rewrite eq. 2.1 by substituting $s = a + iu$

$$f(t) = \frac{1}{2\pi i}\int_{-\infty}^{\infty} e^{(a+iu)t}L\{f(t)\}(a+iu)\ du$$

and since

$$e^{(a+ui)t} = e^{at}(\cos(ut) + i\sin(ut))$$

we have

$$f(t) = \frac{2e^{at}}{\pi}\int_{0}^{\infty} Re(L\{f(t)\}(a+iu))\ \cos(ut)\ du \tag{2.2}$$

which now is a real-valued integral. Equation 2.2 can be approximated by

$$\int_a^b f(t)\ dt \approx h\left(\frac{f(a)+f(b)}{2} + \sum_{k=1}^{n-1} f(a+kh)\right) \tag{2.3}$$

with $h = (b-a)/n$. We set $h = \pi/2t$ and $a = A/2t$, where A is an empirical constant of value 19.1. This gives the following alternating series

$$f(t) \approx \frac{e^{A/2}}{2t}Re\left(L\{f(t)\}\left(\frac{A}{2t}\right)\right) + \frac{e^{A/2}}{2t}\sum_{k=1}^{\infty}(-1)^k Re\left(L\{f(t)\}\left(\frac{A+2k\pi i}{2t}\right)\right) \tag{2.4}$$

To speed up the convergence of the alternating series we deploy Euler summation. Euler summation works as follows. We first calculate the first n elements of the series explicitly. Subsequently we calculate the next m elements of the series as follows:

$$E(t,m,n) = \sum_{k=0}^{m} 2^{-k} \binom{k}{m} \sum_{j=0}^{n+k} (-1)^j Re\left(L\{f(t)\}\left(\frac{A+2j\pi i}{2t}\right)\right)$$

where the truncation error of $E(t,m,n)$ can be estimated by

$$|E(t,m,n) - E(t,m,n+1)|$$

Empirical studies have shown that $n = 20$, $m = 12$ yields a truncation error of 10^{-8}. In practice we have to calculate $m + n + 1$ Laplace transforms for each t-point we are interested in. This implies that the more t-points we want the more s-points we need to consider when doing passage time analysis with subsequent Euler Laplace inversion.

2.4.1.2 Laguerre method

The downside of the Euler Laplace inversion is the increasing computational burden that comes with computing $f(t)$ for a large number of t-points. The Laguerre method allows us to recover $f(t)$ for an arbitrary number of t-points using a fixed number of s-points that is independent of the number of t-points. The disadvantage of the Laguerre method is that it is difficult to guess the number of s-points needed, prior to performing passage time analysis. Also if the kernel of a SMP contains distributions that have discontinuities, it is advisable to use Euler inversion as Laguerre inversion works best on Laplace transforms of smooth distributions.

We can represent $f(t)$ in terms of its Laguerre series

$$f(t) = \sum_{n=0}^{\infty} q_n l_n(t),\ t \geq 0$$

where

$$l_n(t) = \left(\frac{2n-1-t}{n}\right) l_{n-1}(t) - \left(\frac{n-1}{n}\right) l_{n-2}(t)$$

with $l_0 = e^{t/2}$ and $l_1 = (1-t)e^{t/2}$ and

$$q_n = \frac{1}{2\pi r^n} \int_0^{2\pi} Q(re^{iu}) e^{-inu}\, du \tag{2.5}$$

where $r = (0.1)^{4/n}$ and $Q(z) = (1-z)^{-1} L\{f(t)\}((1+z)/(2(1-z)))$. Equation 2.5 can be approximated numerically using the trapezoidal rule

$$q_n \approx \frac{1}{2nr^n} \left(Q(r) + (-1)^n Q(-r) + 2 \sum_{j=1}^{n=1} (-1)^j Re(Q(re^{\pi ji/n})) \right) \tag{2.6}$$

We have $|l_n(t)| \leq 1$ for all n, hence the convergence of the Laguerre series depends solely on the decay rate of q_n as n becomes large. Convergence of q_n can be improved by using exponential dampening and scaling (see [22] for further information). Assume that by applying these techniques we need p_0 (say $p_0 = 200$) terms until q_n is negligible small. This allows us to compute each q_n with a fixed number of $2p_0$ trapezoids. Since q_n is independent of t and $Q(z)$ only has one occurrence of $L\{f(t)\}$ it can be seen that we can obtain $f(t)$ for an arbitrary number of t-points at the constant cost of $2p_0$ evaluations of $L\{f(t)\}$. As we do not know p_0 in advance we need to guess p_0, calculate the necessary Laplace transforms for the required s-points and check if q_n has already converged. If not we apply further scaling and calculate Laplace transforms for further s-points until q_n converges.

2.5 Measures in SMP analysis

This section introduces common measures in performance analysis research which are used for studying SMPs and Markov chains originating from different areas of performance analysis such as network and hardware performance, traffic simulation, simulation of biological processes and various other fields. In this study we mainly look at the impact of aggregation techniques on first-passage time analysis in SMPs at equilibrium. Other measures are briefly introduced in this section for completeness but [22, 23] should be consulted for more detailed information.

2.5.1 Transient and steady-state distribution

Transient distribution

Steady-state distribution

Equilibrium state

Suppose we run a finite SMP with set of states $S = \{1, \ldots, n\}$ for a certain amount of time and record the amount of time spent in every state. The *transient distribution* is a probability vector $\pi^{(t)} = \{\pi_1, \ldots, \pi_n\}$ for a given time $t > 0$, where each element π_i represents the proportion of time t that the SMP has spent in state i. Note that $\pi^{(t)}$ is dependent on the starting state of the SMP. Informally we can say that the *steady-state distribution* describes the probability of being in a particular state in the SMP, given that the SMP has run for a very long time, i.e. $t \to \infty$. In contrast to the transient distribution the steady-state distribution is independent of the starting state provided that every state can reach every other state in the reachability graph of the SMP. In an empirical experiment we say that an SMP has reached steady-state or *equilibrium state* when the transient distribution has converged to the steady-state distribution. For a formal definition of the steady-state distribution see pp.19-21 in [22].

2.5.2 Passage time analysis in semi-Markov models

First-passage time distribution

Another common measure in performance analysis is the probability distribution of the time it takes to get from one system state to another. This distribution is known as the *first-passage time distribution* or simply the *passage time distribution*. In terms of SMPs the first-passage time is a probability distribution of the fastest transition time from any state $i \in \vec{i}$ to any state $j \in \vec{j}$ where $\vec{i}$ and $\vec{j}$ are the set of source and target states respectively. Note that we can measure the transient first-passage time distribution as well as the first-passage time distribution at equilibrium. In the following we assume that we deal with the steady-state case unless stated otherwise.

In most semi-Markov models there is an infinite number of paths from each $i \in \vec{i}$ to each $j \in \vec{j}$. Each of these paths has a probability of being chosen. Also since the firing delay distribution of all transitions in a particular path are known, we can compute the passage time distribution of that path, i.e. the probability distribution over the time it takes to walk the entire path, by convolving the sojourn time distributions of all transitions that form the path. Since the reachability graph may have loops, it is possible that certain transitions contribute multiple times to this distribution. The passage time distribution from the set of source states $\vec{i}$ to the set of target states $\vec{j}$ is obtained by branching the passage time distributions of all possible paths from states in $\vec{i}$ to states in $\vec{j}$. To ensure that more probable paths have a greater impact on the final passage time, we need to weight each path's passage time distribution by its path probability before branching it. In the following we formally describe the calculation of first-passage time distributions in SMPs at equilibrium.

Definition 2.12. Suppose we have a SMP with state space S and kernel $R(i, j, t)$, $0 < i, j \leq |S|$. We define the first-passage time from state i to the set of states $\vec{j}$ in a time homogeneous SMP, i.e. in a SMP with time invariant kernel, as follows

$$P_{i\vec{j}} = \inf\{u > 0 \mid Z(u) \in \vec{j} \wedge Z(0) = i\}$$

where $Z(u)$ is the system state at time u. $P_{i\vec{j}}$ has probability density function $f_{i\vec{j}}(t)$ and cdf

$$F_{i\vec{j}}(t_1) = P(P_{i\vec{j}} < t_1) = \int_0^{t_1} f_{i\vec{j}}(t)\, dt$$

We write the Laplace transform of $f_{i\vec{j}}(t)$ as $L_{i\vec{j}}(s) = L\{f_{i\vec{j}}(t)\}(s)$. Since the kernel of the SMP is defined in terms of the cdfs of the sojourn time distributions for all transitions we use the Laplace-Stieltjes transform to define $L_{i\vec{j}}(s)$ in terms of $F_{i\vec{j}}$[15, 22].

$$L_{i\vec{j}}(s) = \int_0^\infty e^{-st}\, dF_{i\vec{j}}(t) = \int_0^\infty e^{-st}\, \left(\frac{d}{dt}F_{i\vec{j}}(t)\right)\, dt = \int_0^\infty e^{-st}\, f_{i\vec{j}}(t)\, dt$$

Analogously we denote the Laplace transform of the weighted sojourn time density function for the transition from state i to k by

$$r^*_{ik}(s) = \int_0^\infty e^{-st}\, dR(i,k,t) \tag{2.7}$$

The Laplace transform of $f_{i\vec{j}}(t)$ is

$$L_{i\vec{j}}(s) = \sum_{k \in S \setminus \vec{j}} r^*_{ik}(s) L_{k\vec{j}}(s) + \sum_{k \in \vec{j}} r^*_{ik}(s),\ 1 \leq i \leq |S| \tag{2.8}$$

To solve eq. 2.8 we need to solve a set of $|S|$ linear equations regardless of the number of states i for which we actually need to know $L_{i\vec{j}}(s)$. Although there exists an exact solution to the system of linear equations, in practice we only approximate the real solution. In order to calculate $L_{\vec{i}\vec{j}}(s)$, the Laplace transform of the steady-state first-passage time pdf from the set of states $\vec{i}$ to the set of states $\vec{j}$, we calculate

$$L_{\vec{i}\vec{j}}(s) = \sum_{k \in \vec{i}} \alpha_k L_{k\vec{j}}(s) \tag{2.9}$$

where weight α_k is the conditional probability at equilibrium that the system is in state k given that the system is in the set of states $\vec{i}$.

$$\alpha_k = \begin{cases} \pi_k / (\sum_{j \in \vec{i}} \pi_j) & k \in \vec{i} \\ 0 & \text{otherwise} \end{cases} \tag{2.10}$$

2.6 Numerical methods for first-passage time analysis

The set of $|S| = N$ linear equations needed to compute $L_{i\vec{j}}(s)$, for all $i \in S$ can be written in matrix form as follows [9, 15, 22]

$$\begin{pmatrix} 1 & -r^*_{12}(s) & \dots & -r^*_{1N}(s) \\ 0 & 1 - r^*_{22}(s) & \dots & -r^*_{2N}(s) \\ 0 & -r^*_{32}(s) & \dots & -r^*_{3N}(s) \\ \vdots & \vdots & \ddots & \vdots \\ 0 & -r^*_{N2}(s) & \dots & 1 - r^*_{NN}(s) \end{pmatrix} \begin{pmatrix} L_{1\vec{j}}(s) \\ L_{2\vec{j}}(s) \\ L_{3\vec{j}}(s) \\ \vdots \\ L_{N\vec{j}}(s) \end{pmatrix} = \begin{pmatrix} r^*_{1\vec{j}}(s) \\ r^*_{2\vec{j}}(s) \\ r^*_{3\vec{j}}(s) \\ \vdots \\ r^*_{N\vec{j}}(s) \end{pmatrix} \tag{2.11}$$

where $r^*_{i\vec{j}}(s) = \sum_{k \in \vec{j}} r^*_{ik}(s)$ like in eq. 2.8.

2.6.1 Iterative approach

It is possible to solve eq. 2.11 using standard linear equation solvers such as *Jacobi, Successive over relaxation (SOR)* or *Conjugate gradient square(CGS)*. Although both Jacobi and CGS can be parallelised, the iterative approach described in this section has been shown to be the best algorithm for solving systems of linear equations in passage time analysis[22].

Definition 2.13. Using the same notation as in defn. 2.1 we define the r^{th} *transition first-passage time* from state i to the set of states target states $\vec{j}$ as

r^{th} transition first-passage time

$$P^{(r)}_{i\vec{j}} = \inf\{u > 0 \mid Z(u) \in \vec{j} \wedge 0 < N(u) \leq r \wedge Z(0) = i\} \tag{2.12}$$

i.e. the time taken to enter a state in $\vec{j}$ for the first time via a path that has at most r state transitions starting in state i at time 0. Let $L^{(r)}_{i\vec{j}}(s)$ be the Laplace transform of $P^{(r)}_{i\vec{j}}$ and

$$L^{(r)}_{\vec{j}}(s) = \left(L^{(r)}_{1\vec{j}}(s), L^{(r)}_{2\vec{j}}(s), \ldots, L^{(r)}_{N\vec{j}}(s)\right) \tag{2.13}$$

Similar to computing reachability in graphs we can compute $L^{(r)}_{\vec{j}}(s)$ as

$$L^{(r)}_{\vec{j}}(s) = U\left(I + U' + U'^2 + \cdots + U'^{(r-1)}\right) e_{\vec{j}} \tag{2.14}$$

where U is a matrix with elements $u_{pq} = r^*_{pq}(s)$, U' the same matrix as U with all rows $j \in \vec{j}$ being all zero and $e_{\vec{j}}$ the column vector that has 1's in all rows $j \in \vec{j}$ and 0's everywhere else. The initial multiplication with U is needed in case the set of source states intersects with the sets of targets states, which happens if we time cycles. Matrix U' ensures that paths end as soon as they have reached the set of target states. It is straightforward to see that

$$P_{i\vec{j}} = P^{(\infty)}_{i\vec{j}} \text{ and therefore } L_{i\vec{j}}(s) = L^{(\infty)}_{i\vec{j}}(s) \tag{2.15}$$

Having computed $L^{(r)}_{\vec{j}}(s)$ we calculate $L^{(r)}_{\vec{i}\vec{j}}(s)$

$$L^{(r)}_{\vec{i}\vec{j}}(s) = \alpha L^{(r)}_{\vec{j}}(s) \tag{2.16}$$

where vector α is as defined in eq. 2.10. In practice we change the calculation of $L^{(r)}_{\vec{i}\vec{j}}(s)$ slightly. First we calculate vector ν_0

$$\nu_0 = \alpha U \tag{2.17}$$

and subsequently

$$\nu_i = \nu_{i-1}U' \ , \ i \geq 1 \tag{2.18}$$

We sum all ν_i in ν

$$\nu = \sum_{i=0}^{r} \nu_i \tag{2.19}$$

and compute

$$L^{(r)}_{\vec{i}\vec{j}}(s) = \nu e_{\vec{j}} \tag{2.20}$$

as soon as ν has converged. We say that ν has converged after the i$^{\text{th}}$ iteration if

$$|Re(\nu_{ij})| < \epsilon \wedge |Im(\nu_{ij})| < \epsilon \tag{2.21}$$

for all vector elements ν_{ij} of ν_i for some $\epsilon > 0$. All our experiments use $\epsilon = 10^{-16}$ as observations have shown that $\epsilon = 10^{-8}$ does not always ensure convergence. This notion of convergence is sensible as we expect the absolute values of the elements in ν_i to decrease as i becomes larger since ν_i always represents paths of length i, which should have lower path probabilities than paths of length $< i$ and thus contribute less to the final Laplace transform of the s-point.

It is worth noting that although a single iteration of the passage time analyser requires at most as many complex multiplications as there are non-zero elements in the matrix, empirical evidence in [9] shows that the actual actual complexity of a single iteration is $O(N \log N)$ when the matrix multiplication is done in parallel.

2.7 Exact state aggregation

As mentioned in sect. 2.6 the iterative passage time algorithm is preferable to other numerical linear equation solvers as it is substantially faster in large SMPs. However since the complexity of the iterative passage time algorithm is $O(|S|log(|S|))$, the runtime will increase faster than the size of the state space S[15]. Moreover in large systems, reducing the number of *intermediate states* between the set of source states $\vec{i}$ and the set of target states $\vec{j}$, i.e. states that are neither in $\vec{i}$ nor $\vec{j}$, could potentially make the first-passage time calculation faster, provided the we do not increase the number of transitions while aggregating intermediate states. In [1] Bradley, Dingle and Knottenbelt describe a method which aggregates individual states without changing the passage time distribution of the SMP. We refer to this technique as *exact state aggregation* or *exact state-by-state aggregation of the SMP*.

Exact state-by-state aggregation

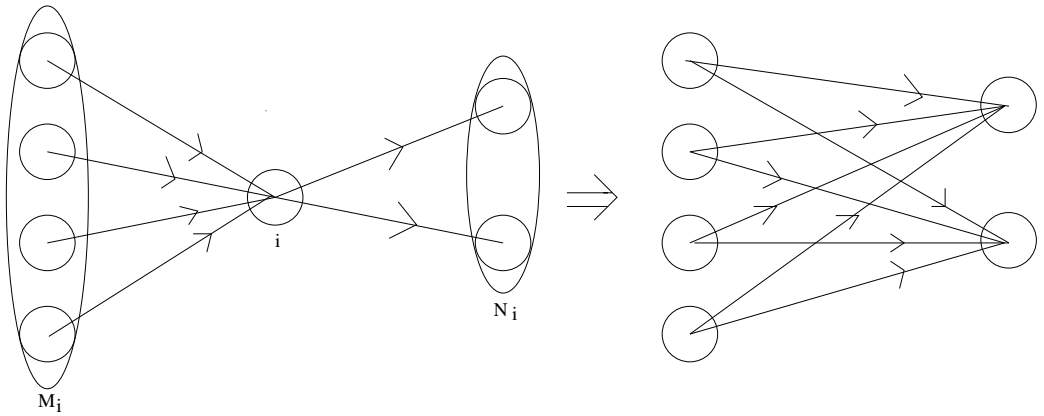


Figure 2.4: On the left hand side the we see the transition diagram of the SMP before the aggregation of state i. The right hand side shows the transition diagram after state i has been aggregated.

Predecessor states
Successor states

Suppose we want to aggregate state i. This state has a set of *predecessor states* M_i (i.e. states that have outgoing transitions to state i) and *successor states* N_i (i.e. states that state i has outgoing transitions to). To aggregate this state using the exact state aggregation technique described in [1] we need to perform the following two steps. Firstly we have to remove any transition from state i to itself. If state i has no such transition then we can skip this step. If it does we need to distribute the transition probability and its firing delay among the remaining outgoing transitions of state i. A state i with a self-cycle is its own predecessor and successor state, hence $M_i \cap N_i \cap \{i\} = \{i\}$. We start by normalising the probabilities of the remaining outgoing transitions of state i. Let p_{ij} denote the probability of a transition from state i to state j then

$$p'_{in} = \frac{p_{in}}{1 - p_{ii}}$$

is the new probability of the transition from i to n for all $n \in N_i \backslash \{i\}$ after we have removed the self-cycle. Next we add the delay of the self-cycle to the sojourn time distributions of the remaining transitions

$$L'_{in}(s) = \frac{1 - p_{ii}}{1 - p_{ii}L_{ii}(s)} L_{in}(s)$$

Having removed the cycle we delete the transition from i to itself from the transition graph. We can now assume that $M_i \cap N_i \cap \{i\} = \emptyset$. The next thing we need to do is to cut the connections between state i and its predecessor and successor states. To do this we first compute the probability and the Laplace transform of the passage time for each two-step path from m to n with $m \in M_i$ and $n \in N_i$ that has state i as its middle state. To calculate these distributions, we convolve the sojourn time distributions of all two-step transitions of the form $m \to i$ and

$i \to n$. Since we represent all sojourn time distribution in terms of their Laplace transform we simply calculate

$$L'_{mn}(s) = L_{mi}(s)L_{in}(s)$$

where $L'_{mn}(s)$ is the Laplace transform of the convolution of the two pdfs of the sojourn time distribution of the two transitions. If there already exists a direct transition from m to n with sojourn time distribution $L_{mn}(s)$ then we have to branch it with the two-step transition to ensure that no information is lost when state i is removed. To branch two transitions we need to compute their combined probability as well the Laplace transform of the sojourn time distribution for the new transition. The new transition probability of the transition from m to n is simply

$$p''_{mn} = p_{mn} + p_{mi}p_{in}$$

the sojourn time is a weighted average of the two Laplace transform samples

$$L''_{mn(s)} = \frac{p_{mn}}{p''_{mn}}L_{mn}(s) + \frac{p_{mi}p_{in}}{p''_{mn}}L'_{mn}(s)$$

If there exists no direct transition from m to n then we simply take the two-step transition as the new transition from m to n. Note that the sum of the probabilities of all outgoing transitions of state m add up to one once we have computed p''_{mn} for all $n \in N_i$. Having computed all possible transitions from a particular m to all $n \in N_i$ we can remove the transition from m to i from the transition graph of the SMP. We repeat the same process for all $m \in M_i$. After that we simply remove state i along with all its outgoing transitions from the transition graph. In [1] it has been shown that performing state aggregation in this manner does not influence the result of the final first-passage time calculation as long as none of the source or target states is aggregated.

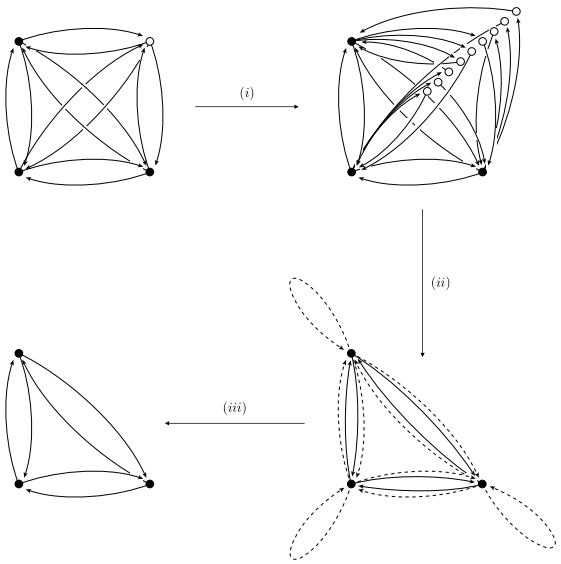


Figure 2.5: Reducing a complete 4 state graph to a complete 3 state graph [1].

In sect. 4.1.2 we introduce a formula that allows us to compute the exact number of new transitions created by the exact aggregation of a state i. In essence this is the number of new transitions between predecessor and successor states after the aggregation minus $(|M_i \backslash \{i\}| + |N_i \backslash \{i\}| + |M_i \cap N_i \cap \{i\}|)$, where a new transition between a predecessor and a successor state is a transition between a pair of predecessor and successor states that did not exist prior to the aggregation of state i (see fig. 4.1). It is easy to see that after the aggregation of state i each predecessor of i is connected to every successor of i (see fig. 2.4). Unless there are many direct connections between predecessor and successor state it is likely that the aggregation of state

i creates new transitions in the transition graph of the SMP. Experiments in [1] have shown that exact state-by-state aggregation creates a large number of temporary transitions during aggregation, even if we choose the order in which we aggregate intermediate states intelligently using techniques such as the fewest-paths-first state sorting technique (see sect. 4.1.1). Extra transitions are highly unwanted as they require additional memory and increase the amount of computation needed to perform aggregation. In practice extra transitions imply a fill-in of the adjacency matrix that represents the reachability graph of the SMP. In the following we use the terms *transition matrix fill-in* and *transition explosion* interchangeably, as there is a bijective mapping between the two representations of the SMP. In chapter 4 we present new techniques for finding a state ordering for state-by-state aggregation which significantly reduces the transition matrix fill-in compared to existing methods. These techniques are based on sparse graph/matrix partitioning algorithms which we use to partition the reachability graph/adjacency matrix of the SMP so that we can subsequently aggregate entire partitions of states using state-by-state aggregation.

Transition matrix fill-in

2.8 Graph partitioning

Graphs are widely used models for representing data dependencies. The close relationship between data and computations performed on sets of data naturally relates to the structure of graphs [2]. The agility of graph models allows them to be applied to a vast number of computational challenges, which explains the ubiquity of graph models in computer science. We can easily map data to vertices and use edges to model computations between the data-vertices, especially when the data is available in the form of an adjacency matrix. Graph partitioning techniques can be used on the resulting graph. The partitioning mapping produced by the partitioner can be applied to partition the adjacency matrix. Two main applications that have driven the development of graph models for efficient partitioning are VLSI circuit design and parallel computation. In VLSI circuit design [16] common objectives are the minimisation of the wire length between the components in the circuit, as well as the optimisation of the intercommunication between the individual components and the minimisation of silicon layers in microchips. Similarly for parallel algorithms we try to minimise the total volume of communication between processors, while balancing various other properties between partitions to ensure for instance, that all processors are equally busy. Further application domains of graph partitioning are neural net simulation, particle simulation and data mining [2, 7, 9, 10, 18] just to name a few. All graph partitioning problems lie in NP [2, 4, 7, 9, 10, 16] some problems such as the optimal k-way hypergraph partitioning are even NP-complete [9]. All graph and hypergraph partitioning tools therefore use heuristics to find solutions that are close to the optimal partitioning.

k-way partitioning

Definition 2.14. We say that $\Pi = \{\Pi_1, \Pi_2, \ldots, \Pi_k\}$ is a *k-way partitioning* of the set of vertices V of a graph Γ s.t. $\Pi_i \cap \Pi_j = \varnothing,\ 0 < i < j \leq k$ and $\bigcup_{i=1}^{k} \Pi_i = V$.

2.8.1 Graph Models

As graph models are used in many different areas of research, several graph models have been developed over the course of time. The collection of graph representations presented in this section is by no means exhaustive, but it gives an overview of commonly used models. One important thing to note is that in most applications we use graphs to represent *sparse matrices*. A matrix is considered sparse if the vast majority of its entries is zero. The sparsity of the matrix is important for successful application of graph models in practice, as for large dense matrices the memory requirements of the sparse matrix representation becomes too high. Hence from now on we assume that all matrices mentioned in this report are sparse unless explicitly stated otherwise.

Sparse matrix

2.8.1.1 Standard (undirected) graphs

Definition 2.15. Let $\Gamma(V, E)$ be an *undirected graph* with vertex set V and the set of edges $E \subseteq V \times V$. To represent a $n \times n$ matrix A using a standard graph, we assign the rows to be the vertices of $\Gamma(V, E)$, i.e. $V = \{row_1, row_2, \ldots, row_n\}$. For every non-zero element a_{ij} in A the model has two edges $e_{ij} = (row_i, row_j) \in E$ and $e_{ji} = (row_j, row_i) \in E$.

A partitioning algorithm allocates each row to a certain partition Π_i while optimising certain objectives under given balance constraints for all partitions in Π. However, as Hendrickson notes in [3] this type of graph has some severe shortcomings as it can only be used to represent square matrices.

2.8.1.2 Bi-partite graphs

To overcome the limitations of the standard undirected graph model, Kolda and Hendrickson came up with a more expressive model, which uses a *bi-partite graph* to represent matrices [2].

Definition 2.16. Let $\Gamma(V, E)$ be a graph with vertex set $V = V_1 \cup V_2$, $V_1 \cap V_2 = \emptyset$ and the set of edges $E \subset V_1 \times V_2$. Assume A is a $m \times n$ matrix, and let $V_1 = \{row_1, \ldots, row_m\}$ and $V_2 = \{col_1, \ldots, col_n\}$. For each nonzero element a_{ij} in A we have a corresponding edge $e_{ij} = (row_i, col_j) \in E$.

Despite the fact that the bi-partite model overcomes the limitations mentioned in defn. 2.15, the model was superceeded by the hypergraph model, which gives a far more intuitive way of calculating the total communication volume[2, 4], which is an important metric for graph partitioning algorithms (see sect. 2.8.2.2).

2.8.1.3 Hypergraphs

Hypergraph

Recent graph partitioning tools use *hypergraph* representations for the underlying data. This has two major reasons. The first one is that hypergraphs are much more flexible than other types of graphs and can therefore be applied to a vast range of problems. Secondly in [4] U. Catalyurek, C. Aykanat show an intuitive relationship between hyperedge cuts and the total communication volume (see sect. 2.8.2) of a partitioning. The hyperedge cut is equivalent to the boundary cut metric introduced in sect. 2.8.2.2. In experiments they show that partitioners which use the hyperedge-cut metric produce far better partitionings than partitioners that deploy the edge-cut (sect. 2.8.2.1) metric.

Definition 2.17. A hypergraph $\Psi(V, H)$ has a vertex set V and a hyperedge set $H \subset P(V)$, a subset of the powerset of V.

In literature hyperedges are sometimes referred to as nets and vertices spanned by a hyperedge as pins. This has historical reasons because many of the early applications of hypergraphs were in the field of VLSI circuit partitioning. Various 1D and 2D hypergraph representations have been developed for different types of matrices to create tailored representations for different problems [4, 6, 7, 10, 11, 12, 18, 19]. In 1D row-wise hypergraph partitioning rows of the matrix become the vertices and each column is represented by a hypernet. A vertex lies in the hypernet of a column if its corresponding row has a non-zero entry in that column. In 1D column-wise hypergraph partitioning the roles of rows and columns are swapped. In 2D hypergraph partitioning every non-zero element in the matrix becomes a vertex and both rows and columns are interpreted as hypernets. In most applications 1D hypergraphs are preferred to 2D representations although 2D hypergraphs allow more fine-grained partitioning. This is due to the fact that 2D representations require more memory and are also more computationally expensive to partition.

2.8.2 Partitioning metrics

It is computationally infeasible to search for optimal hypergraph partitionings as this problem is NP-complete. The function and the quality of heuristics used to find good approximations to the optimal partitioning vary between tools and applications domains. Therefore we will only introduce the two most commonly used metrics, which approximate/represent the *total volume of communication*.

Total volume of communication

Example. When partitioning matrices for parallel computation of a matrix vector product $Ab = v$, we need to distribute the data elements of A, b and v between the processors. Assume processor p_1 needs to compute row_i of A. If it has all necessary elements of b and element v_i allocated then it can compute v_i without any extra communication. If, however, element a_{ij} is non-zero and b_j is allocated to processor p_2, then p_2 has to send the value of b_j to p_1 before p_1 can calculate v_i. This exchange is called pre-communication. Similarly we might need post-communication when we divide our matrix into columns, or even pre- and post-communication in case we have a 2-dimensional graph partitioning [4, 6, 10, 14]. The total amount of communication in this case is the amount of vector elements that need to be exchanged between processors during pre- and post-communication of each matrix vector multiplication.

2.8.2.1 The edge-cut metric

Edge-cut

In a k-way graph partitioning $\Pi = \{\Pi_1, \Pi_2, \ldots, \Pi_k\}$ the *edge-cut* metric represents the number of edges whose vertices lie in two different partitions. Formally that is

$$|\{(v_i, v_j) \mid (v_i, v_j) \in E \wedge v_i \in \Pi_r \wedge v_j \notin \Pi_r\}|$$

There exist many variations of this metric, some algorithms for instance accumulate the weights of edges that cut(i.e. cross) partition boundaries rather than counting the number of edges that cut partitions[3]. The edge-cut metric is used in many standard graph partitioning tools such as MeTiS and Chaco[2], but it came under scrutiny when Hendrickson pointed out in [3] that it was flawed since it only approximates the total communication volume. Whilst the edge-cut metric gives good approximations for matrices representing certain differential equation problems, it is less accurate for matrices originating from other problems.

Example. To illustrate why the edge cut metric does not represent the exact volume of communication let us assume the following case. Imagine a parallel sparse matrix vector multiplication algorithm as described above. Assume we have allocated b_i to processor p_1 and processor p_2 needs b_i to calculate row_f and row_g. The edge-cut metric will be 2 as we have two edges cutting the boundaries, however the true communication volume is actually 1, as we only have to transfer b_i once.

The example shows that the accuracy of the edge-cut metric heavily depends on the structure of the underlying matrix, which is not ideal as it restricts the use of the metric to specific types of problems.

2.8.2.2 The boundary-cut or hyperedge-cut metric

Boundary-cut

The *boundary-cut* metric measures the total communication volume of a k-way partitioning Π exactly. Optimising this metric is hard [2, 3, 4, 10, 11, 12, 14, 16, 17], especially as we might also need to

- balance the amount of communication between partitions, to avoid heavy communication loads on some partitions.
- take into account that latency costs for setting up an initial communication channel between partitions (e.g. networks, processors, FPGAs, etc.) are often more expensive than transferring larger volumes.
- balance the size of partitions

As it is an exact measure of the total volume of communication the boundary-cut metric has become the standard metric for all algorithms that seek to minimise the partition intercommunication. Calculating the boundary-cut metric in hypergraphs is straightforward [4]. All we need to do is accumulate the number of cuts for every hyperedge in the hypergraph

$$\sum_{h \in H} \lambda(h) - 1$$

where $\lambda(h)$ is the number of partitions that hyperedge h connects. This formula is also referred to as the *hyperedge-cut* metric. *Hyperedge-cut*

Note. Hypergraph partitioning tools such as PaToH [4], hMetis [18] and Parkway[14] offer a vast number of configurations, so that users can optimise the partitioner for specific types of matrices.

2.8.3 Recursive bi-partitioning vs. k-way partitioning

Recursive bi-partitioning algorithms split a graph multiple times. Starting on the *flat*, i.e. entire graph, they first create 2 partitions which are further divided into $4, 8, 16, \ldots$ partitions. Recursive bi-partitioning is a greedy algorithm, i.e. once two partitions have been split the algorithm cannot move vertices between them in later stages of the recursion. A k-way partitioner on the other hand divides a graph into k partitions and consequently moves vertices between all k partitions until no further improvement can be achieved. In [7] Trifunovic and Knottenbelt show that k-way partitioning algorithms can create better partitionings for large k than recursive bi-partitioning algorithms. On the other hand recursive bi-partitioning tends to be faster than k-way partitioning as k-way partitioners need to check more moves when doing iterative improvement on the partitioning. *Flat graph*

2.8.4 Objective functions

Balance constraints and optimisation objectives are needed by hypergraph partitioners to compute *gain* and balance values for changes made during the iterative refinement phase of the graph partitioning process. These objectives vary depending on the application of the hypergraph partitioner. A typical balance constraint is the the weight of partitions, i.e. the computational load of a partition. This ensures for instance that processors in a parallel cluster need to perform a similar amount of computation. Optimisation constraints, for example, are the minimisation of the total communication volume and the minimisation of the maximum communication volume per partition. In practice algorithms often use two or more objectives (i.e. multi-constraint partitioning) to produce better graph partitionings. *Gain*

2.8.5 Flat vs. Multilevel hypergraph partitioning

There are different paradigms when it comes to hypergraph partitioning. The most intuitive one is the flat partitioning approach which creates a partition by analysing the entire graph without preprocessing. Usually these algorithms start building an initial partitioning around randomly chosen vertices. Subsequently variations of Kernighan-Lin(KL) [16] and Fiduccia-Mattheyses(FM)[13] iteratively refine the initial partitioning by moving vertices between the partitions. The downside of flat partitioning algorithms is that their performance and the quality of their solution decreases rapidly as the problem size increases. Because of these shortcomings, modern hypergraph partitioning tools such as hMeTiS, Parkway and PaToH implement the multilevel approach which gives better partitionings in less time for large graphs due to the graph coarsening phase [7, 12, 18, 19].

2.8.6 Multilevel hypergraph partitioning

The multilevel approach involves the following three consecutive phases

- Coarsening (clustering) phase
- Initial partitioning of the coarsened graph
- An uncoarsening and iterative refinement phase

2.8.6.1 Coarsening phase

The aim of the coarsening phase is to produce a compact version of the graph that has a topology similar to the one of the original graph. The more the coarsenend graph resembles the initial graph the better the initial partitioning will be. Most hypergraph clustering algorithms create a series of successively coarser graphs $\{\Psi(V,H), \Psi(V_1,H_1), \ldots, \Psi(V_{coarse}, H_{coarse})\}$ until a minimal threshold for the number of vertices in the coarsened graph has been reached. There are many techniques for efficient hypergraph coarsening such as Heavy Connectivity Matching (HCC), Heavy Connectivity Clustering (HCC) [4], edge-coarsening (EC) [18] and first choice (FC) [12].

The EC algorithm for instance works as follows. At the beginning of level i of the coarsening phase all vertices of the hypergraph $\Psi(V_i, H_i)$ are unmarked. A random vertex v_r is chosen and clustered with the unmarked adjacent vertex v_s for which the $gain(v_r, v_s)$ is highest among all unmarked vertices adjacent to v_r. The *gain* function gives a heuristic that can be used to decide whether two vertices $v_r, v_s \in h$ are a good match (see [4, 12, 18] for examples of gain functions). A new cluster vertex is then formed and marked so that it cannot merge with any other vertex at level i of the coarsening phase. The sets of hyperegdes describing the in-flux and out-flux of the two vertices are joined, too. All references to v_r and v_s in existing hyperedges are updated to point to the newly formed cluster vertex. Singleton hyperedges are dropped altogether. If v_r has no suitable neighbour to cluster with, it becomes a marked singleton cluster. Once there doesn't exist any further unmarked vertex that has an unmarked neighbour, a new level starts and all vertices become unmarked again. The process ends when the graph has been coarsened to a predefined number of vertices.

The FC clustering is similar to the EC algorithm. The difference being that FC allows unmarked vertices to merge with marked clusters of vertices. This requires some extra control at each coarsening level to ensure that the amount of vertices reduces by a fixed ratio at each level. Additionally the *gain* function has to penalise large clusters to prevent polarisation towards particularly large clusters. In either method the mappings from each coarsening level to the next have to be stored in memory to allow uncoarsening later.

2.8.6.2 Initial partitioning phase

The initial partitioning is usually computed using standard flat graph partitioning. However, many of the flat partitioners choose their seeds for the partitions in a non-deterministic manner. Thus running the algorithm multiple times on the initial unpartitioned coarsened graph results in partitionings that vary in quality. Bad choices at this level can lower the quality of later partitionings significantly as a bad initial partitioning is propagated to later stages of the uncoarsening phase, where the algorithm does fine grained improvements only. In [18] Karypis et al. suggest a way to avoid this problem. In their implementation of hMeTiS they create various initial partitionings, which are uncoarsened concurrently. At each refinement level they then keep all partitionings that have cut sizes within 10% of the best partitioning at that level. This technique has been shown to improve the quality of the partitionings at the cost of a small computational overhead, as the number of alternative partitionings is only high when the graph is coarse and decreases as the partitionings are uncoarsened since many partitionings are filtered out by the 10% cut size requirement.

2.8.6.3 Uncoarsening and iterative refinement phase

Once an initial partitioning has been calculated, variations of the KL or the FM algorithms are used to refine the initial partitioning. In the case where we create multiple initial partitionings we have to uncoarsen and refine each of them. The KL or the FM algorithm then optimises partitionings based on given optimisation objectives and balance constraints. Iterative refinement is run at each level of the uncoarsening phase. The iterative refinement algorithm stops as soon as it converges, i.e. when no legal vertex move brings any more gain. The graph is then uncoarsened to the next finer level. The algorithm ends as soon as the iterative refinement algorithm converges on the partitioning of the initial flat graph. As iterative refinement in multilevel partitioning algorithms is initially performed on coarse graphs and gradually moves more fine grained clusters as the graph is uncoarsened, the multilevel approach is less likely to be trapped in a local minima/maxima, which easily happens to flat hypergraph partitioners. By using the multilevel paradigms for hypergraph partitioners we thus get the hill-climbing feature for free [18, 19].

2.8.6.4 Multiphase refinement with restricted coarsening

A possible add-on is the multiphase refinement technique. This technique takes the initial partitioning and repeats the multilevel k-way hypergraph partitioning algorithm. The difference of the second partitioning run is that the coarsening algorithm only allows clustering of vertices that lie in the same partition. More information on the multiphase refinement can be found in [18].

CHAPTER 3

Partitioning the SMP state space

In [1] exact state-by-state aggregation is performed on the unpartitioned state space of SMP transition graphs. In this chapter we introduce different graph partitioning techniques and evaluate the effect of aggregating entire partitions of states with respect to the fill-in of the SMP transition matrix caused by the aggregation. In particular we compare the number of transitions in the SMP transition matrix before and after the aggregation of each partition. From now on we refer to these observations as *partitionwise observations*. Additionally we compare different partition sorting methods, which determine an order in which individual partitions are aggregated. Our main aim is to find partitionings that are suitable for aggregation techniques that help us to perform faster passage time calculation.

Partitionwise observations

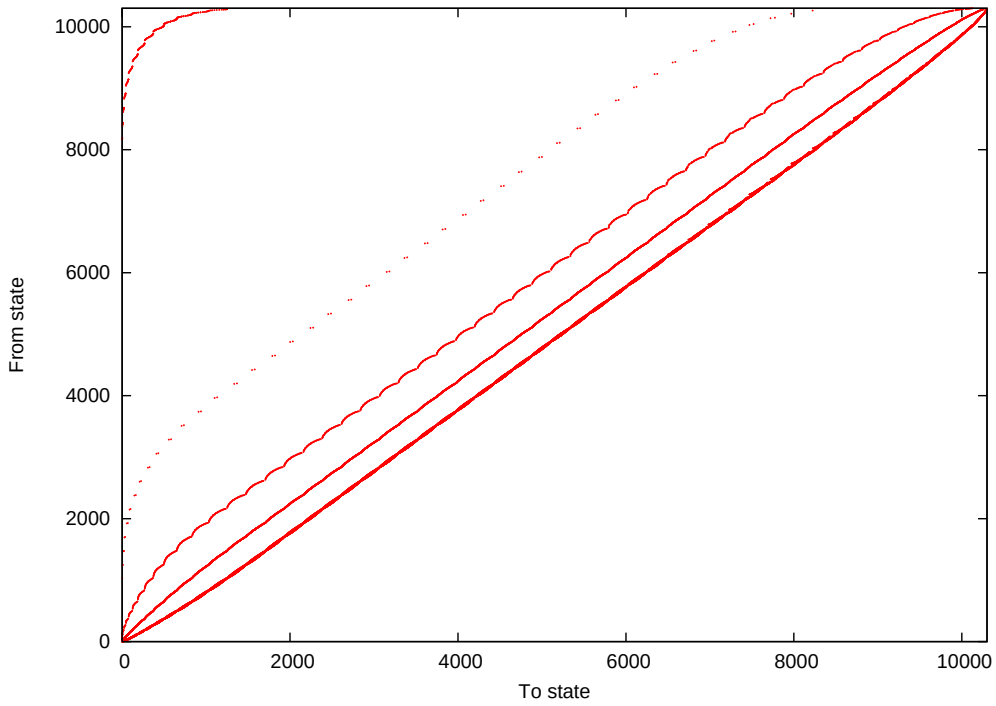


Figure 3.1: Unpartitioned transition matrix of voting model with 10300 states.

3.1 SMP transition matrix partitioners

The following graph partitioners are used to divide the state space into partitions of states. When doing passage time calculations we cannot aggregate source and target states as this might affect the result of the passage time calculation. We thus assume that we only partition *intermediate states*, i.e. reachable states in the SMP transition graph that are neither source nor target states. We further assume that we can divide n intermediate states into k partitions, such that $k|n$ and that the state space is enumerated starting with state 0.

Intermediate state

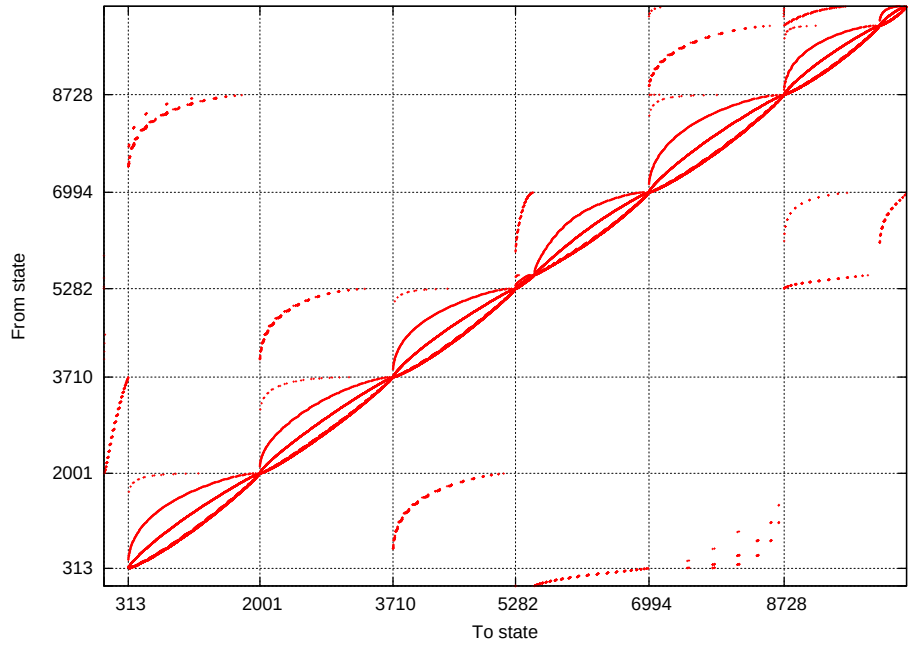


Figure 3.2: PaToH2D 6-way partitioned transition matrix of voting model with 10300 states. Non-zero elements in rows denote outgoing transitions from states. Non-zero elements in diagonal blocks represent partition internal transitions. Note that the state numbering is a permutation of the numbering in fig. 3.1.

3.1.1 Row striping

Definition 3.1. *Row striping* is a simple partitioning technique which splits the n rows of the transitions matrix into k partitions, each containing n/k elements. In terms of partitioning the state space of a semi-Markov model this implies that the first of the k partitions contains the first n/k intermediate states, that is the n/k intermediate states with the lowest indices. The second partition consequently contains the next n/k intermediates states and so on. The k^{th} partition contains the n/k intermediate states with the highest indices.

Row striping partitioner

3.1.2 Graph partitioner

Definition 3.2. *Graph partitioning* is a method which partitions an undirected graph. Since SMP graphs are usually directed graphs we have to introduce the notion of a weight for each transition in order to express connectivity more accurately. To represent the SMP transition matrix as a graph for the purpose of graph partitioning we represent each state as a vertex. There is an edge between two vertices v_i, v_j if their underlying states are connected in the SMP transition graph, i.e. if row i in the transition matrix has a non-zero element a_{ij} in column j or row j has a non-zero element a_{ji} in column i. If both states can reach each other in a 1-step

Graph partitioner

transition, i.e. if both a_{ij} and a_{ji} are non-zero, then this edge has weight 2, otherwise it has weight 1.

Graph partitioners are optimised for partitioning sparse matrices for parallel matrix vector multiplication. They try to minimise the edge-cut metric (see sect. 2.8.2.1) while balancing the number of non-zero elements in each partition. We use the MeTiS library [24], a sequential k-way graph partitioning utility library, for our implementation of the graph partitioner. Unfortunately MeTiS does not support directed graphs, which is why we represent the SMP as an undirected graph in this case. To keep the computational overhead of the partitioning low, we weight all edges with 1. Note that this uniweight approach potentially produces worse partitioning results than the edge weighting approach described in the graph partitioning definition.

3.1.3 Hypergraph partitioner

Hypergraph partitioner

Definition 3.3. A *hypergraph partitioner* partitions hypergraphs using the multilevel approach discussed in sect. 2.8.1.3. To use hypergraph partitioners on the SMP transition matrix we first translate the underlying directed graph into a hypergraph. As for the graph partitioner we define the states of the SMP to be the vertices of the hypergraph. We distinguish between 1D hypergraph partitioning where the hypernets either represent the successor states of each state (rows) or the predecessor states of each state (columns) and the 2D approach, where we use both successor and predecessor hypernets. Note that our definition of 2D hypergraph partitioning differs slightly from the definition commonly found in literature, where each non-zero matrix element becomes a vertex in the 2D hypergraph. In our case 2D simply implies that we use information from both rows and columns of the SMP transition matrix to construct hypernets.

Likewise graph partitioners hypergraph partitioners are optimised for parallel sparse matrix vector multiplication problems. In contrast to graph partitioners, hypergraph partitioners minimise the boundary-cut metric (see sect. 2.8.2.2). The different hypergraph partitioning methods used in our experiments are based on the PaToH library [25].

3.1.4 Next-Best-State-Search (NBSS) partitioner

Next-Best-State-Search (NBSS) partitioner

Definition 3.4. The *Next-Best-State-Search (NBSS) partitioner* attempts to create partitions by naturally extending a partition from an initial seed state. Starting from a random intermediate state the NBSS partitioner adds all successor states of that particular state into a priority queue. The states in the queue are in increasing order with respect to the number of extra successor states they would introduce if they were added to the partition. To determine this value we have to keep track of the successor states of the partition as well as the partition internal states. Every time a state is added to the partition we have to add all its successor states that are not partition internal states to the list of successor states of the partition and also add them to priority queue of states. The priority queue then has to be reordered. Consequently we add the next best state to the partition. This is done until the partition has exceeded a predefined number of successor states. Note that although only intermediate states can be added to the partition, it is possible that some of the predecessor and successor states of the partition are source and target states.

This partitioning method aims at generating a partition that is well-suited for aggregation techniques described in chapter 5. Although it is possible to partition the entire state space using this technique, we only use it to find a single partition. We thus do not compare it to the other partitioners in this chapter.

3.2 Aggregation of partitions

Ideally aggregating a partition results in a transition matrix with fewer transitions. In the experiments conducted for the discussion in this section, we aggregated all k partitions, such that only the source and target states of the SMP remained. When applying aggregation

algorithms in practice this might not be the best approach as the computational costs as well as the memory costs for complete aggregation can be very high. In sect. 3.2.2 we introduce techniques that allow us to predict when it is best to stop aggregation.

3.2.1 Partition sorting strategies

Having partitioned the state space we have to decide an order in which to aggregate the partitions. We compare three methods with respect to the partitionwise number of transitions in the transition matrix that each sorting method produces for a given partitioning.

3.2.1.1 Fewest-paths-first (FPF) sort

Fewest-Paths-First (FPF) sort

FPF sort has been inspired by the fewest-paths-first state aggregation technique described in [1]. To choose a partition for aggregation using FPF sort we simply calculate the FPF-value of all available partitions and choose the one with the lowest FPF-value. Suppose a partition has m predecessor states, i.e. states that lie outside the partition but have outgoing transitions to states in the partition and n successor states, i.e. states that lie outside the partition and have incoming transitions from states in the partition. The number of transitions from the predecessor to the successor states in the SMP transition matrix after the aggregation of the partition is mn if all m predecessor states can reach all n successor states via paths through the partition. In this case we say that the partition is *fully connected*. The FPF-value of the partition is:

Fully connected

$$mn - outgoing\ transitions$$

where *outgoing transitions* is the total number of outgoing transitions from states in the partition. FPF sort is very fast as the calculation of mn and *outgoing transitions* is inexpensive, provided the transition matrix is represented as a sparse row-matrix.

3.2.1.2 Enhanced-fewest-paths-first sort

Despite a being a good estimator for the total number of new transitions created after the aggregation of a partition, the FPF-value does not take into account the number of *incoming transitions* from the predecessor states of the partition. Further it does not count the *existing transitions* between the predecessor and successor states of the partition. The total number of new transitions after the aggregation can thus be estimated more accurately using *enhanced-fewest-paths-first (EFPF) sort*. The EFPF-value is:

Enhanced-Fewest-Paths-First (EFPF) sort

$$mn - outgoing\ transitions - incoming\ transitions - existing\ transitions$$

Even though it is more expensive to calculate, our experiments show that EFPF sort usually gives better results than FPF or choosing the partitions in a random order. Figure 3.3 shows a situation where EFPF sort produces better results than FPF and Random sort.

The EFPF-value of a partition is only an upper bound for the total number of new transitions in the transition matrix after the aggregation of a partition. This is because there may not be a path from every predecessor state to every successor states with all intermediate states of the path being partition internal states. Even for small values of m and n this may cause significant differences between the estimated and the actual number of partitionwise transitions. The only way to determine the exact number of transitions in the transition matrix after the aggregation of a partition is to do reachability check for each pair of predecessor and successor states, which is a rather expensive calculation. We discuss this matter further in sect. 3.2.2.

3.2.1.3 Non-greedy sorting techniques

Both FPF and EFPF sort are greedy algorithms. It is therefore worth considering a *Look-Ahead-N-Steps* approach, which takes into account the effect on the remaining partitions when aggregating a particular partition. This is important if we want to aggregate more than one

Look-Ahead-N-Steps

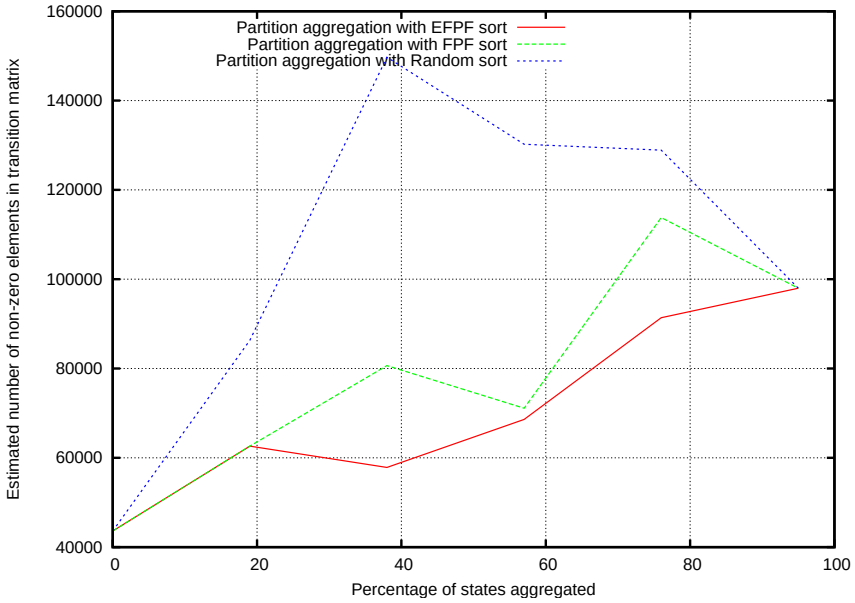


Figure 3.3: Comparing EFPF sort with FPF sort on a 5-way partitioning of the 10300 states voting model. The aggregation was done using the transition matrix predictor and is thus not exact. Clearly both EFPF sort and FPF sort do better than Random sort in this case. In fact in none of our experiments FPF sort or Random sort outperformed EFPF sort.

partition. The problem with a Look-Ahead-N-Steps approach is that the aggregation of a partition takes a considerable amount of time, even if done with a fast method such as the transition matrix predictor (see sect. 3.2.2). Thus Look-Ahead-N-Steps is only feasible for a small number of partitions, which implies that this sorting technique restricts the freedom of our partitioning. Therefore we do not further investigate it.

3.2.2 Transition matrix predictor

In most practical cases we do not want to aggregate all k partitions, hence we need a means to decide when to stop the aggregation process. The fastest way to assess a given partitioning is a transition matrix predictor. In essence this is just another atomic partition aggregator (see chapter 5), with the difference that it only connects the m predecessor states with the n successor states using dummy transitions and discards all partition internal states along with their incoming and outgoing transitions. Recall that in practice it is possible that a partition is not fully connected. Thus the transition matrix predictor only gives an upper bound of the partitionwise number of transitions. Figure 3.4 compares the predicted number of transitions with the exact number of partitionwise transitions, which we obtained by doing exact state aggregation on the same partitioning.

3.2.3 Quality of partitionings

Our previous examples illustrate the benefit of the EFPF partition sorting method compared to other partition sorting methods. Furthermore we have the choice between using an estimator or exact state aggregation in order to determine the number of transitions in the transition matrix after the aggregation of a partition. In the following discussion we investigate how the choice of the partitioner affects the partitionwise number of non-zeros in the transition matrix. Here we do not assess the quality of the partitionings produced by the partitioners in terms of their suitability for exact state (see chapter 4) or atomic partition (see chapter 5) aggregation. These later chapters discuss which partitioners produce the best partitionings for state-by-state

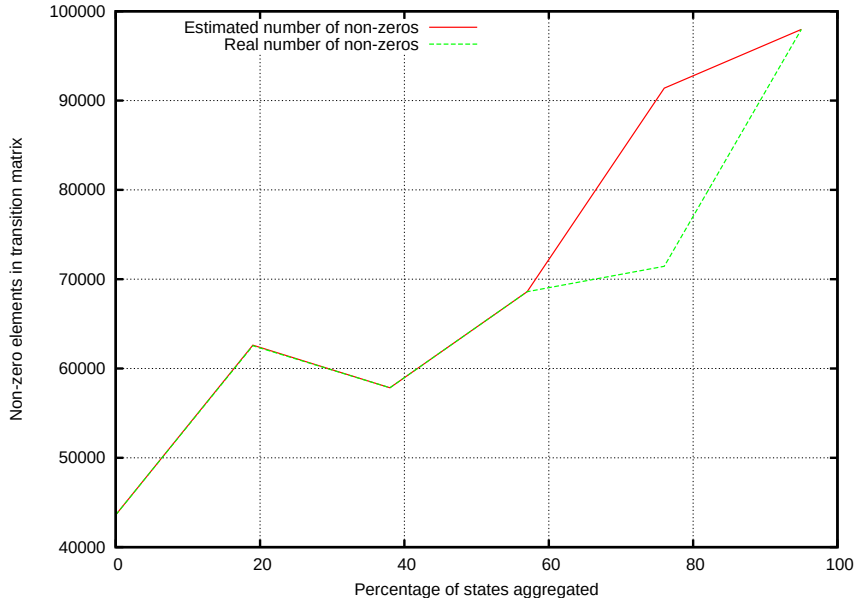


Figure 3.4: Comparing the estimated number of transitions with the real number of transitions after the aggregation of each partition. Both aggregators use the same 5-way partitioning of the 10300 states voting model as in fig. 3.3. The partitions were sorted using EFPF sort. Surprisingly there are many points where the predictor matches the exact value of transitions in the transition matrix. This implies that those partitions are fully connected. This behaviour was observed in various experiments. In some cases the estimator even oscillates between matching the real number of transitions and giving too large estimates.

and atomic aggregation of partitions respectively. It is important to make this distinction when thinking about the quality of a partitioning, as merely keeping the number of transitions as low as possible may not yield the best partitionings for some aggregation techniques. Nevertheless it is crucial for good partitionings that they can be aggregated in a way that keeps the total number of transitions in the SMP model low, as the final passage time calculation requires more computation if the transition matrix becomes dense. Moreover we cannot afford an explosion in the number of transitions as we only have a limited amount of physical main memory available.

The diagrams in fig. 3.6 show the quality of different partitionings for different models and partitioners. For the tests we used an Intel P4 with 3 Ghz and 4 Gbyte of RAM. As it is not feasible to perform state-by-state aggregation on large models, all aggregations were done using the transition matrix predictor. Having studied many graphs such as the ones in fig. 3.6 we conclude that PaToH(1D), which only uses the rows of the matrix as hypernets for partitioning, produces the worst partitionings out of all partitioners we tested. In the smcourier model (see fig. 3.6(b)) the partitioner yields the highest matrix fill-in and in the larger voting and web-server models the partitionings produced by PaToH(1D) either took too long to aggregate or caused the test machine to run out of memory. The naïve row striping yielded good results in the web-server and smcourier model, but in the slightly more dense voting model it performed a lot worse than MeTiS and PaToH2D. MeTiS produces the most stable results of all partitioners. This and the fact that MeTiS is a deterministic partitioner makes it the best partitioner for the purpose of keeping the partitionwise number of non-zeros in the matrix low. Introducing weights to our SMP graph might further improve the MeTiS partitioning. However, even though MeTiS fluctuates less than PaToH2D, which often creates poor partitionings for larger models, we need to point out that the best partitionings that we found for each model were always produced using PaToH2D. As PaToH2D is non-deterministic this obviously comes with the overhead of having to run the partitioner multiple times to find a suitable partitioning. For a single

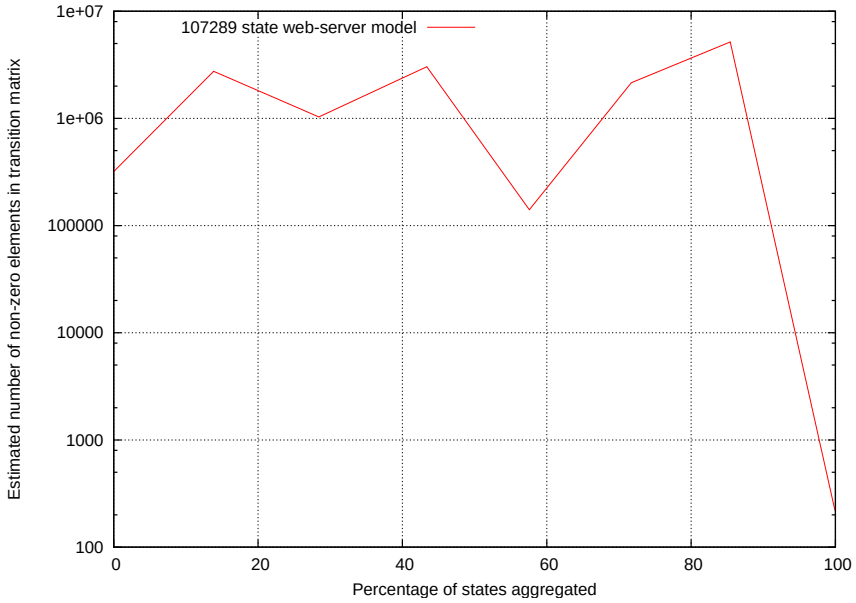


Figure 3.5: This graph shows the result of using the transition matrix predictor on a 7-way partitioning of the 107289 web-server model created with the PaToH2D partitioner. We used logscale for the y-axis to emphasize that about 60% of the state space can be aggregated while halving the original number of transitions in the SMP model. Given this partitioning, our predictor takes less than a minute to produce this estimation. The example shows that the predictor is also capable of producing valuable estimates in larger models. Unfortunately most of the partitionings produced by PaToH2D for this model were of far worse quality.

calculation it is therefore better to use MeTiS. If, however, a partitioning is reused multiple times then PaToH2D should also be considered. Another interesting observation is that the row striping method is the only partitioner that allows us to increase the number of partitions in the partitioning without significantly decreasing the quality of the partitioning. Whilst MeTiS and PaToH2D perform best on 5-10 partitions, row striping often yields better partitionings when using a larger number of partitions (see fig. 3.7) though its best partitionings are still much worse than the best MeTiS and PaToH2D partitionings. It should be noted that except for relaxing the restrictions on the partition size this study does not thoroughly investigate the effect of different setups for MeTiS and PaToH. Both partitioners offer a vast variety of configurations, which can potentially improve the partitioning.

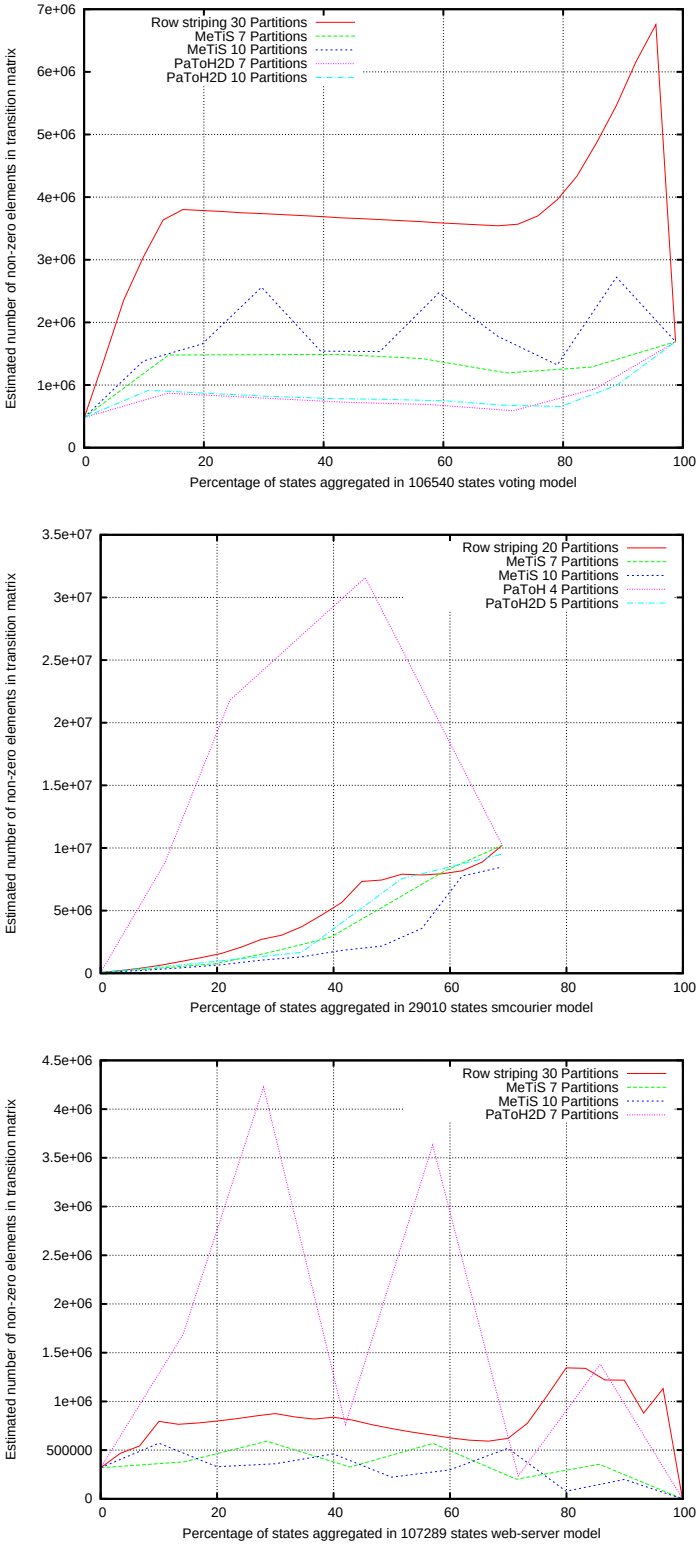


Figure 3.6: The diagrams show the predicted number of transitions in the transition matrix of different models, partitioned with the partitioners we introduced in this chapter.

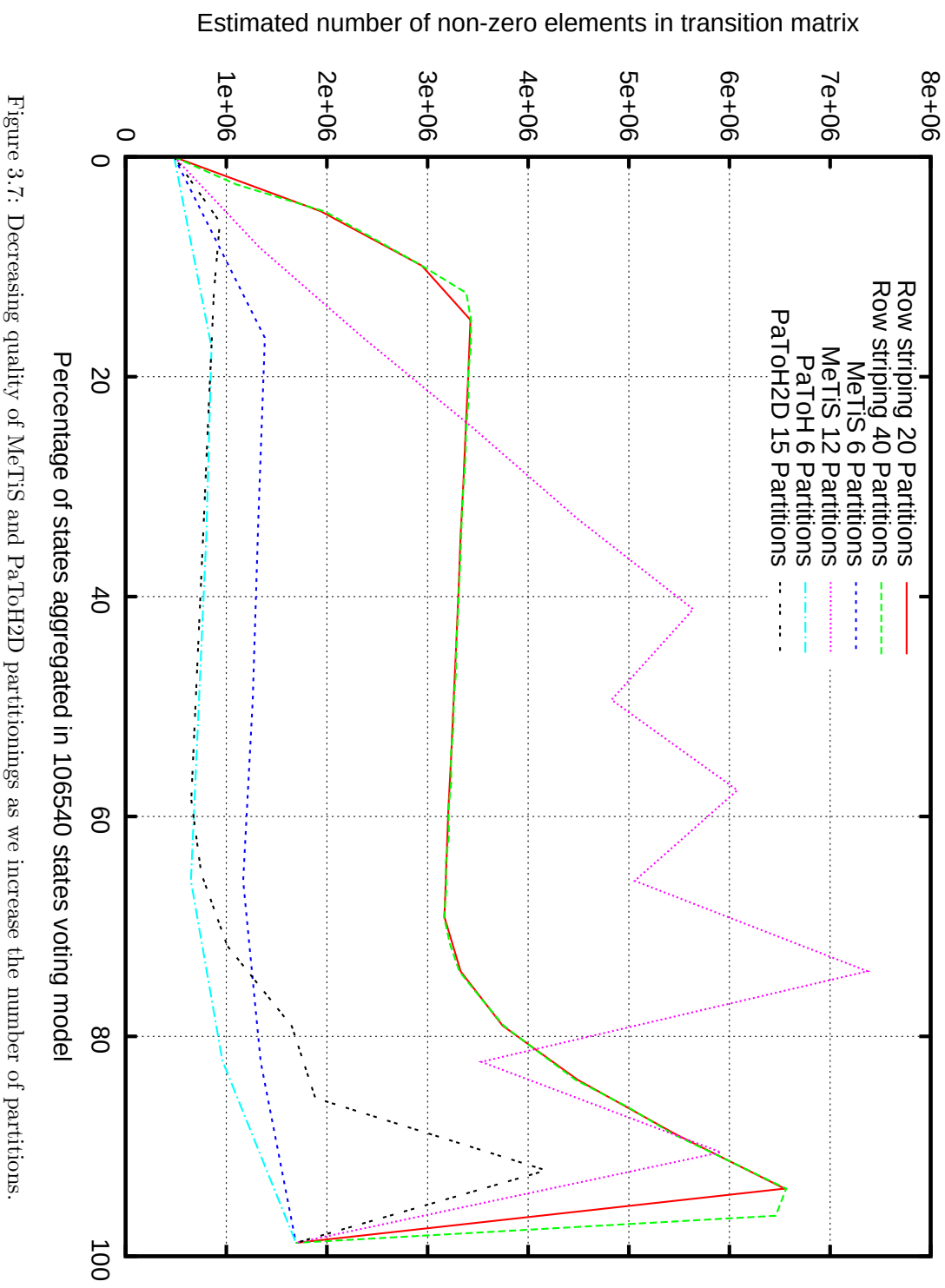


Figure 3.7: Decreasing quality of MeTiS and PaToH2D partitionings as we increase the number of partitions.

CHAPTER 4

State-by-state aggregation of partitions

In this chapter we discuss the application of the exact state-by-state aggregation technique described in sect. 2.7 to aggregate partitions of states. These partitions are generated using partitioners discussed in sect. 3.1. Our main focus in the following investigation lies on the *sub-matrix* fill-in during the state-by-state aggregation of a partition, where the sub-matrix is the part of the transition matrix that consists of the rows and columns of the partition's predecessor, internal and successor states only.

Sub-matrix

4.1 State aggregation techniques

The time and memory requirements of exact state aggregation vary hugely depending on the order in which states are aggregated. In [1] various state sorting techniques are introduced and tested. In this section we introduce a new state ordering method that performs better than previous techniques.

4.1.1 Fewest-Paths-First aggregation

Out of all exact state aggregation techniques discussed in [1] *Fewest-Paths-First(FPF) aggregation* is the one that causes the lowest matrix fill-in. In FPF the next state chosen for aggregation is the one with the lowest product mn, where m is the number of predecessor states and n the number of successor states. If more than one such state exists we choose the one with the lowest index. Intuitively this is a good approach since minimising the FPF-value should keep the number of newly created transitions low when aggregating a state. The downside of FPF though is that it does not take into account existing transitions between predecessor and successor states of the state that we are aggregating. Figure 4.1 illustrates this problem. Even though this difference is minor when the matrix is sparse, it is not hard to see that once the transition matrix becomes more dense, FPF aggregation no longer gives accurate predictions of how many new transitions the aggregation of a state will generate.

Fewest-Paths-First(FPF) aggregation

4.1.2 Exact-Fewest-Paths-First aggregation

To overcome the inaccuracy of the FPF metric, we introduce *Exact-Fewest-Paths-First(EFPF) aggregation*. Suppose a state s has m predecessors, n successors and $i \in \{0, 1\}$ self-loops. Moreover assume that there are t existing transitions between the successor and predecessor states, not including the transitions starting or ending in state s. The latter restriction is important as a state with a self-loop is its own predecessor and successor state. The EFPF-value of state s is $(m - i)(n - i) - m - n - t$. Note that we do not count self-loops, which are created when

Exact-Fewest-Paths-First(EFPF) aggregation

the set of predecessor states intersects with the set of successor states, as new transitions. This is because all these loops can be removed after each aggregation. Figure 4.2 gives an example where the EFPF aggregation technique outperforms the FPF technique.

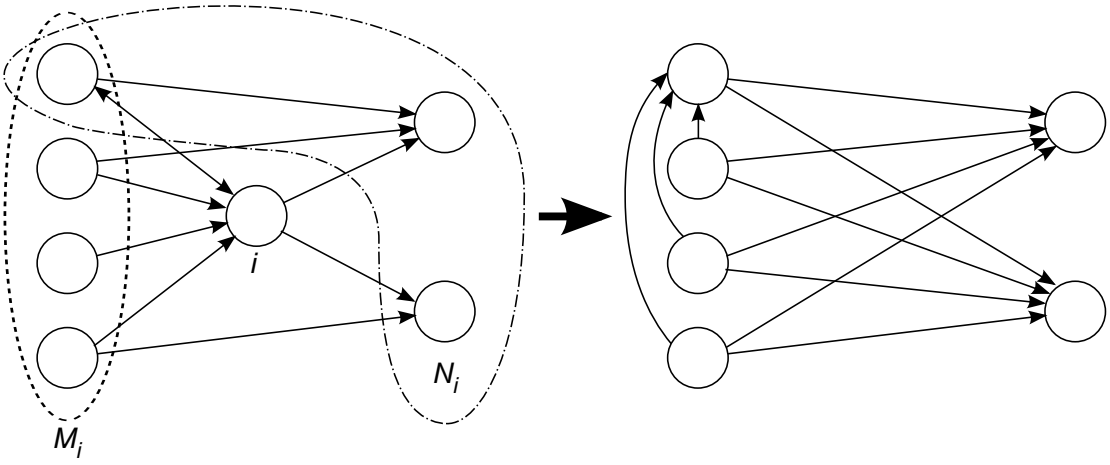
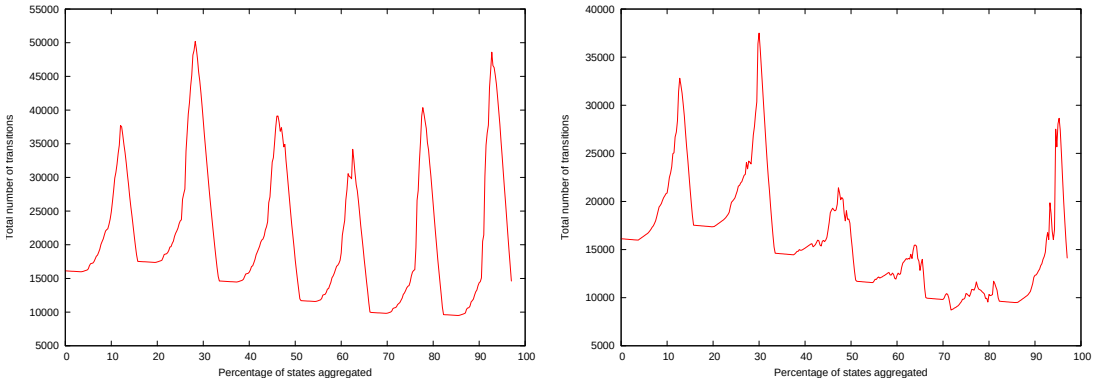


Figure 4.1: On the left hand side the we see the transition diagram of a SMP before the aggregation of state i. M_i is the set of predecessor states, N_i the set of successor states of state i. Note that M_i and N_i have a non-empty intersection in this example. The right hand side shows the transition diagram after state i has been aggregated. The FPF algorithm would calculate a cost of $4 \cdot 3 = 12$, whilst the actual number of newly created transitions is only $4 \cdot 3 - 4 - 3 - 4 = 1$. Note that the self cycle of the state that lies in M_i and N_i has been removed after the aggregation of state i. [27]



(a) State aggregation with FPF state sorting

(b) State aggregation with EFPF state sorting

Figure 4.2: Voting model with 4050 states and 6 partitions. Partitions were sorted using EFPF partition sort (see sect. 3.2.1.2).

4.2 Transition matrix fill-in during aggregation of partition

The following experiments where conducted using EFPF partition sorting (see sect. 3.2.1.2) and EFPF state sorting. To compare the quality of different partitionings of the state space, we compare both the transition matrix fill-in during the aggregation of the individual partitions as well as the partitionwise number of non-zero elements in the matrix. Especially the evaluation of the exact number of transitions during aggregation is of interest when performing state-by-state aggregation.

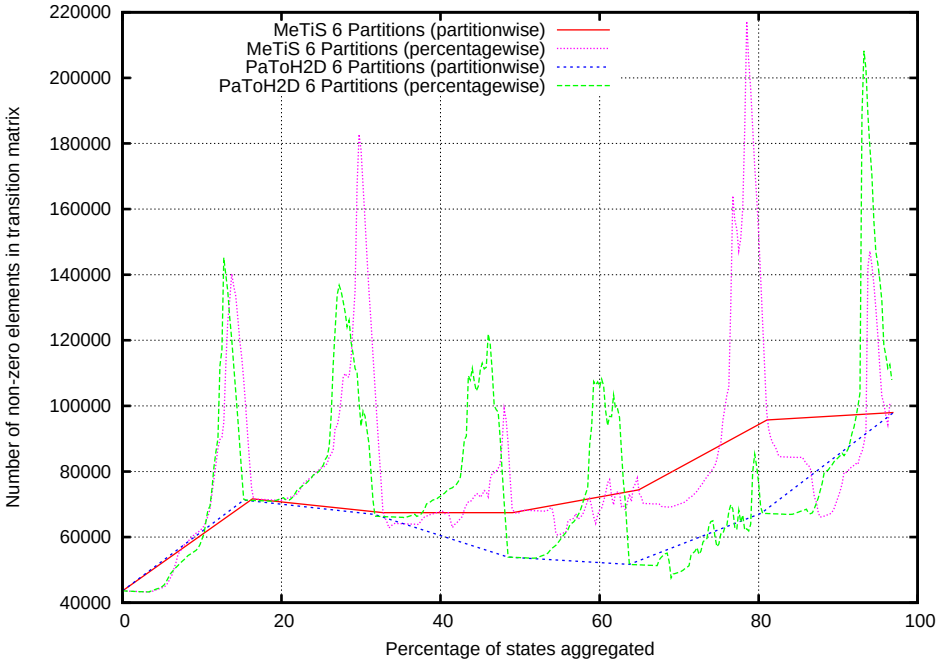


Figure 4.3: State-by-state partition aggregation on 10300 states voting model. Note that the sub-matrix of the partition fills in rather quickly during its aggregation causing peaks in the number of transitions that we need to hold in memory. In terms of the maximum number of transitions created during aggregation MeTiS does slightly worse than PaToH2D in this example. Note that percentagewise implies that we are taking continuous measurements of the number of transitions in the transition matrix.

Figure 4.4 illustrates the benefits of the partition aggregation approach. Instead of having a single global matrix density peak, each partition as a smaller local peak. This entails that the aggregation of the entire state space can be done using a lot less memory. Even though the example in fig. 4.4 proves that the state-by-state aggregation of partitions is more efficient than exact aggregation on the flat graph, there remains the problem of finding suitable partitionings for a given graph. Results in sect. 3.2.3 highlight that the quality of partitionings decrease, i.e. more transitions are generated upon aggregation, for partitionings from MeTiS and PaToH2D as we increase the number of partitions in the partitioning. This obviously limits the extend to which exact state aggregation can be used in practice, since we cannot partition large models into many small partitions without compromising the quality of the resulting partitions. Having small partition sizes is essential for state-by-state aggregation of partitions as this is the only way to keep the height of the local matrix density peaks low. The quality of partitionings produced by the row striping partitioner seems to be less affected by the increase in the number of partitions, but since the partitioner generally produces poor partitionings it cannot overcome this problem either. One way to solve this problem might be to use different partitioners for coarse- and fine-grained partitioning of the state space.

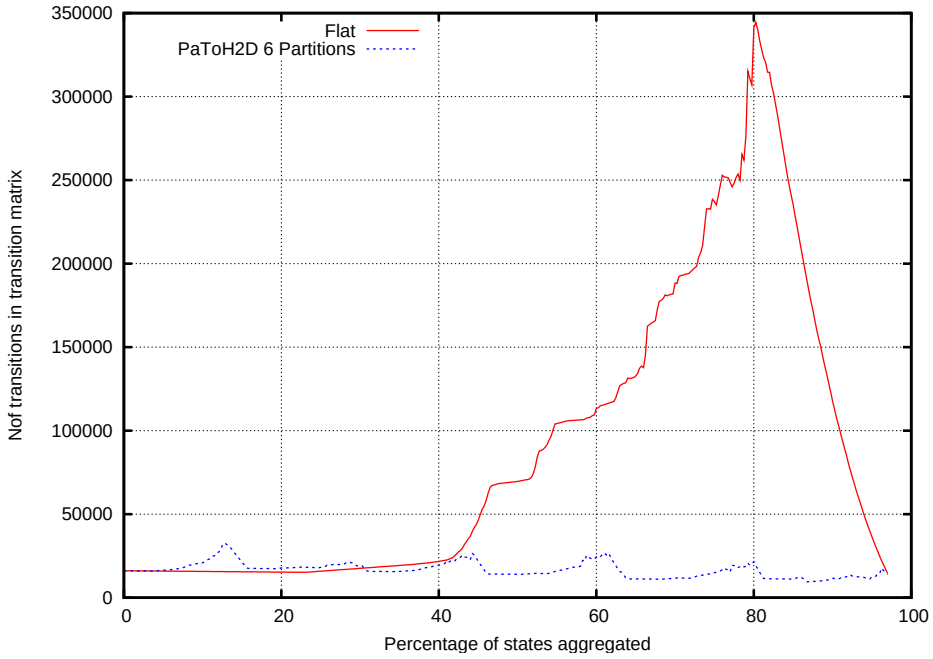


Figure 4.4: The above diagram illustrates the effect of partition aggregation compared to flat aggregation of the 4050 states voting model. The partition aggregation graph has many local density peaks caused by the local fill-in of the sub-matrices of the partitions during state-by-state aggregation. As expected, restricting the fill-in to the rows of the predecessor, successor and internal states of the partition that is being aggregated reduces the maximum number of transitions created during aggregation. As a consequence partition aggregation is a lot faster than flat state-by-state aggregation.

4.3 Partial aggregation of partitions

An alternative strategy for state space reduction is the partial aggregation of partitions, which only performs aggregation on a particular partition until a particular cost level is reached. One way of doing partial aggregation is to set a sub-matrix fill-in limit for each partition. This might of course cause us to stop aggregating states of a particular partition just before reaching a peak point, but since we cannot predict the exact height of the peaks there is no way to avoid this. If no peak point is overcome then the effect of partial aggregation of partitions is similar to aggregating states on the flat graph, the only difference being that the partitioning reduces the search space of the EFPF state sorter. Figure 4.5 shows the effect of aggregating all states that have an EFPF-value of 10 or less.

4.3.1 Cheap state aggregation

The results in fig. 4.5 inspired us to check whether it is possible to aggregate states and without increasing the number of transitions in the transition matrix. Note that this type of aggregation does not require state space partitioning per se, but it can be used in conjunction with aggregation techniques that do use state space partitioning. We refer to states that can be aggregated without increasing the number of transitions as *cheap states*.

Cheap states

Definition 4.1. *Cheap states* are states with EFPF-value ≤ 0 (see sect. 4.1.2).

As the calculation of the EFPF-values is expensive when applied to all states in the state space, it is sensible to examine whether there is another way of detecting cheap states in a SMP. Since the initial transition matrix is sparse it is reasonable to assume that the m predecessor states

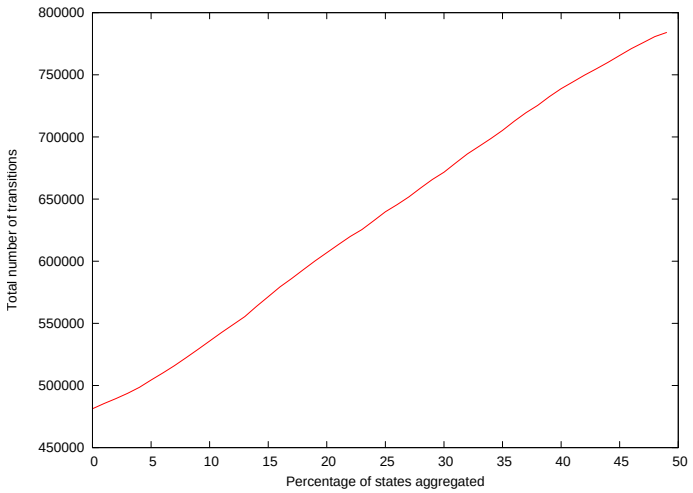


Figure 4.5: Partial aggregation on 106540 states voting model. In this example we can aggregate half of the state space at the expense of doubling the number of transitions.

of a particular state are generally not connected to the n successor states. Furthermore we can remove all self-loops before aggregating a particular state without creating any extra transitions. Under these assumptions cheap states are states such that $mn - m - n \leq 0$. This either forces $m = 2$ and $n = 2$ or $m = 1$ or $n = 1$. In practice aggregating a state s with $m = 2$ and $n = 2$ is not feasible. This is because of the case in which the successor states of state s happen to be cheap states, too. In this situation the successors states may no longer be cheap states after the aggregation of s as they potentially gain an extra predecessor state. Therefore we only concentrate on the case when $m = 1$ or $n = 1$. When implementing cheap states aggregation it is best to restrict cheap aggregation to all states that have $m = 1$ (alternatively to all states with $n = 1$). Figure 4.6 illustrates the problem that can occur when aggregating all states that have either $m = 1$ or $n = 1$. Limiting the search space to those states which have $m = 1$, for instance is advantageous since aggregating cheap states with $m = 1$ only, does not change the cheap state property of other cheap states with $m = 1$. In our implementation we aggregate all states with $m = 1$, since we are working with a row matrix, which makes it easier to find the successor states of a particular state. When doing cheap state aggregation in an implementation with a column matrix, aggregating all cheap states with $n = 1$ is preferable. The table below presents the number of non-source and non-target states that are cheap states with $m = 1$ in different SMP models. Note that in the 3 models we tested, most cheap states satisfied both $n = 1$ and $m = 1$.

	Cheap states in transition graph of model		
Number of states	Voting	Web-server	SMCourier
30000	-	-	42.82%
100000	19.94%	27.63%	-
250000	19.98%	27.63%	-
500000	19.95%	27.61%	-
1000000	-	27.60%	-
1100000	16.66%	-	-

Table 4.1: Percentage of cheap states in the state space.

The table clearly shows that in some models a significant proportion of the state space consists of cheap states. As these states are neither target nor source states, their aggregation potentially has a positive effect on the final passage time calculation since the exact aggregation of cheap states can be done much faster than general state aggregation. We can also save memory by

aggregating cheap states during the process of reading the transition matrix from a file. We investigate cheap state aggregation further in chapter 6.

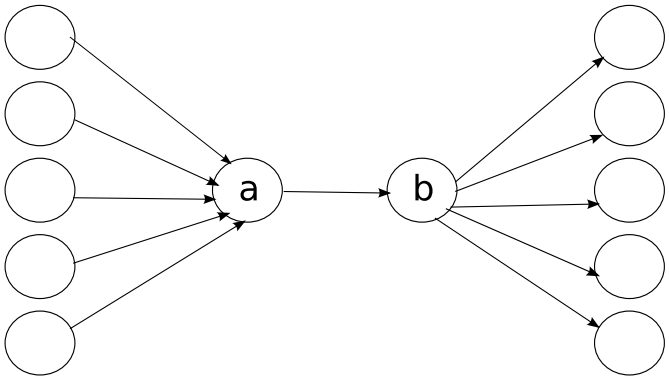


Figure 4.6: State a and state b are both cheap states by defn. 4.1. However, if we aggregate both of them we connect all predecessor states of a with all successor states of b, which obviously creates new transitions. To avoid this we would have to do extra checks on cheap states before aggregating them. Alternatively we can simply use the convention of only aggregating all cheap states with $m = 1$. That way we only have to do a single search for cheap states and the subsequent aggregation can be done without further checks.

4.4 Implementation of state-by-state aggregation

State-by-state aggregation requires regular updates of columns and rows in the transition matrix. For state aggregation to be efficient we need to ensure that we can quickly manipulate the transition matrix. On the other hand we also have to keep the memory requirements of the matrix low, as we potentially want to perform state aggregation on models with large state spaces.

4.4.1 Data structures

Although intuitively a sparse row and column matrix seems to be the best choice, there is one problem with this particular data structure. Whenever a state has been aggregated we need to delete all column and all row entries of that particular state. Since we are working with sparse data structures, which use set or map data containers, it is more expensive to update a sparse row and column matrix than updating a sparse row matrix or column matrix. Deleting a row in a 2D row and column matrix requires us to remove the row from the row matrix as well as all the entries of the row in the column matrix. Similarly when deleting a column we need to remove all entries in that column from all the rows in the row matrix. Therefore deleting a state in a 1D matrix only requires a single expensive deletion whereas in a 2D matrix it needs two. Furthermore a 1D matrix requires less memory than a 2D row and column matrix. Table 4.2 illustrates the access and manipulation costs for a 1D matrix using an array for storing the row/columns with rows or columns being map containers. In a row matrix a row contains the outgoing transitions of a state, in a column matrix a column contains the incoming transitions of a state. A 2D matrix contains a 1D row and a 1D column matrix. Since the operation of finding incoming or outgoing transitions of states in a 1D matrix can be made faster through the means of caching, we decided to use a 1D sparse row matrix instead of a 2D matrix for the representation of the sparse matrix in order to keep the memory demands of our implementation low.

We further experimented with balanced tree structures for storing the double values of the sojourn time distributions and the transition probabilities [22, 23]. In the end we decided not

Operation	1D row	1D column	2D row and column
Find outgoing transitions of state	$O(1)$	$O(n \log n)$	$O(1)$
Find incoming transitions of state	$O(n \log n)$	$O(1)$	$O(1)$
Add transition	$O(\log n)$	$O(\log n)$	$O(2 \log n)$
Delete transition	$O(\log n)$	$O(\log n)$	$O(2 \log n)$
Delete state	$O(n \log n)$	$O(n \log n)$	$O(2n \log n)$

Table 4.2: Comparison between time complexity of operations on sparse 1D and 2D matrices in a model with n states.

to use such trees for caching, since their structural overhead can diminish the saving in memory in some cases. This happens especially in transition systems of models that have marking dependent sojourn time distributions (c.f. defn. 2.7), which create a great variety of different Laplace transform samples and thereby limit the extend to which distribution information can be reused and shared between different transitions.

4.4.2 Validation

To validate our state aggregation algorithm, we did two first-passage time computations on the 4050 states voting model. In the first experiment we partitioned the intermediate states into 6 partitions, then aggregated two of them and subsequently did a first-passage time analysis with convergence precision of 10^{-16} to compute 198 Laplace transform samples for subsequent Euler Laplace inversion. The validation was done by performing the same passage time analysis on the unpartitioned graph. The results were identical up to an error term of 10^{-13} (see sect. 6.1.1 for further details on the error evaluation). Finally we checked that our results were similar to the results produced by discrete event simulation on the same model.

4.4.3 Performance

To assess the performance of state-by-state aggregation of partitions compared to state aggregation on the unpartitioned SMP graph we tested both memory and time requirements of the two algorithms for a voting model with 4050 states. EFPF aggregation of the unpartitioned SMP transition matrix took 244 seconds on an Intel P4 3.0 GHz processor with 4 Gbyte of RAM. In contrast to this the EFPF aggregation only took 11 seconds using a 6-way PaToH2D partitioning. Note that in either case we did not compute the Laplace transform points for each transition, so we would not have been able to use the resulting aggregated graph for any meaningful performance analysis. A first-passage time analysis with 693 Laplace transform samples takes 4 seconds on the original transition matrix using a precision of 10^{-8} for the convergence check of the iterative FPTA algorithm. This comparison shows that even though state-by-state aggregation of partitions performs a lot better than flat state aggregation it is still far too slow to speed up the actual passage time analysis.

4.5 Summary

Even though we are able to show that partition-by-partition aggregation speeds up the exact state aggregation introduced in sect. 2.7, state-by-state aggregation on partitions is still slower than doing the passage time calculation on the initial graph. The main reasons for this are the computationally expensive operations on the transition matrix as well as the EFPF-value calculation for individual states. It is therefore reasonable to conclude that exact state aggregation can only speed up SMP passage time analysis if the search cost for states that we want to aggregate is kept low and if changes made to the transition matrix during state aggregation are kept simple. One possible way of doing this is to limit state aggregation to cheap states. We investigate the performance of cheap state aggregation in chapter 6. Another way to speed up state aggregation would be to find new partitioning methods, which allow to create a higher

number of partitions while keeping the number of partitionwise transitions as low as MeTiS and PaToH2D partitionings with a small number of partitions do when being aggregated. Despite the fact there is a lot of potential for improvement, we doubt that state-by-state aggregation of state space partitions can actually speed up the computation of the passage time analysis in large SMP models. In the next chapter we therefore introduce ways of aggregating large partitions in one go.

CHAPTER 5

Atomic aggregation of entire partitions

Compared to flat state-by-state aggregation the partition-by-partition aggregation approach reduces the transition matrix fill-in drastically. However, there is still the problem that the partitionwise number of transitions is generally much lower than the maximum number of transitions during the aggregation of a partition (see fig. 4.3). Such density peaks are undesirable because it requires a significant amount of memory to store all temporary transitions. Additionally the fill-in slows down the aggregation of states as we need to convolve and branch more transitions when the sub-matrix of a partition becomes dense. This observation inspired us to investigate whether atomic aggregation of an entire partition can speed up the process of state aggregation.

5.1 Aggregation techniques

Atomic partition aggregation

In this section we introduce several techniques for *atomic aggregation of entire partitions*. Given the transition matrix and a partition of intermediate states, an atomic aggregation algorithm computes the structure of the graph as it would be after all states in the partition had been aggregated using exact state aggregation. This implies that we have to compute the new Laplace transform of the sojourn time distribution and the new probability for each transition from each of the predecessor states to each of the successor states of the partition that we are aggregating. Note that the calculation of the path probability is done implicitly by weighting the Laplace transforms of each transition by their conditional transition probability before convolving them (c.f. $r^*_{ik}(s)$ in eq. 2.7). Atomic partition aggregation requires two major steps. First we need to compute the transition from each predecessor state to every successor state by adding the weighted Laplace transforms of all convolved *partition transient paths*, i.e. paths of the form $p - i_1 - i_2 - \cdots - i_r - s$, where p is the predecessor state, s the successor state and i_k is a partition internal state. In a second step we add the Laplace transform of the transition to the existing one-step transition from p to s if such a transition exists. If it does not exist then the transition we computed in the first step becomes the new transition from p to s. We term this calculation a *restricted first-passage time analysis (RFPTA)*. RFPTA has the same computational complexity as the standard first-passage time computation (see sect. 2.6.1). The main difference between RFPTA and FPTA is that RFPTA is a FPTA on the sub-matrix of a partition excluding all direct transitions from the predecessor to states that do not lie in the partition that we are aggregating. In the following we discuss techniques for atomic partition aggregation. Note that the aggregators only describe ways to execute the first step. The final branching with existing one-step transitions from predecessor to successor states is the same for all aggregators.

Partition transient path

Restricted first-passage time analysis (RFPTA)

5.1.1 Restricted FPTA aggregator

Restricted FPT aggregator

Definition 5.1. Our first aggregator is based on the concept of restricted first-passage time analysis. A *restricted FPT aggregator* is an aggregator which computes the new transitions from predecessor to successor states by creating a single new transition for every pair of predecessor and successor states, which encapsulates the information of all partition transient paths between the two states. Technically this aggregation is done by performing a first-passage time analysis using the predecessor states of a partition as its start states and the successor states as its target states. The successor states, however, become absorbing states for the purpose of this calculation. The actual restriction refers to the outgoing transitions of the predecessor states. The RFPTA aggregator only considers outgoing transitions from predecessor states of the partition to partition internal states. All other outgoing transitions of the predecessor states are ignored as they are not needed for the computation of partition transient paths. Note that the RFPTA aggregator does not make use of the normalised steady-state vector α that is used in FPTA (see eq. 2.9), since we only ever aggregate intermediate states.

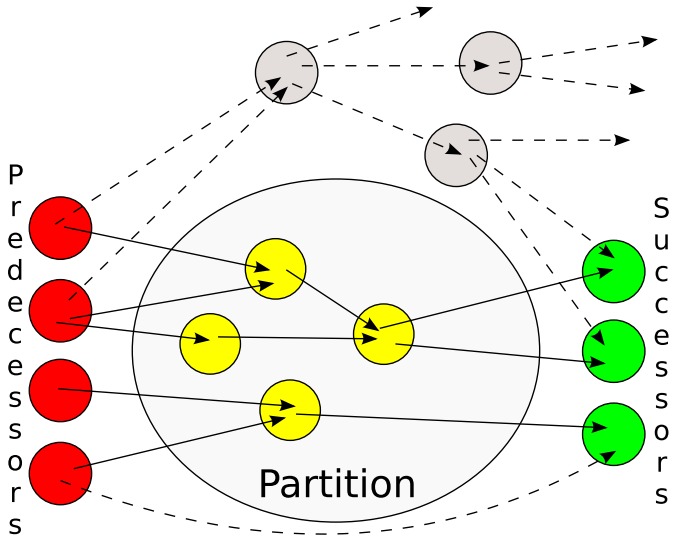


Figure 5.1: During RFPTA aggregation we first compute new transitions by adding the weighted Laplace transforms of the convolved partition transient paths, i.e. the ones using transitions with solid lines. Having computed these transitions we branch them with matching one-step transitions, if such transitions exist for a given predecessor successor pair (see for example the bottommost dashed transition).

Note. We do not normalise the transition probabilities of outgoing transitions from the predecessor states as the sum of probabilities of the transitions from a predecessor state to each of the successor states after the aggregation of the partition is the same as the sum of probabilities of the transitions from the predecessor state to the partition internal states and successor states before aggregation. This can be formally justified by the flow conservation law, as we ensure that there are no final strongly connected components of states within the partition[27].

It is worth mentioning that RFPTA aggregation can be used for partition aggregation prior to first-passage time analysis, but also potentially prior to transient probability analysis (see sect. 2.5.1). In the latter case we have to ensure that none of the predecessor states of the partition we are aggregating is a target state, otherwise we might corrupt the reliability function of that particular state. For further information on transient probability analysis see [22].

Even though RFPTA appears to be an optimal strategy for aggregating an entire partition in one go, it has one major disadvantage. Suppose we want to compute the first-passage time from

the set of source states $\vec{s} = \{s_1, s_2, \ldots, s_m\}$ to the set of target states $\vec{t} = \{t_1, t_2, \ldots, t_l\}$ of the SMP. To do this we have to calculate the vector $L_{\vec{s}} = \{L_{\vec{s}1}, L_{\vec{s}2}, \ldots, L_{\vec{s}n}\}$, i.e. the vector of first-passage time densities from the set of source states to all other states in a SMP. This vector can only be computed by solving n linear equations. We then compute $L_{\vec{s}\vec{t}}$ from elements $L_{\vec{s}t} \in L_{\vec{s}}$ with $t \in \vec{t}$ to obtain the first-passage time density as in eq. 2.9. If we want to calculate individual passage time densities from each source state to each target state we need to do more work as we cannot infer these distributions from $L_{\vec{s}t}$. Instead we have to solve m sets of n linear equations to get $L_{s_k} = \{L_{s_k1}, L_{s_k2}, \ldots, L_{s_kn}\}$ for each source state s_k.

More generally, suppose we want to aggregate an entire partition using RFPTA. Assume we have m predecessor, n successor and i partition internal states. In order to calculate the transition from every predecessor to every successor state using partition internal paths only, we have to solve m sets of $i + n$ equations. Alternatively we can reverse the computation by calculating the FPT from every successor state to every predecessor state on the transposed transition matrix. The *reverse RFPTA* computation requires us to solve n sets of $i + m$ linear equations. Note that prior to transposing the transition matrix and swapping the roles of the source and target states we still have to remove the outgoing transitions from the old target states to make them absorbing states. Reverse passage time calculation works well in Laplace space since complex multiplication is an associative operation. The technique can also be used to do normal passage time calculation without aggregation, in which case the old source states are still the ones that have to be weighted by their steady state probability (see eq. 2.10) but the target states, which become the new source states, are not weighted by their steady state probabilities. The minimum work required to aggregate a partition using RFPTA is to solve l sets of $i + g$ linear equations, where $l = min(m, n)$ and $g = max(m, n)$ as a single RFPTA computation can solve one set of linear equations at a time. Hence for RFPTA aggregation we need to find partitions that do not only keep the number of partitionwise transitions low, but also minimise either the number of predecessor states or the number of successor states of the partition. Naturally these metrics are correlated as a small number of predecessor states or successor states limits the number of transitions that are created when aggregation the partition.

Reverse RFPTA

In experiments with PaToH2D we were able to find large partitions with the required properties for efficient RFPTA aggregation in some cases. The best partitioning we could find for the web-server model with 107289 states is a PaToH2D partitioning with 4 predecessor states and 1444 successor states spanning about one third of the state space S. Aggregating this partition decreases the number of transitions by roughly one third. However, for this partitioning we have to compute 4 sets of $1/3|S|$ linear equations to aggregate the partition. To perform the passage time analysis on the resulting aggregated transition matrix we need to solve another $2/3|S|$ linear equations. Unless aggregation of a large partition makes the final passage time calculation converge faster, the combined aggregation and passage time analysis approach is likely to be slower than the computation of the passage time on the unaggregated graph. For the 106540 voting model we managed to find a partition that spans roughly 50% of the state space, while only having one predecessor state. To get this partition we had to make the target states absorbing. Therefore this partitioning is only useful if we want to apply first-passage time analysis to the aggregated transition matrix of that model. Partitionings we produced for the voting model with non-absorbing target states did not have the desired properties for RFPTA aggregation. For the smcourier model with 29010 states we did not find a suitable partition even when making all target states absorbing.

In general it is difficult to find good partitions for RFPTA aggregation. Most of the partitioners introduced in chapter 3 are designed to partition sparse matrices for parallel matrix vector multiplication. For this kind of problem it is best to balance the number of non-zero elements per partition as well as the communication load for each processor. RFPTA on the other hand works best on partitions that have an extremely low number of predecessor or successor states. In this case it does not matter whether the number of predecessor and successor states is balanced. Therefore standard graph partitioning algorithms might not be ideal for finding suitable

RFPTA partitions. Nevertheless it would be interesting to investigate if graph and hypergraph partitioners can be modified to produce better partitionings for RFPTA. This could potentially be done by finding more suitable configurations for the PaToH and MeTiS partitioner. However, it is likely that there are better algorithms for RFPTA partitioning. One algorithm we tried is the NBSS partitioner presented in defn. 3.4. Even though building a partition from a single state seems to be a sensible approach to create partitions with a low number of predecessor states, NBSS partitioning performed worse than MeTiS and PaToH. Further research into this matter might produce partitioning strategies that extend the range of semi-Markov models and performance measures for which RFPTA aggregation can be used in practice. When the measure of interest is the passage time distribution then barrier partitioning introduced in sect. 5.2 is one such alternative.

5.1.2 Discrete event simulation aggregator

For RFPTA aggregation we do not only compute the restricted first-passage time densities of the time it takes to get from the set of predecessor states $\vec{s}$ to the set of successor states $\vec{t}$, but we also keep track of all the restricted first-passage time densities from each predecessor to all other states in the partition. This is unavoidable if we want to determine the exact transitions from states in $\vec{s}$ to states in $\vec{t}$ which encounter all highly probable partition transient paths. However, we can potentially get reasonably accurate approximations to the required sojourn time distribution by examining a smaller subset of all paths considered by the RFPTA aggregator.

Discrete event simulation (DES) aggregator

Definition 5.2. Suppose we want to aggregate an entire partition in one go. For every predecessor state p of the partition the *discrete event simulation (DES) aggregator* generates partition transient paths of the form $p - i_1 - i_2 - \cdots - i_r - s$, where s is a successor state and i_k are partition internal states. It then calculates the weighted Laplace transform of the passage time of such a path and adds it to in L_{ps}. To keep the amount of computation low we only calculate a fixed amount of paths for each predecessor p.

DES aggregation is an expensive calculation if we simulate the SMP using a sub-matrix of the initial transition matrix, as we have to sample from the probability distribution over all outgoing transitions in each state. A better way of doing DES aggregation would be to generate a new high-level model for the sub-matrix so that we can perform DES without having to look up outgoing transitions in a transition matrix. For simplicity we perform DES aggregation using the sub-matrix of the SMP transition matrix. Hence we are only able to compute a small number of partition transient paths as we need to keep the time requirements of the aggregator low. It turns out that this particular DES aggregator is not suitable for atomic partition aggregation. In all our experiments with DES we did not obtain a meaningful probability distribution when performing first-passage time analysis on the aggregated transition matrix. This result is not too surprising as our DES aggregator only considers a very small subset of all partition transient paths that the RFPTA aggregator takes into account and therefore doesn't enforce flow conservation. As a consequence we do not recommend our DES aggregation approach for atomic partition aggregation.

5.1.3 RFPTA with extra vanishing state

The main problem with RFPTA aggregation is that we need to solve several sets of linear equations if the partition we are aggregating does not either have only one predecessor or successor state. We now introduce a technique that guarantees that we only need to solve one set of linear equations in order to do an approximate aggregation of a partition.

Vaninishing state

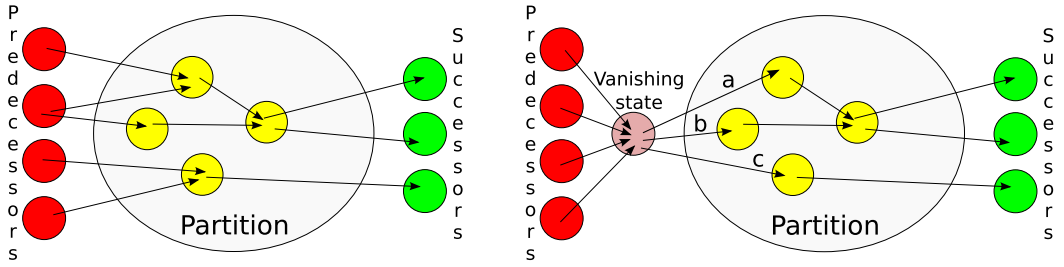
Partition entry state

Definition 5.3. States, which only have outgoing transitions with immediate sojourn time distributions are referred to as *vanishing states*. We define an *extra vanishing predecessor state* to be a vanishing state that we use to separate the predecessor states of a partition from the *partition entry states*. A partition entry state is a partition internal state that has at least

one incoming transition from one of the predecessor states of the partition. We say the extra vanishing predecessor state separates the predecessor states from the partition entry states because all transitions from the predecessor state into the partition are channelled through the extra state (see fig. 5.2(b)). Note that we can define an *extra vanishing successor state* similarly, only that in this case the extra state separates the *partition exit states* from its successor states. Exit states are partition internal states that have outgoing transitions to successor states. From now on we refer to either of the two as an *extra vanishing state* or simply an *extra state*.

Partition exit state

Extra vanishing state



(a) Transition graph before adding extra vanishing state (b) Transition graph after adding extra vanishing state

Figure 5.2: These diagrams illustrate the creation of an extra vanishing predecessor state. Through the extra state all four predecessor states have become connected to all partition entry states and can thereby reach each of the successor states of the partition. This obviously implies that the resulting graph no longer represents the initial transition system. Hence measures in the modified SMP will evaluate to different values, too.

```
1  Define Get transition(matrix, predecessor state number, successor state number);
2  Define Add extra row(matrix);
3  Define Delete transition(transition);
4  Define Find all entry states of predecessor states(partition);
5  Define Find entry states connected to state(state number, partition);
6  Define Add empty outgoing transition(matrix, predecessor state number, successor state number);
7  Define Get steady-state probability(state number);
8  Define Get sum of predecessor steady state probabilities(partition);
9  Define Sum transition probabilities of outgoing transitions to entry states(predecessor state number, set with successor state numbers);
10 Define Normalise probabilities of outgoing transitions(state number);

   input: Sparse SMP transition row matrix matrix, partition p

11 setOfEntryStates = Find all entry states of predecessor states(p);
12 extraStateNo = Add extra row(matrix);
13 foreach Entry state e in setOfEntryStates do
14     Add empty outgoing transition(matrix, extraStateNo, e);
15     transitionFromExtraState = Get transition(matrix, extraStateNo, e);
16     transitionFromExtraState.laplace = 1 + 0i /*immediate transition*/;
17 end

18 sumOfSteadyStateProbs = Get sum of predecessor steady state probabilities(p);
19 foreach Predecessor state ps in p do
20     steadyProb = Get steady-state probability(ps)/ sumOfSteadyStateProbs;
21     tempSetOfEntryStates = Find entry states connected to state(ps, p);
22     sumOfTransProbs = Sum transition probabilities of outgoing transitions to entry states(ps, tempSetOfEntryStates);
23     Add empty outgoing transition(matrix, ps, extraStateNo);
24     transitionToExtraState = Get transition(matrix, ps, extraStateNo);
25     transitionToExtraState.prob = sumOfTransProbs;

26     foreach State es in tempSetOfEntryStates do
27         transition = Get transition(matrix, ps, es);
28         transitionToExtraState.laplace += transition.prob/sumOfTransProbs * transition.laplace;
29         transitionFromExtraState = Get transition(matrix, extraStateNo, transition.destination);
30         transitionFromExtraState.prob += transition.prob * steadyProb;
31         // Disconnect predecessor state from entry state
32         Delete transition(transition);
33     end
34 end

35 Normalise probabilities of outgoing transitions(extraStateNo)
```

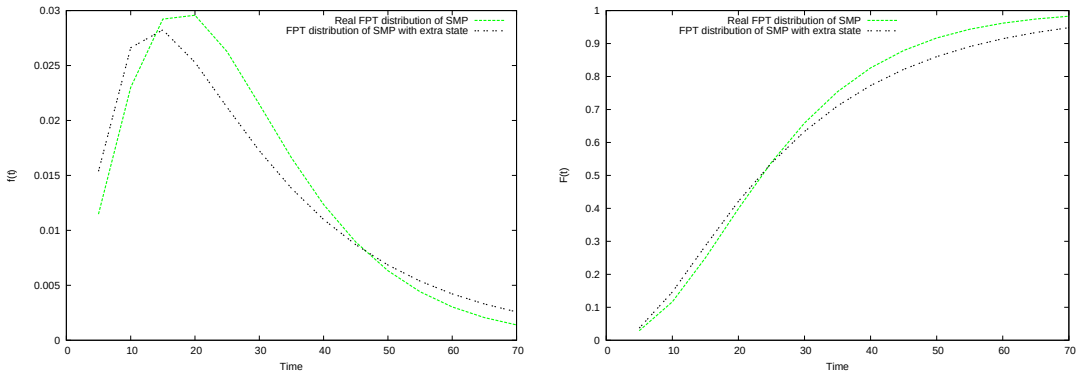
Algorithm 1: Adding an extra vanishing predecessor state

Our algorithm for adding an extra predecessor state (see algo. 1) starts by detecting all entry states of the partition we are aggregating. Subsequently we create the extra state in the SMP matrix and add outgoing transitions from the extra state to all entry states. These transitions are initialised with zero probabilities and the Laplace transform of an immediate transition. Whilst the Laplace transforms remain unchanged we compute the probabilities of the transitions. Suppose predecessor state p has an outgoing transition to entry state e with transition

probability q. Further assume that t represents the state's steady-state probability, normalised by the sum of all predecessor states' steady-state probabilities (see eq. 2.10). We add qt to the probability of the transition from the extra state to e. We repeat this for all outgoing transitions from predecessor states to partition entry states. The reason we multiply the transition probability q by t is that we try to assign more weight to transitions coming from predecessor states with higher steady state probability. We now need to disconnect each predecessor state of the partition from the entry states and channel the discarded transitions through the extra state. Each predecessor has precisely one outgoing transition to the extra state. The transition probability for each of these transitions is simply s, the sum of the probabilities of outgoing transitions from a predecessor state to the partition entry states. The Laplace transform of the sojourn time distribution of the transition is the sum of the weighted Laplace transforms of the outgoing transitions from the predecessor state to partition entry states divided by s. Note that by our construction the sum of the outgoing transition probabilities of each predecessor state still adds up to one. In the final step we normalise the outgoing transition probabilities of the extra state. An extra successor state can be added in a similar manner. To do this we use the same procedure as in algo. 1 except that we now channel the outgoing transitions of the exit states of the partition through an extra successor state. In contrast to DES aggregation, performing passage time analysis on a SMP graph that has been aggregated using RFPTA aggregation on a partition with an extra state always yields meaningful probability distributions.

5.1.3.1 Error introduced by extra vanishing state

To determine the error in the first-passage time distribution introduced by adding an extra state to the transition matrix, we created different partitionings for different models and compared the results of the FPTA on the unmodified model with results from the FPTA on the same model with an extra predecessor state. All partitionings used for the test contained partitions of similar size and we always added the extra state to the partition with the lowest number of predecessor states.



(a) PDF (b) CDF

Figure 5.3: Impact of adding an extra state to the smcourier model with 29010 states. The partition we use spans about 25% of the state space and has roughly 6000 predecessor states. The FPT distribution is calculated using the iterative approach with precision 10^{-16}. The largest error in the distribution data produced by the SMP with the extra state is of the magnitude 10^{-2}. Nevertheless the resulting pdf and cdf are good approximations to the real distribution.

Figure 5.3 shows that we can get decent approximations to the first-passage time distribution of the original SMP when analysing the modified graph with the extra state. In a second experiment we tested the impact of adding an extra predecessor state to the beforementioned partition of the 107289 states web-server model with 4 predecessor states. In this experiment we achieve a slightly better approximation with the magnitude of the maximum error being 10^{-3}

in the cdf of the FPT distribution (see sect. 6.1.1 for further details on the error evaluation). Despite encouraging results from our experiments, the biggest problem of the extra state method remains that the error introduced by the extra state heavily depends on the structure of the SMP graph. In general the only means to keep the error low is to keep the number of predecessor states low. Nevertheless the extra state method is a valuable tool as it allows approximate aggregation of partitions that are unsuited for exact aggregation using normal RFPTA. Adding the extra state was inspired by the application of hidden nodes in Bayesian inference (for more information see [26]). It is possible that there are ways of channelling the outgoing transitions of predecessor states through an extra vanishing state that keep the error term lower than our algorithm does. One way of doing this might be to introduce multiple extra vanishing states. This would allow us to refine the connectivity of the graph with the extra states to reflect the original structure of the network more accurately than a graph with only one extra vanishing state can.

5.2 Barrier partitioning

Both RFPTA aggregation as well as RFPTA aggregation using an extra vanishing state require us to find large partitions which have a low number of predecessor or successor states. As partitioners such as PaToH and MeTiS are not guaranteed to find such partitions, we need to find more suitable partitioning methods for transition graphs of large semi-Markov models. In this section we introduce a new partitioning method called *barrier partitioning*, a technique which is well-suited for first-passage time analysis. Strictly speaking it is not a partitioning method designed to generate partitions for atomic partition aggregation. However, we introduce a modified first-passage time algorithm that can be applied to barrier partitionings of the transition graph, which is similar to performing atomic partition aggregation using RFPTA.

In order to perform first-passage time analysis on a SMP with n states we need to solve n linear equations to obtain $L_{\vec{s}}$ (see sect. 5.1.1). The reason this calculation can be done at a relatively low cost is because we reduce the entire set of source states and consequently treat it as one joint state. This implies that we do not calculate the first-passage time for every pair of source and target states, but from the set of source states to each of the target states. As we mentioned earlier, first-passage time analysis can be done forward, i.e. from the set of source states to the individual target states as well as reverse, i.e. from the set of target states to the individual source states, by transposing the SMP transition matrix and swapping source and target states. The barrier partitioning method exploits the duality between the forward and reverse calculation of the first-passage time distribution and allows us to split the first-passage time calculation into two separate calculations. The combined cost of doing the two separate calculations is the same as the cost of the original first-passage time calculation.

Definition 5.4. Assume we have an SMP with a set of start states S and a set of target states T. If any state is a source and a target state at the same time it can be split into a new target and a new source state. The new source state is assigned all outgoing transitions of the old state, the new target state all incoming transitions. Finally adding an immediate transition from the new target state to the new source state gives a modified transition graph that will yield the same passage time distribution as the original graph. We then divide the state space into two partitions SP and TP. SP contains all source states and a proportion of the intermediate states such that any outgoing transitions from SP to TP go into a set of barrier states B in TP. Furthermore the only outgoing transitions from states in TP to states in SP are from target states T to source states S. The resulting partitioning is a *barrier partitioning*. See fig. 5.4 for a graphic representation of the partitioning.

Barrier partitioning

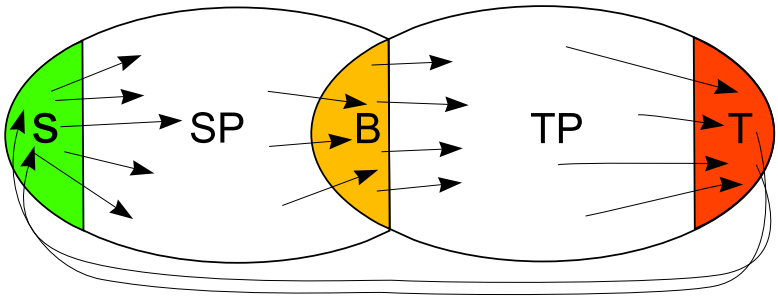


Figure 5.4: The source partition SP contains all states in S as well as the intermediate states between S and B. The target partition TP contains all barrier states B as well as all states between B and T and of course the target states in T. Note that B and T may intersect. All outgoing transitions in SP are either internal or go into B. Similarly all outgoing transitions of states in TP are either internal or transitions from states in T to states in S. Thus once a path has entered TP it can only ever go back to SP by going through T.

Proposition 5.1. Assume that we can divide the state space Ω of a connected SMP graph into two partitions such that the resulting partitioning is a barrier partitioning. Clearly we have $S \cap T = \emptyset$, $SP \cup TP = \Omega$. We denote the set S as $\vec{s}$, the set of barrier states B as $\vec{b}$ and the set of target states T as $\vec{t}$. The result of the first-passage time calculation from a source state s to the set of target states $\vec{t}$ is same as the result obtained by doing a first-passage time calculation from s to the set of barrier states $\vec{b}$ convoluted with the first-passage time calculation from the set of barrier states $\vec{b}$ to the set of target states $\vec{t}$. In the Laplace domain this translates to:

$$L_{s\vec{t}} = \sum_{b \in \vec{b}} L^R_{sb} L_{b\vec{t}}$$

where L^R_{sb} denotes a restricted first-passage time distribution from state s to state $b \in \vec{b}$, where all states in $\vec{b}$ are made absorbing for the calculation of L^R_{sb}. This ensures that we only consider paths of the form $s - i_1 - \cdots - i_k - b$, with $i_j \in SP$. In other words we do not consider paths through TP for the calculation of L^R_{sb}.

Note. $L_{b\vec{t}}$ is the Laplace transform of the first-passage time distribution from state b to the set of target states $\vec{t}$ which are absorbing states in first-passage time analysis.

Proof. By eq. 2.8 we have

$$L_{s\vec{t}} = \sum_{k \in (SP \cup TP) \setminus \vec{t}} r^*_{sk} L_{k\vec{t}} + \sum_{k \in \vec{t}} r^*_{sk}$$

hence

$$L_{s\vec{t}} = \sum_{k \in (SP \cup TP)} r^*_{sk} L_{k\vec{t}}$$

where $L_{k\vec{t}}$ is equal to 1 if $k \in \vec{t} \cap \vec{b}$. We can rewrite $k \in SP \cup TP$ as $k \in SP \cup \vec{b}$ since there is no transition from any state in SP to any state in $TP \setminus \vec{b}$ by construction of the barrier.

$$\begin{aligned} L_{s\vec{t}} &= \sum_{k \in (SP \cup \vec{b})} r^*_{sk} L_{k\vec{t}} \\ &= \sum_{b \in \vec{b}} r^*_{sb} L_{b\vec{t}} + \sum_{k \in SP} r^*_{sk} L_{k\vec{t}} \end{aligned}$$

also by construction of the barrier partitioning and the fact that target states are absorbing states we know that once we have entered TP (i.e. reached a state in $\vec{b}$) we cannot find a path

back to a state in SP. Hence

$$\begin{aligned} L_{s\vec{t}} &= \sum_{b\in\vec{b}} r^*_{sb} L_{b\vec{t}} + \sum_{k\in SP} r^*_{sk} \sum_{b\in\vec{b}} L^R_{kb} L_{b\vec{t}} \\ &= \sum_{b\in\vec{b}} r^*_{sb} L_{b\vec{t}} + \sum_{b\in\vec{b}} \sum_{k\in SP} r^*_{sk} L^R_{kb} L_{b\vec{t}} \\ &= \sum_{b\in\vec{b}} \left(r^*_{sb} L_{b\vec{t}} + \sum_{k\in SP} r^*_{sk} L^R_{kb} L_{b\vec{t}} \right) \\ &= \sum_{b\in\vec{b}} \left[\left(\sum_{k\in SP} r^*_{sk} L^R_{kb} + r^*_{sb} \right) L_{b\vec{t}} \right] \end{aligned}$$

by definition $\sum_{k\in SP} r^*_{sk} L^R_{kb} + r^*_{sb}$ is the restricted first-passage time from state s to barrier state b. Therefore

$$L_{s\vec{t}} = \sum_{b\in\vec{b}} L^R_{sb} L_{b\vec{t}}$$

■

Corollary 5.1.1.

$$L^R_{s\vec{t}} = \sum_{b\in\vec{b}} L^R_{sb} L^R_{b\vec{t}}$$

Proof. We have

$$L^R_{b\vec{t}} = L_{b\vec{t}}$$

since target states are absorbing states by assumption and because none of the outgoing transitions of non-target barrier states go into SP. Furthermore

$$L^R_{s\vec{t}} = L_{s\vec{t}}$$

as the restricted first passage time distribution on the entire state space is just the normal passage time distribution. ■

Corollary 5.1.2. Let $L^R_{\vec{sb}} = \{L^R_{\vec{sb}_1}, \ldots, L^R_{\vec{sb}_l}\}$, where $L^R_{\vec{sb}_i} = \{\alpha_1 L^R_{s_1 b_i} + \cdots + \alpha_l L^R_{s_l b_i}\}$ and $L_{\vec{bt}} = \{L_{b_1\vec{t}}, \ldots, L_{b_l\vec{t}}\}$ then in steady-state we have $L_{\vec{st}} = \sum_{b\in\vec{b}} L^R_{\vec{sb}} L_{b\vec{t}} = L^R_{\vec{sb}} \,.\, L_{\vec{bt}}$

Proof. Let $\alpha_1, \alpha_2, \ldots, \alpha_l$ be the normalised steady-state probabilities of the source states $\vec{s} = (s_1, s_2, \ldots, s_l)$ as defined in eq. 2.10. By eq. 2.9 we have

$$\begin{aligned} L_{\vec{st}} &= \alpha_1 L_{s_1\vec{t}} + \alpha_2 L_{s_2\vec{t}} + \cdots + \alpha_l L_{s_l\vec{t}} \\ &= \sum_{b\in\vec{b}} \left(\alpha_1 (L^R_{s_1 b} L_{b\vec{t}}) + \cdots + \alpha_l (L^R_{s_l b} L_{b\vec{t}}) \right) \\ &= \sum_{b\in\vec{b}} \left(\alpha_1 L^R_{s_1 b} + \cdots + \alpha_l L^R_{s_l b} \right) L_{b\vec{t}} \end{aligned}$$

■

5.2.1 Passage time computation on barrier partitionings

In practice there are two ways of computing the steady state first-passage time distribution of a model whose state space has been split into partitions SP and TP. The first one is purely sequential. We start by calculating vector $L_{\vec{sb}}$ using the iterative first-passage time solver. For this calculation the source states remain unmodified, but the barrier states become absorbing target states. Also as this calculation is part of the final first-passage time calculation we need to weight the source states by their normalised steady state probabilities. Having calculated $L_{\vec{sb}}$ we use it

as our ν_0 (see eq. 2.17) in the subsequent first-passage time calculation from the set of barrier states to the set of target states. Note that the calculation of $L_{\vec{sb}} = \nu_0$ for the subsequent calculation of $L_{\vec{bt}}$ is in fact an atomic aggregation of the intermediate states in source partition SP.

Another way of doing first passage time analysis on a barrier partitioning is to compute $L_{\vec{sb}}$ and $L_{\vec{bt}}$ independently. By coroll. 5.1.2 the dot product of the two vectors gives us $L_{\vec{st}}$. To calculate vector $L_{\vec{bt}}$ independently from $L_{\vec{sb}}$ we do a reverse first-passage time calculation from the set of target states to the barrier states. In order to do this we need to remove all transitions from SP into the set of barrier states. All incoming transitions from any state in the target partition to any of the barrier states remain, including transitions from one barrier state to another. Note that we do not need to weight the target states by α as we have already weighted the source states during the calculation of $L_{\vec{sb}}$.

Both techniques can be used to reduce the amount of memory that we need for a first-passage time calculation as we only have to keep either the sub-matrix of the source partition or the target partition in memory at any point in time. Moreover the second approach is parallelisable.

5.2.2 Balanced barrier partitioner

Another advantage of barrier partitionings over partitionings produced by graph and hypergraph partitioners presented in chapter 3 is that we can easily find barrier partitions in large models at low cost. A barrier partitioning can be found as follows. Firstly since we are doing first-passage time analysis we can discard the outgoing transitions from all target states. Secondly we explore the entire state space using breadth-first search, with all source states being at the root level of the search. We store the resulting order in an array. To find a barrier partitioning we first add all non-target states among the first m states in the array to our source partition. Note that m has to be larger or equal to the number of source states in the SMP. We then create a list of all predecessor states of the resulting partition. In the next step we add all predecessor states in the list to the source partition and recompute the list of predecessor states. We repeat this until we have found a source partition with no predecessor states. Since we discarded all outgoing edges of the target states, this method must give us a barrier partitioning. In the worst case this partitioning has all source and intermediate states in SP and TP only contains the set of target states. Fortunately in all models we analysed we were able to find far better barrier partitionings. Algorithm 2 describes a general method for finding balanced barrier partitionings given a transition matrix of a semi-Markov or Markov model. *Balanced barrier partitionings* are barrier partitionings where SP and TP contain a similar number of transitions.

Balanced barrier partitioning

In both the voting and the web-server model (see fig. B.1) it is possible to split the state space such that each partition contains roughly 50% of the total number of transitions. Even more surprisingly we easily found balanced partitionings for large versions of these two models with several million transitions. In addition to this our barrier partitioning algorithm is very fast (see sect. 5.4.1). However, despite the fact that barrier partitioning works well on the first two models it is not possible to barrier partition the smcourier model such that each partition has an equal amount of transitions. Figure 5.5 shows the best barrier partitioning for the smcourier model. The main reason why it is impossible to balance the barrier partitions in this model is because of the fact that roughly 50% of the state space are source and target states.

```
1  Define Make target states absorbing(matrix, target states);
2  Define Find breath-first ordering(matrix, source states);
3  Define Get number of rows(matrix);
4  Define Get first m non-target states(array, stopIndex);
5  Define Get predecessor states(matrix, partition, target states);
6  Define Merge arrays(array, array);
7  Define Count number of transitions(matrix, optional array);

   input  : Sparse SMP transition row matrix matrix, source states s⃗, target states t⃗
   output: Barrier source partition

8  Make target states absorbing(matrix, t⃗);
9  bforder = Find breath-first ordering(matrix, s⃗);
10 numSourceStates = |s⃗|;
11 numStates = Get number of rows(matrix);
12 m = numStates / 2;
13 mStep = numStates / 4;
14 partition = ∅;
15 foundBalancedBarrierPartitioning = false;
16 while foundBalancedBarrierPartitioning == false && mStep > 1 do
17     partition = Get first m non-target states(bforder, m);
18     predecessors = Get predecessor states(matrix, partition, t⃗);
19     while predecessors is not empty do
20         partition = Merge arrays(partition,predecessors);
21         predecessors = Get predecessor states(matrix, partition, t⃗);
22     end
23     SPTPBalance = Count number of transitions(matrix,partition) / Count number of transitions(matrix);
24     if SPTPBalance < 0.45 then
25         m += mStep;
26     end
27     else if SPTPBalance > 0.55 then
28         m -= mStep;
29     end
30     else
31         foundBalancedBarrierPartitioning = true;
32         break;
33     end
34     mStep = mStep / 2;
35     partition = ∅;
36 end
37 return partition;
```

Algorithm 2: Balanced barrier partitioning

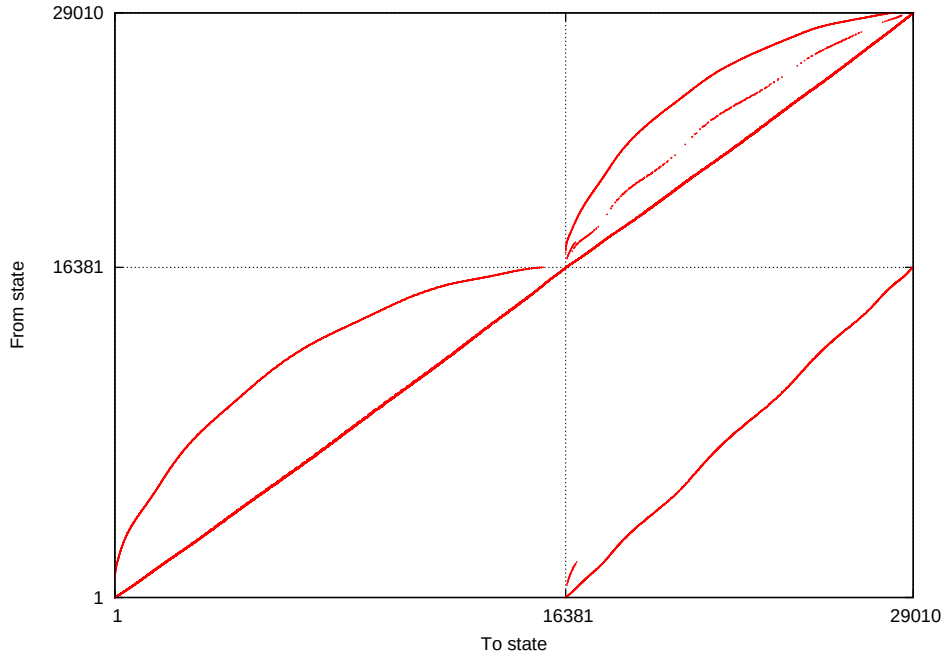


Figure 5.5: This is the best barrier partitioning for the smcourier model. It was obtained by choosing m to be the number of source states. The source partition contains 56% of all states and 64% of all transitions. Note that the diagonal matrix in the upper left corner has no entries. Thus there is no transition from the target partition to the source partition. Further note that every state in the target partition is a barrier state in this example. In balanced barrier partitionings of the voting and web-server model the set of barrier states is only a small subset of the target partition.

5.3 K-way barrier partitioning

k-way barrier partitioning

The idea of barrier partitioning described in the previous section is a huge improvement to the straightforward passage time calculation, as it reduces the amount of memory needed for the passage time computation while introducing very little overhead. In this section we investigate the idea of *k-way barrier partitioning*. In practice a k-way barrier partitioning is desirable since it allows us to reduce the amount of memory needed to perform passage time analysis on Markov and semi-Markov models by even more than 50%.

Definition 5.5.

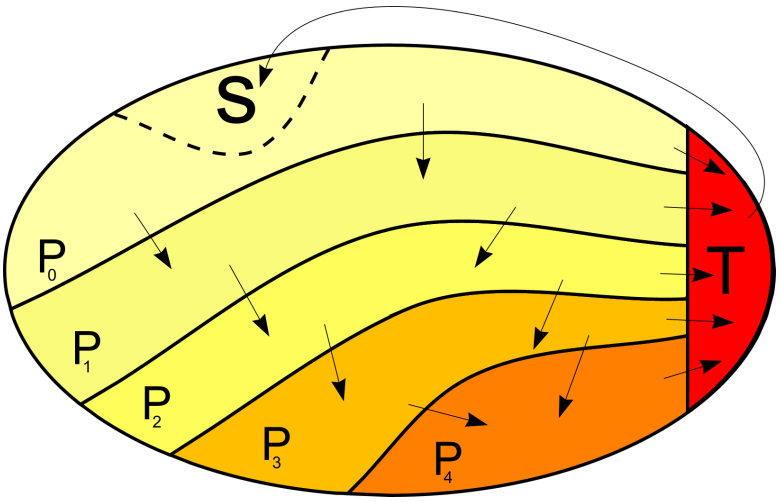


Figure 5.6: In a *k-way barrier partitioning*, partition P_0 contains the source states, partition T the target states. There are $k-2$ intermediate partitions and $k-1$ barriers in total. In general partition P_j is sandwiched between its predecessor partition P_{j-1} and its successor partitions P_{j+1} and T. Note that there are no transitions from partition P_i to P_j if $i > j$, hence the barrier property is satisfied in the sense that once we have reached P_j the only way to get back to any state in P_{j-1} is to go through T. T is the only predecessor partition of P_0. The barrier states of partition P_j are the union of T and the states of P_{j+1} that have incoming transitions from states in P_j.

Note. Definition 5.5 generalises defn. 5.4. The latter definition corresponds to a 2-way barrier partitioning. In defn. 5.4 we did not define the set of barrier states to be the union of states that separate SP from TP and the set of states in T. However, this generalisation has no impact on prop. 5.1 as we assumed that B and T may intersect.

The difference between the standard 2-way barrier partitioning and the general k-way barrier partitioning with $k > 2$ is the way we compute the passage time on the transition matrix of a model that has been partitioned into k barrier partitions. Whilst the passage time analysis on the 2-way partitioning is fully parallelisable by coroll. 5.1.2, the analysis on a k-way barrier partitioning is generally less parallelisable. The following proposition verifies the correctness of the passage time analysis on a k-way barrier partitioning.

Proposition 5.2.

$$L_{s\vec{t}} = L^R_{s\vec{b}_1} \; M^R_{\vec{b}_1\vec{b}_2} \; \cdots \; M^R_{\vec{b}_{k-2}\vec{b}_{k-1}} \; L^R_{\vec{b}_{k-1}\vec{t}} \tag{5.1}$$

where $L^R_{s\vec{b}_1}$ is the $1 \times m_1$ row vector containing the resulting Laplace transforms of the restricted passage time analysis from start state s to the states in the first barrier $\vec{b_1}$. $L^R_{\vec{b}_{k-1}\vec{t}}$ is a $m_{k-1} \times 1$ column vector of the Laplace transforms from the passage time from the states in the $k-1^{\text{st}}$

barrier to the joint set of target states and

$$M^R_{\vec{b_{i-1}}\vec{b_i}} = \begin{pmatrix} L^R_{\vec{b}_{i-1,1}\vec{b}_i} \\ L^R_{\vec{b}_{i-1,2}\vec{b}_i} \\ \vdots \\ L^R_{\vec{b}_{i-1,m_{i-1}}\vec{b}_i} \end{pmatrix} = \begin{pmatrix} L^R_{\vec{b}_{i-1,1}\vec{b}_{i,1}} & \cdots & L^R_{\vec{b}_{i-1,1}\vec{b}_{i,m_i}} \\ \vdots & & \vdots \\ L^R_{\vec{b}_{i-1,m_{i-1}}\vec{b}_{i,1}} & \cdots & L^R_{\vec{b}_{i-1,m_{i-1}}\vec{b}_{i,m_i}} \end{pmatrix}$$

the $m_{i-1} \times m_i$ matrix containing the Laplace transform samples from the restricted passage time analysis from barrier $i-1$ to barrier i for each pair of barrier states, i.e. pairs (a, b) where a lies in barrier $i-1$ and b in barrier i. Note that if state j is a target state then $L^R_{\vec{b}_{i-1,j}\vec{b}_{i,j}} = 1$ and $L^R_{\vec{b}_{i-1,j}\vec{b}_{i,l}} = 0,\ \forall\ l \neq j$ as j must be an absorbing state.

Proof. First we show that

$$L^R_{s\vec{b}_2} = L^R_{s\vec{b}_1}\ M^R_{\vec{b}_1\vec{b}_2}$$

by coroll. 5.1.1 we have

$$L^R_{sb_{2,i}} = \sum_{j=1}^{m_1} L^R_{sb_{1,j}} L^R_{b_{1,j}b_{2,i}}$$

then

$$\begin{aligned} L^R_{s,\vec{b}_2} &= \left(\textstyle\sum_{j=1}^{m_1} L^R_{sb_{1,j}} L^R_{b_{1,j}b_{2,1}},\ \ldots\ , \sum_{j=1}^{m_1} L^R_{sb_{1,j}} L^R_{b_{1,j}b_{2,m_2}}\right) \\ &= L^R_{s\vec{b}_1}\ M^R_{\vec{b}_1\vec{b}_2} \end{aligned}$$

using this argument repeatedly reduces eq. 5.1 to

$$\begin{aligned} L_{s\vec{t}} &= L^R_{s\vec{b}_{k-1}} L^R_{\vec{b}_{k-1}\vec{t}} \\ &= \textstyle\sum_{j=1}^{m_{k-1}} \left(L^R_{sb_{k-1,j}} L^R_{b_{k-1,j}\vec{t}}\right) \end{aligned}$$

which holds by prop. 5.1 since

$$L^R_{b_{k-1,j}\vec{t}} = L_{b_{k-1,j}\vec{t}}$$

as target states are absorbing states during first-passage time analysis. ∎

Corollary 5.2.1.

$$L_{\vec{st}} = L^R_{\vec{sb}_1}\ M^R_{\vec{b}_1\vec{b}_2}\ \cdots\ M^R_{\vec{b}_{k-2}\vec{b}_{k-1}}\ L^R_{\vec{b}_{k-1}\vec{t}}$$

Proof. Similar argument as in coroll. 5.1.2 ∎

Algorithm 3 describes how sequential passage time analysis can be performed on a k-way barrier partitioning. The basic idea is to initialise $\nu_0^{(0)}$ (see *eq.* 2.17) with the α weighted source states, compute $L^R_{\vec{sb}_1} = \nu_0^{(1)}$ using $\nu_0^{(0)}$ and subsequently use $\nu_0^{(1)}$ as the new start vector for the calculation of $L^R_{\vec{sb}_2} = \nu_0^{(2)}$. We continue until we obtain $\nu_0^{(k)} = L_{\vec{s}}$ (see sect. 5.1.1), the final s-point $L_{\vec{st}}$ is computed by summing all Laplace transforms $L_{\vec{st}} \in L_{\vec{s}}$ with $t \in \vec{t}$. We can avoid calculating the matrices M^R explicitly as we treat the source states as one joint state. Intuitively this approach makes sense because $\nu_0^{(i)}$ always contains the Laplace transform distribution from the initial set of source states to the states of the i^{th} barrier and when used as the start vector for the next iterative restricted passage time analysis, we obtain the Laplace transform of the distribution from the joint set of source states to all states that lie in the i^{th} partition and the states of the $i+1^{\text{st}}$ barrier. Since we are only interested in the Laplace transform of the passage time distribution from the set of source states to the current barrier states, we can set all other values in $\nu_0^{(i+1)}$ to zero, as their values will not be used during the next restricted passage time computation due to the nature of the barrier construction.

```
1  Define Set ν_0^(0) to the α weighted source states(matrix, start states);
2  Define Get successor states of partition(partition);
3  Define do RFPTA(matrix, start distribution, barrier states);

   input  : Sparse SMP transition row matrix matrix, source states s⃗, target states t⃗, barrier partitioning Π
   output: S-point

4  Set ν_0^(0) to the α weighted source states(matrix, s⃗);
5  i = 1;
6  foreach Partition P in Π do
7  |   barrierStates = Get successor states of partition(P) ∪ t⃗;
8  |   ν_0^(i) = do RFPTA(matrix, ν_0^(i−1), barrierStates);
9  |   set all non-barrier state entries in ν_0^(i) to 0;
10 |   i++;
11 end
12 complex sPoint = 0;
13 foreach State t in t⃗ do
14 |   sPoint += ν_0^(k)[t];
15 end
16 return sPoint;
```

Algorithm 3: Passage time analysis on k-way barrier partitioning.

In principal k-way passage time computation can be done in parallel, however, the fact that we need to compute $k-1$ passage time matrices means that we have to do a lot more work than in the sequential algorithm. Recall that we were experiencing the exact same problem in sect. 5.1.1, when we discussed RFPTA aggregation of partitions with multiple predecessor and successor states. Due to this problem we prefer the sequential passage time algorithm for the k-way barrier case. We can still parallelise the k-way barrier passage time analysis using two groups of machines for the computation of a single s-point. Both groups use the sequential algorithm, but perform each restricted passage time analysis in parallel. One group does the forward passage time calculation starting from the start states, the other one does the reverse passage time calculation starting from the target states. Just like in the 2-way barrier case the two groups of processors will stop when they have reached the middle barrier. By coroll. 5.1.2 we can then compute the final Laplace transform of the s-point. Note that using k-way barrier partitioning in order to partition the matrix is useful as graph and hypergraph partitionings are much more expensive to compute on large matrices than barrier partitionings. However, it is still advisable to use a hypergraph partitioner to partition each of the resulting barrier partitions when doing parallel restricted passage time analysis.

5.3.1 K-way barrier partitioner

There are various ways of creating k-way barrier partitionings for SMPs. One way is recursive bi-partitioning using algo. 2 to split sub-partitions into two balanced barrier partitions at each step. Alternatively we can modify algo. 2 to get the maximum number of barriers for a given transition matrix. The modified partitioner works as follows. First we make all target states absorbing states. We then add the source states and all their predecessor states to the first partition. Subsequently we add the predecessor states of the predecessor states of the source states to the partition and so on. Once we have no more predecessor states we have found the first partition. The non-target successor states, i.e. non-target barrier states, of that partition are then used to construct the second partition in the same manner. However, we now only consider those predecessor states of the non-target barrier states that have not been explored yet, i.e. those that haven't been assigned to any partition. We continue partitioning the state space until all states have been assigned to a partition. This partitioning approach yields the maximum number of barrier partitions for a given transition graph as we only include the minimum number of states in every barrier partition. We term this a k_{max}-way barrier partitioning, but we will also refer to it as a *max-way barrier partitioning*. Note that from this partitioning we can generate any k-way partitioning with $k < k_{max}$ since joining two neighbouring barrier partitions creates a new larger barrier partition. The k_{max}-way barrier partitioning also minimises the maximum partition size among the barrier partitionings. Another important thing to note is that the partitioner is very memory efficient as we never have to hold the entire matrix in memory during the partitioning process. As we only have to scan every transition twice -

Max-way barrier partitioning

once when we look for the predecessor states of a state and a second time when we look for its successor states - a disk-based partitioning approach is also feasible. This is a huge advantage compared to algo. 2, for which a disk based solution is less feasible since we need to scan large parts of the matrix multiple times in order to create two balanced partitionings.

We tested the new partitioning method on the 1100000 states voting model and the 1000000 states web-server model. In the voting model we found a 349-way barrier partitioning, whose largest partition contains only 0.6% of the total number of transitions. In the web-server model a 332-way barrier partitioning exists in which the largest partition contains about 0.5% of the total number of transitions. For both models it is thus possible to compute the exact first-passage time while saving 99% of the memory needed by the standard iterative passage time analysis that works on the unpartitioned transition matrix. This is because of the fact that algo. 3 only ever has to hold the matrix elements of one single partition in memory. Like algo. 2 the general k_{max}-way barrier partitioning method is very fast (see sect. 5.4.1). In sect. 6.1.2 we further show that first-passage time analysis on k-way barrier partitioned transition matrices is faster than the first-passage time analysis on the unpartitioned graph.

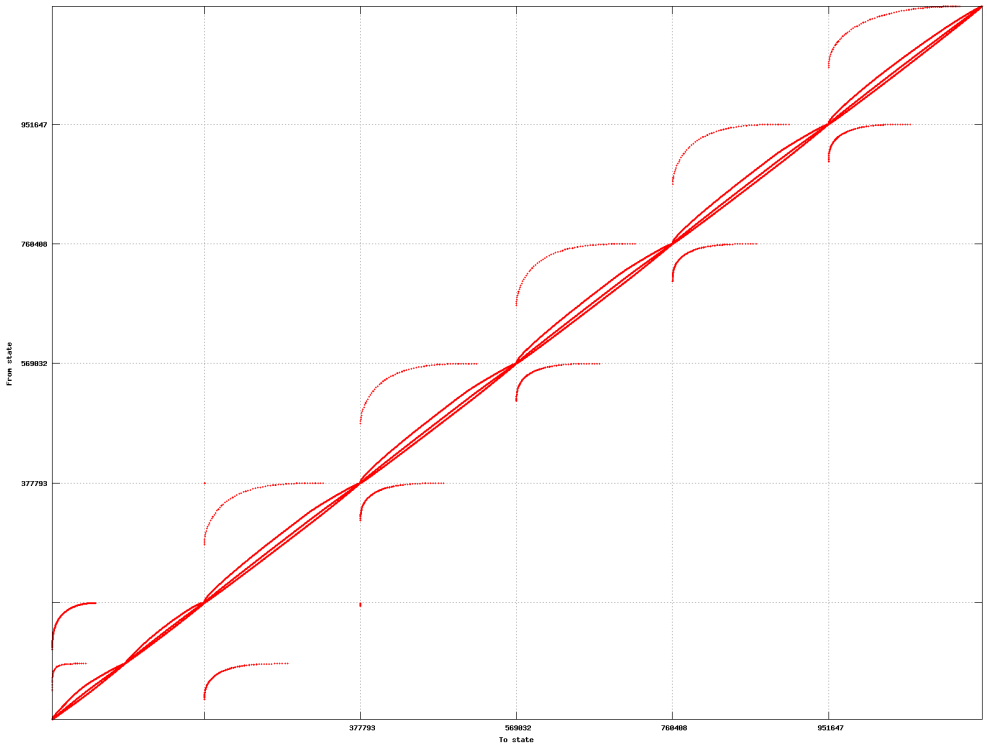


Figure 5.7: 6-way barrier partitioning of 1100000 voting model.

5.4 Implementation of atomic partition aggregation

Atomic partition aggregation requires simpler algorithms than state-by-state aggregation (see sect. 4.4). The first difference between the two forms of aggregation is that partition sorting techniques are not needed for atomic partition aggregation. In contrast to state-by-state aggregation of partitions we look for partitionings with one large partition, which has a small number of predecessor or successor states. Therefore the search space for potential partition orderings is much lower. When doing barrier partitioning we do not need to determine any aggregation ordering at all. In addition to this the only time we modify elements in the rows of the transition matrix during atomic aggregation is when we update the transitions from predecessor

states to successor states of the partition after we have performed RFPTA. FPTA with barrier partitioning can even be done without any matrix manipulation. Finally atomic aggregation of partitions does not suffer from the transition matrix fill-in problem during aggregation of a partition as exact state-by-state aggregation of partitions does.

The access patterns for the row matrix in atomic partition aggregation are far more linear than in the state-by-state aggregation case as we usually read, write and delete entire rows at once. It is thus feasible to use arrays or vector containers rather than map containers to store the rows of the transition matrix. Vector containers are more or less intelligent arrays which keep a record of how much memory the underlying array has allocated and resizes automatically if more memory is required. This is beneficial in two ways. Firstly the access times for vectors are much faster than those for maps. If we sort the destinations of the transitions in each row, which is a sensible thing to do since we do not modify the rows that often, we can even find a single element in a row in $O(\log n)$. Also as we modify rows seldom we hardly ever need to perform the expensive vector resizing operation. The second major advantage of vector containers is that they require far less memory than maps. In the C++-STL maps are balanced binary trees[30], where each node contains a pointer to its parent and its children. Thus maps require 3 extra integer pointers per element, whilst vectors do not have this overhead.

5.4.1 Performance RFPTA

In our implementation RFPTA is faster than RFPTA with extra vanishing state as we have not optimised the algorithm for inserting a new state (see algo. 1). However, it is very likely that this overhead can be minimised using caching techniques. We only tested RFPTA to explore the error introduced by adding an extra state prior to calculating the passage time distribution. The runtime of the passage time analyser for the smcourier model with extra state (see sect. 5.1.3.1) is twice as long as the runtime of the first-passage time calculation on the unmodified SMP graph. Even adding an extra state to the 4 predecessor states of the web-server model partition slows down the passage time computation by a factor of 2.

We did further tests for the normal RFPTA without extra states. Unfortunately the only model that we could test the algorithm on was the voting model, as we could not find large partitions with only one predecessor or successor state in other models. The RFPTA algorithm was tested on an Intel Duo Core 1.8 Ghz processor with 1 Gbyte of RAM. For the 106540 states voting model the total time taken by our program to do RFPTA aggregation on a large partition and the subsequent passage time analysis for 165 Laplace transform samples with convergence precision 10^{-16} was 306 seconds. The total number of complex Laplace transform multiplications was 2,553,489,711. In contrast to that it took 398 seconds and 3,709,928,347 complex multiplications to do the same passage time calculation on the initial SMP graph without aggregation. The 165 Laplace transform samples were inverted using the Euler inversion technique with $m = 20$ and $n = 12$ (see sect. 2.4.1.1). The maximum error occurred in the 13^{th} decimal place in both pdf and cdf.

In our last example atomic aggregation actually yields a speed-up on top of its ability to do first passage time analysis using less memory than the first-passage time calculation on the unaggregated SMP. Clearly the reason for this is that FPTA on an aggregated transition matrix allows us to explore longer paths with fewer iterations, since the transitions between the former predecessor and successor states of the aggregated partition encapsulate the information of many paths. Unfortunately further experiments on FPTA using barrier partitioning and cheap state aggregation revealed that aggregation does not always achieve speed-ups. One reason for this is the convergence check used by the iterative passage time algorithm (see eq. 2.21). The iterative algorithm only stops once the largest absolute value of any element in ν_r becomes less than the chosen precision ϵ. If we aggregate partitions whose internal paths have lower probabilities than those going through the states outside the partition then the largest value in ν_r can remain unaffected by the aggregation during the final computation of the passage time on the

aggregated matrix. At the same time the average absolute value among all elements in ν_r after every iteration is lower than in the unaggregated case. This makes sense because r iterations with the iterative passage time analyser on the aggregated transition matrix include paths of length longer than r in the original unaggregated graph, which should make at lot of values in vector ν_r tend to zero faster. If aggregation does not speed up the convergence of the passage time analysis, then first-passage time analysis with atomic aggregation can be more expensive than standard FPTA as we have the overhead of performing the aggregation.

Another reason for a slow-down is the fill-in behaviour of ν_r during the iterative passage time calculation. In our implementation we only multiply elements in ν_r with the matrix that have non-zero values. In some experiments we found that aggregation slows down passage time analysis as ν_r fills in faster with non-zero values when passage time analysis is done on aggregated SMP models. Hence even if aggregation speeds up the convergence of the passage time algorithm in these cases, it can happen that we need more complex multiplications to aggregate the matrix and do the final passage time calculation than we need for the analysis of the unaggregated transition matrix. In chapter 6 we therefore discuss techniques that allow us to reduce the number of complex multiplications without introducing significant numerical errors.

5.4.2 Performance of the barrier strategies

The computation of a balanced barrier partitioning for the 1.1 million state voting model takes less than 10 seconds on an Intel Duo Core 1.8 Ghz processor and 1 Gbyte of RAM. The computation of a 2-way partitioning with PaToH2D takes about 60 seconds on the same machine, but the resulting partitioning is not even suitable for RFPTA. For the 1100000 states voting model the max-way barrier partitioner needs 72 seconds on an Intel P4 3 Ghz with 4 Gbyte of RAM to find the barrier partitioning with the maximum number of partitions. In the 1000000 states web-server model the partitioner takes 35 seconds to find the max-way barrier partitioning. Given the fact that the voting model has about twice as many transitions as the web-server model (see tables A.1 and A.2), it is reasonable to assume that the complexity of finding a partitioning grows linearly with the size of the problem. This assumption is realistic as the partitioning algorithm looks at the incoming transitions of every state exactly once. This result is promising as it suggests that the partitioning algorithm is likely to perform well on larger models, too. Hence barrier partitioning does not only allow us to save an enormous amount of memory during passage time analysis but also the partitioning method itself has a much lower complexity than for instance graph and hypergraph partitioners.

5.5 Summary

Provided we find a suitable partition, atomic partition aggregation is a lot more feasible than state-by-state aggregation of partitions, as we can use the efficient iterative passage time algorithm for RFPTA aggregation. Aggregation should be considered as a tool for reducing the amount of memory needed for extracting measures from SMPs, but we should not necessarily expect speed-ups. When performing first-passage time analysis in semi-Markov and Markov models, k-way barrier partitioning certainly is the method of choice, provided we can find such a partitioning. We have shown that first-passage times in models that qualify for k-way barrier partitioning can be computed using significantly less memory, which should enable us to massively increase the size of models for which first-passage time analysis is feasible on modern computer hardware. The smcourier model example (see fig. 5.5), however, suggests that barrier partitioning is only feasible if the proportion of source and targets states in the transition graph is low. Further research is needed to explore if Markov and semi-Markov models, which satisfy this requirement, generally have suitable k-way barrier partitionings for passage time analysis.

CHAPTER 6

Applying new techniques for faster FPTA calculation

In this chapter we investigate how well techniques discussed in chapters 3, 4 and 5 perform with respect to first-passage time analysis on semi-Markov processes. First we compare the effect of 2-way barrier partitioning and cheap state aggregation with regard to the number of complex multiplications needed for aggregation and subsequent first-passage time analysis of semi-Markov models. We then introduce and test a new numerical truncation technique, which enables us to reduce the cost of computing the first-passage time distribution in large semi-Markov models up to 75% without introducing significant errors. Finally we discuss the k-way barrier method and briefly investigate how our truncation technique for the iterative passage time analysis can be parallelised.

6.1 FPTA techniques

In the following we distinguish between doing first-passage time analysis with a 2-way barrier partitioning and without. Furthermore we test the effect of cheap state aggregation. In fig. 6.1 the relevant measurements for this discussion are labelled *NoBarrier, NoBarrierCheap, Barrier* and *BarrierCheap*, where NoBarrier is the standard application of the iterative passage time algorithm on the initial transition matrix as described in sect. 2.6.1. See fig. C.1 for the results of the same experiment on the web-server model.

6.1.1 Error analysis

The largest error term introduced by 2-way barrier partitioning and cheap state aggregation is of magnitude 10^{-12} for the cdf in all our experiments on the voting model and the web-server model. Since we only used a convergence precision of 10^{-16} for the convergence test of the iterative passage time solver the error is acceptable. This validates our theoretical results about the exactness of first-passage time analysis on aggregated matrices and on barrier partitionings. Note that the error we describe here is the *Kolmogorov–Smirnov* statistic[29]. In our case the K–S statistic measures the absolute difference between the cdfs of the NoBarrier method and the cdfs of the other FPTA techniques.

Kolmogorov–Smirnov (K–S)

It is hard to say which of the four techniques yields the results that are closest to the theoretical distribution. In general we would expect cheap state aggregation to yield the most accurate results, but due to the nature of the convergence check of the iterative passage time algorithm, which we discussed in sect. 5.4.1 it could also be the case that the normal FPT computation without aggregation yields more accurate results. In the 1100000 states voting model it certainly

is the NoBarrierCheap method as it does as many iterations as the NoBarrier method, but on an aggregated graph, which implies that more paths are taken into account. In any case it is reassuring to know that the difference between the results of all four techniques is small. This allows us to freely choose between any of these techniques for passage time computation.

6.1.2 Performance

The graph of the NoBarrierCheap method in fig. 6.1 proves our earlier conjecture that aggregation does not necessarily reduce the amount of computation needed for first-passage time analysis. It is interesting to see such a sudden increase in the number of complex multiplications needed by the NoBarrierCheap method between the 500000 states voting model and the one with 1100000 states. Further investigation revealed that in this case the increase in the number of transitions is caused by the faster fill-in of the ν_r vector (see sect. 5.4.1). This can be deduced from the fact that the actual number of iterations needed by the iterative passage time algorithm for the NoBarrier method is precisely the same as for the NoBarrierCheap method in the 1100000 states voting model. Since we only count multiplications with non-zero element in ν_r this implies that ν_r must fill-in faster when using the NoBarrierCheap method in this case. In the web-server model (see fig. C.1) this phenomenon does not occur. The reason for this behaviour can be explained by figs. C.2 and C.3. The ν_r vector in the large voting model fills in more slowly than in the 1000000 states web-server model. Aggregation of states in the voting model may speed-up the vector fill-in and thus cause the increase in the number of multiplications needed for the NoBarrierCheap method.

Another observation we made is that the 2-way Barrier method generally seems to do better than the NoBarrier method. However, the steep increase in the number of complex multiplications needed by the Barrier method between the 500000 and the 1100000 states voting model might highlight a trend that the 2-way Barrier method needs more complex multiplications than the NoBarrier method in large SMPs. Further investigation on larger models is necessary to see if this is a general trend or if it is simply due to the nature of the voting model.

6.2 Path truncation

Recall that the convergence criteria of the iterative passage time analyser as well as the fill-in behaviour of the ν_r vector can cause state and partition aggregation with subsequent passage time analysis on the aggregated graph to be more computationally expensive than the initial passage time analysis. This effect is clearly visible in the graph of the NoBarrierCheap in fig. 6.1. A larger number of complex multiplications obviously yields a higher accuracy when doing the Laplace inversion, however we may not need this extra precision, especially if it only affects the least significant decimal places of our distribution samples. During our analysis of the ν_r vector fill-in we observed that the ν_r vector often contains a high proportion of elements with very small complex values. In the following we study the impact of truncating these elements (i.e. setting them to zero during the iterative passage time analysis) on the accuracy and the performance of our four first-passage time calculation methods introduced in sect. 6.1. Note that this truncation technique can also be used for iterative passage time analysis in Markov models.

Negligibly small Laplace transform sample

Definition 6.1. We define a *negligibly small Laplace transform sample* L to be a complex number L for which $|Re(L)| < \epsilon^2$ and $|Im(L)| < \epsilon^2$ where $\epsilon > 0$ is the precision of the iterative passage time solver in eq. 2.21.

Note. Setting an element in ν_r to zero can create an error that is larger than the absolute value of the truncated element. This is because of the cascading effect of the matrix vector multiplication. Any non-zero element in a non-target column of ν_r contributes to the value of at least one other column in ν_r during the next iteration. As many states have more than one

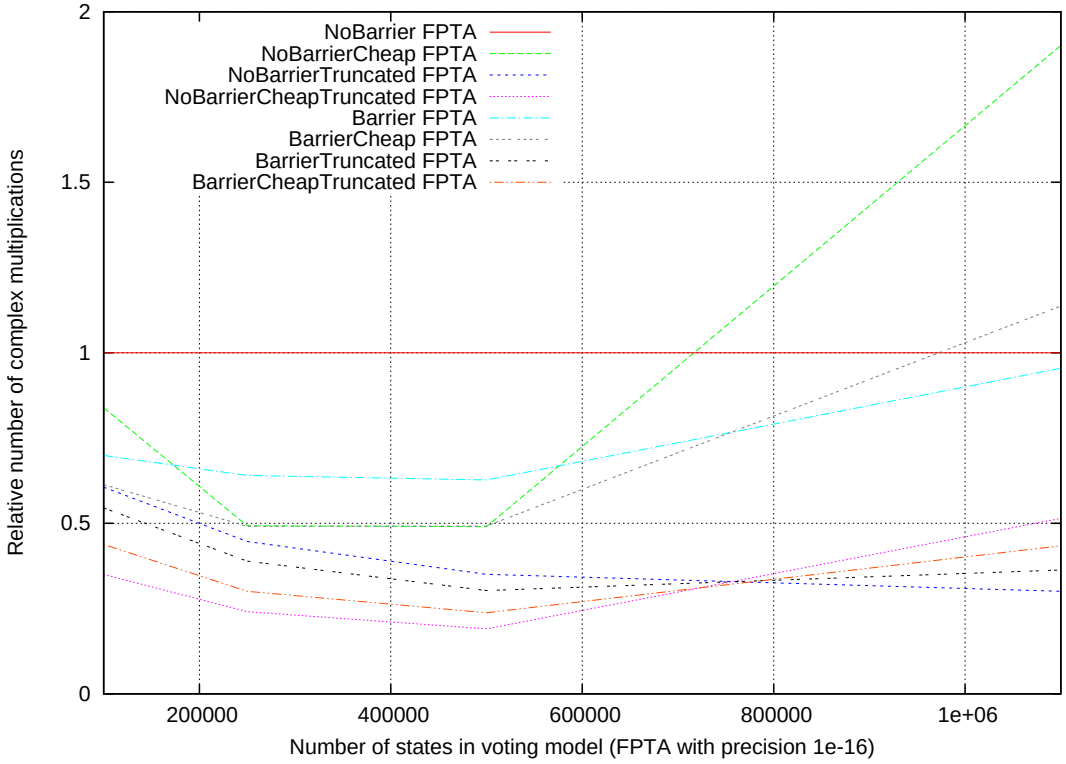


Figure 6.1: The diagram shows how different combinations of aggregation and first-passage time analysis techniques perform relative to the standard iterative first-passage time technique on the voting model. For each model size we divide the number of complex multiplications needed for the first-passage time calculation for a given technique by the number of complex multiplications needed by the standard technique on the unaggregated SMP transition matrix. The first-passage time calculates 165 Laplace transform samples that allow us to estimate a t-point near the mode of the distribution and 2 t-points to either side of that point. See table C.1 for the exact data used to plot this diagram

outgoing transition the value of one element in the ν_r vector usually contributes to the sums of a large percentage of the elements in ν_r since the number of states a single state can reach in k state transitions can be exponentially high. It is thus important to restrict truncation to elements whose absolute values are much smaller than our required precision, otherwise truncation might have a negative impact on the accuracy of the results of the passage time calculation.

6.2.1 Error analysis

The graphs labelled *NoBarrierTruncated, NoBarrierCheapTruncated, BarrierTruncated* and *BarrierCheapTruncated* in fig. 6.1 and fig. C.1 show the performance of the truncation method in the first-passage time analysis of the voting and the web-server model. We chose $\epsilon = 10^{-16}$, hence $\epsilon^2 = 10^{-32}$. As truncation requires us to test all non-zero values of ν_r we decided to remove negligibly small values from ν_r every 25 iterations of the iterative passage time analyser. Comparing the samples of the first-passage time distributions produced by the 4 techniques discussed in sect. 6.1 with those produced by their truncated counterparts we found that they had matching results up to an error term of 10^{-25}. Hence our truncation technique does not seem to have a negative impact on the accuracy of first-passage time distribution.

6.2.2 Performance

Table 6.1 shows that truncation significantly reduces the amount of complex multiplication needed for all 4 different passage time computation techniques. Furthermore we can observe that the saving becomes larger as we increase the size of the model, which suggests that our truncation technique is scalable.

	Voting model			
Number of states	NoBarrier	NoBarrierCheap	Barrier	BarrierCheap
100000	61%	42%	78%	71%
250000	45%	49%	60%	48%
500000	35%	39%	48%	48%
1100000	30%	27%	38%	38%

	Web-server model			
Number of states	NoBarrier	NoBarrierCheap	Barrier	BarrierCheap
100000	36%	37%	44%	45%
250000	33%	34%	41%	42%
500000	30%	32%	38%	39%
1000000	25%	27%	32%	33%

Table 6.1: Relative number of complex multiplications needed by the truncated versions of the FPTA methods compared to their untruncated counterparts.

Table 6.2 shows the different timings we obtained running our first-passage time analyser. Although the *BarrierCheapTruncated* approach has the fastest runtime on the web-server model with 1100000 states, we recommend to use cheap states aggregation with care, since its effect is hard to predict (see sect. 6.1.2). The runtime of the 2-way barrier method on the other hand is always very close to the time needed for the standard calculation. However, at the time we conducted the experiments the test program was not optimised for barrier FPTA.

On average the *NoBarrierTruncated* and *BarrierTruncated* methods yield the highest time saving. This is not surprising as the overhead for removing negligibly small Laplace transform samples is quite low but the saving in complex multiplication is reasonably large (see table C.1). Further improvements on our truncation technique may allow us to reduce its overhead further, so that our time saving matches the saving in complex multiplications more closely. One way to speed up the first-passage time calculation might be to relax defn. 6.1 or to increase the frequency with which we remove negligibly small values from ν_r. The ν_r vector fill-in illustrated in figs. C.2 and C.3 suggests that some models a more suited for truncation than others. As the ν_r vector fills in more slowly in the large voting model than in the web-server model the saving we obtain by truncation is lower than in the web-server model. Consequently the time saving through truncation is greater in the web-server model, too.

One interesting observation we made is that the relative saving in the number of complex multiplication needed by the truncated versions of the FPT analysers appears to be s-point invariant. Further study with rigorous statistical tests is needed to confirm this conjecture, however, if it holds it would enable us to run a pilot study on a single s-point in order to find the optimal configuration for the actual passage time analysis of the t-points we are interested in.

6.2.2.1 FPTA with k-way barrier partitioning

We deliberately postponed the discussion of the k-way barrier partitioning up until now, for the first-passage time analysis using k-way barrier partitionings combines many of the characteristics of techniques that we have discussed so far. First of all we emphasise that the iterative

	Runtime for FPTA in seconds	
Method	Voting model(1100000)	Web-server model(1000000)
NoBarrier	3517	18120
NoBarrierCheap	10319	19762
NoBarrierTruncated	1707	5370
NoBarrierCheapTruncated	4979	7346
Barrier	4218	12820
BarrierCheap	5290	11858
BarrierTruncated	2308	4860
BarrierCheapTruncated	2878	4739

	Relative time compared to NoBarrier	
Method	Voting model(1100000)	Web-server model(1000000)
NoBarrier	100%	100%
NoBarrierCheap	290%	109%
NoBarrierTruncated	48%	30%
NoBarrierCheapTruncated	142%	41%
Barrier	120%	71%
BarrierCheap	150%	65%
BarrierTruncated	66%	27%
BarrierCheapTruncated	82%	26%

Table 6.2: The first table shows the time needed to do a FPTA calculation on an Intel P4 3.0 Ghz for 165 Laplace transforms with precision 10^{-16} and a truncation threshold of 10^{-32}. In the second table we see the relative time needed by methods compared to the standard *NoBarrier* technique.

k-way barrier passage time algorithm (see algo. 3 in sect. 5.3) automatically truncates elements that are no longer needed, i.e. those elements in $\nu_r^{(i)}$ that have no impact on the next restricted iterative first-passage time computation. The second feature of the algorithm is that the fill-in of the $\nu_r^{(i)}$ vector is reduced to those Laplace transform samples that represent the restricted first-passage time distribution from the set of source states to those states that lie in the sub-matrix of the barrier partition on which we perform restricted passage time analysis. As a consequence we see multiple small density peaks in figs. C.2 and C.3. This observation is similar to the one made in fig. 4.4, only that in this case the lower peaks correspond to a reduction in the number of complex multiplications per iteration. Note, however, that we need to compare the total number of complex multiplication to show that the k-way barrier FPTA needs less complex multiplications since the k-way barrier method does significantly more iterations than FPTA methods on the unpartitioned transition graph. Nevertheless the principle of saving memory in exact state aggregation and reducing the amount of complex multiplications in k-way barrier FPTA is the same: by limiting the scope of the computation to the sub-matrix of a partition, the number of new transitions in the case of exact state aggregation as well as the number of Laplace transforms in $\nu_r^{(i)}$ during passage time analysis is physically bounded. In both cases state space partitioning enables us to solve the problem using a less computationally expensive divide and conquer approach.

The results in table 6.3 show that the k-way barrier approach is as least as fast the NoBarrierTrunc method. In the web-server model the k-way barrier passage time analyser is even up to three times faster than the NoBarrierTrunc method. 40-way BarrierTrunc is generally faster than Max-way BarrierTrunc because of the overhead incurred by managing the extra barriers. A comparison between the K–S error of Max-way Barrier and Max-way BarrierTrunc gives more evidence for our earlier conjecture that truncation does not have any significant impact on the accuracy of the passage time analysis. However, it seems that in the voting model the error increases with the number of partitions in the barrier partitioning. For the web-server model

	Voting model(1100000)		
Method	Complex mults	Runtime(secs)	K-S error
NoBarrier	91,067,403,088	5317	0
NoBarrierTrunc	27,362,071,935	1707	0
2-way BarrierTrunc	33,038,568,429	2308	8.18789e-13
40-way BarrierTrunc	20,631,960,444	1630	1.25518e-12
Max-way Barrier	14,675,308,020	2110	9.81359e-12
Max-way BarrierTrunc	14,613,972,603	1936	9.81359e-12
	Web-server model (1000000)		
Method	Complex mults	Runtime(secs)	K-S error
NoBarrier	287,181,545,505	18120	0
NoBarrierTrunc	75,954,719,825	5370	0
2-way BarrierTrunc	52,391,817,571	4860	2.81538e-13
40-way BarrierTrunc	14,826,831,044	1338	1.55187e-12
Max-way Barrier	17,070,767,235	1955	1.48844e-12
Max-way BarrierTrunc	10,733,105,688	1545	1.48844e-12

Table 6.3: Timings were done on a Intel P4 3.0 Ghz with 4 GByte of RAM. Note that the runtime was not timed on a dedicated machine. This is probably the reason why *Max-way Barrier* takes longer than its truncated version despite the number of multiplications being almost identical. In the voting model the Max-way barrier partitioning corresponds to a 349-way partitioning, in the web-server model to a 332-way partitioning.

on the other hand this does not hold. If too many barrier partitions were to cause numerical instability in the iterative passage time analysis, then we would either have to generate a k_{max}-way partitioning and subsequently join neighbouring partitions in order to reduce the number of partitions or impose a stronger convergence criteria for the iterative solver. Given the data in table 6.3 this is just mere speculation though, especially because the K–S errors are not that much larger than 10^{-16}, which is the convergence criteria of the iterative solver we use throughout all experiments in this chapter. Finally note that there might potentially be a correlation between how efficient truncation can be applied to a model and how fast k-way barrier passage time analysis is. Such a dependence would explain why k-way barrier passage time analysis is much faster than the NoBarrierTrunc method in the web-server model than it is in the voting model, since we already observed in sect. 6.2.2 that truncation works better on the web-server model than on the voting model.

6.3 Parallelisation

In [15] the parallelisation of the standard iterative passage time analysis is discussed. As we mentioned in sect. 5.3 the principle could be extended to work for k-way barrier passage time analysis, too. Parallelisation becomes harder though, when the iterative passage time analyser is to be used in conjunction with truncation of negligibly small values. Existing load balancing schemes might perform poorly when applied to parallel iterative FPTA with truncation as they only consider the sparsity of the transition matrix but not the sparsity of the ν_r vector. As the sparsity of the ν_r vector changes during iterative passage time analysis, load balancing is potentially more difficult than for general sparse matrix vector multiplication. One way to address this problem might be to use a probabilistic load balancing scheme, which assigns weights to states in ν_r not only dependent on how many outgoing transitions the corresponding state has, but also dependent on how likely it is to be non-zero during the iterative passage time analysis. If the beforementioned invariability of the relative effect of truncation held across all s-points then a pilot study on a single s-point could be used to optimise the load balancing for different partitions. Another way of determining states that are less likely to be truncated would be to use the steady state distribution to infer the computational load of a single state

during iterative passage time analysis with truncation. Further research might give us a better understanding of the fill-in behaviour of the ν_r vector. This knowledge could then be used to optimise load balancing for parallel iterative FPTA with truncation.

6.4 Summary

All passage time analysis techniques discussed in this chapter were shown to be exact. Moreover applying truncation and k-way barrier partitioning speeds up the iterative passage time analysis in the voting and the web-server model significantly. The main questions that are left unanswered are whether Max-way BarrierTrunc is numerically stable and how aggressive truncation can be applied without causing numerical errors. All in all the results in this chapter leave us with the impression that truncation and k-way barrier partitioning could become enabling strategies for passage time evaluation of massive Markov and semi-Markov models that are computationally intractable when iterative passage time analysis is performed on the unaggregated state space.

CHAPTER 7

Evaluation, conclusion and further work

7.1 Evaluation

When used in combination with k-way barrier partitioning and truncation our sparse vector implementation of the passage time analyser is a lot faster than SMARTA. On an Intel Core Duo 2.66 Ghz the passage time analysis with 165 s-points on the 1100000 states voting model and the 1000000 states web-server model took 5475, 10024 seconds respectively in SMARTA. With our new k-way barrier truncation algorithm the same calculations took 2053 seconds on the voting and 2168 seconds on the web-server model. This evaluates to a speed-up of roughly 2.5 in the voting model and 5 in the web-server model. We have thus shown that our new partitioning and passage time analysis techniques are indeed improving existing passage time evaluation methods.

Due to the limited time available for this study we only used the Kolmogorov-Smirnov statistic for our error analysis. We are, however, confident that results concerning the accuracy of cheap state aggregation, k-way barrier partitioning and truncation are correct. The conjecture about the s-point invariability regarding the relative saving in complex multiplications through truncation (c.f. sect. 6.2) still requires thorough validation. It should also be noted that our passage time analyser was not implemented using the memory saving feature of k-way barrier truncation. The reason this was not done is that the k-way barrier method was only developed towards the end of the project and hence there was no time to rewrite the analyser. However from algo. 3 it can be seen that for every iteration of the passage time analyser we only have to hold the states and transitions of the current barrier partition in memory. Thus although the memory saving has not been shown to work in an actual implementation there is no reason to believe that it does not work in practice.

7.2 Conclusion

Atomic partition aggregation is a lot more feasible than exact state-by-state aggregation, even if the latter is done on partitions of the transition matrix rather than on the flat transition matrix as in [1]. Moreover our study shows that state space partitioning of the transition graph of semi-Markov processes significantly decreases the amount of memory and time needed for the computation of passage time distributions. To find suitable partitions for atomic aggregation in larger SMPs, graph and hypergraph partitioners do not seem to be a good choice. The barrier partitioning example, however, illustrates that the structure of the semi-Markov chains can be exploited to develop better partitioning methods tailored for the partitioning of the transition matrix prior to passage time analysis. There are potentially further partitioning concepts,

which can be used for state aggregation prior to the calculation of other performance metrics (see sect. 2.5).

7.3 Further work

The first list depicts research projects whose results would consolidate the theoretical and practical results of this study. It would be especially useful to

1. show that the theoretical memory reductions in passage-time analysis with k-way barrier partitioning can be achieved in practice
2. test the new passage time analysis techniques and the barrier partitioning algorithm on larger SMP models with several million states
3. investigate the numerical stability of Max-way Barrier partitioning for passage time analysis
4. show that the relative saving in the number of complex multiplications is s-point invariable
5. investigate different models to determine characteristics of models that gain most from the application of the truncation technique

Further interesting research could also be done to investigate

1. the use of cheap state aggregation in combination with k-way barrier partitioning and truncation
2. how parallel passage time analysis can be optimised for the new partitioning and truncation techniques
3. partitioning and aggregation techniques for improved computation of other performance metrics
4. how techniques developed in this study can be used for the analysis of Markov models

7.3.1 Building the billion state semi-Markov response time analyser

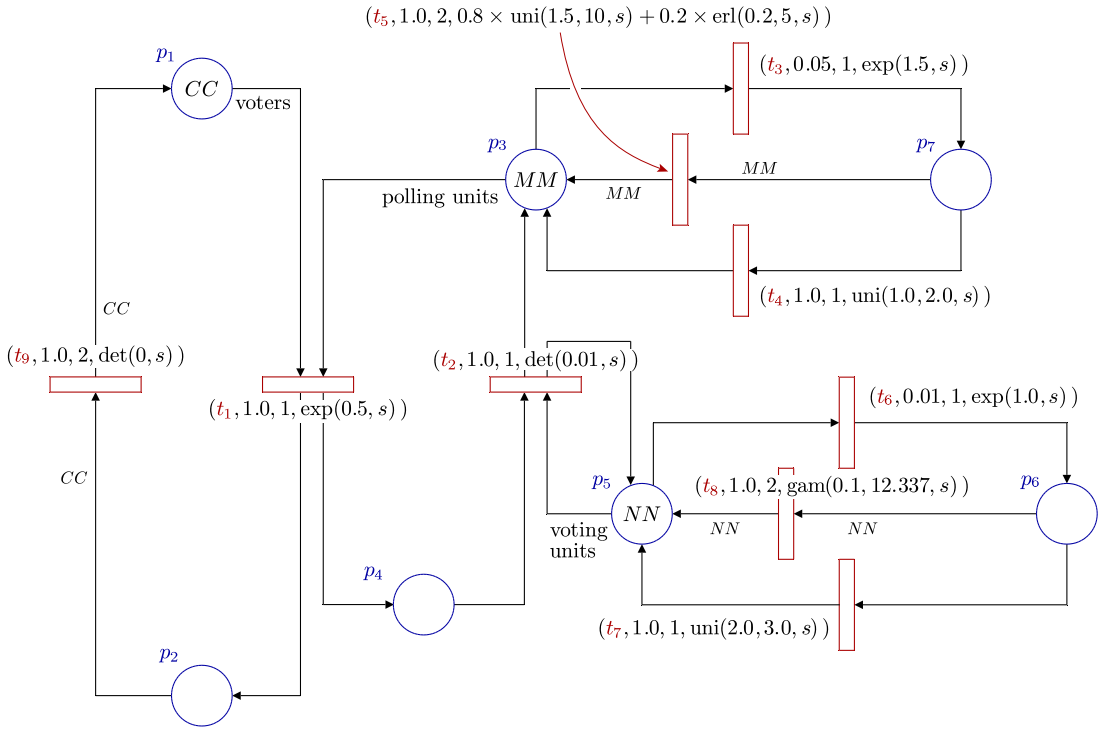
In view of the new techniques for exact first-passage time analysis it is worth considering to create a new semi-Markov response time analyser tool chain. Although barrier partitioning and passage time analysis seem to be scalable, there are various other challenges when computing passage times in extremely large SMPs, i.e. chains with more than 100 million states. Provided that extremely large transition matrices can be generated using parallel breadth first search, there still remains the problem of doing functional analysis and steady-state probability computation on such huge state spaces. Especially with regard to the steady-state vector computation, which is needed in order to compute the α vector (c.f. eq. 2.10), the development of new partitioning and aggregation techniques might improve existing evaluation techniques described in [22, 23, 35]. The construction of the tool chain is desirable as current solvers can only approximate passage time distributions in extremely large semi-Markov chains.

APPENDIX A

Models studied

This appendix detailly specifies the 3 Petri net models that we use to test and verify our aggregation and passage time computation methods. The first model is a SM-SNP for an electronic voting model, the second one a SM-SNP for a parallel web-server and the third model is a GSPN of the courier communication protocol. For information on Petri nets see sect. 2.2.

A.1 Voting model



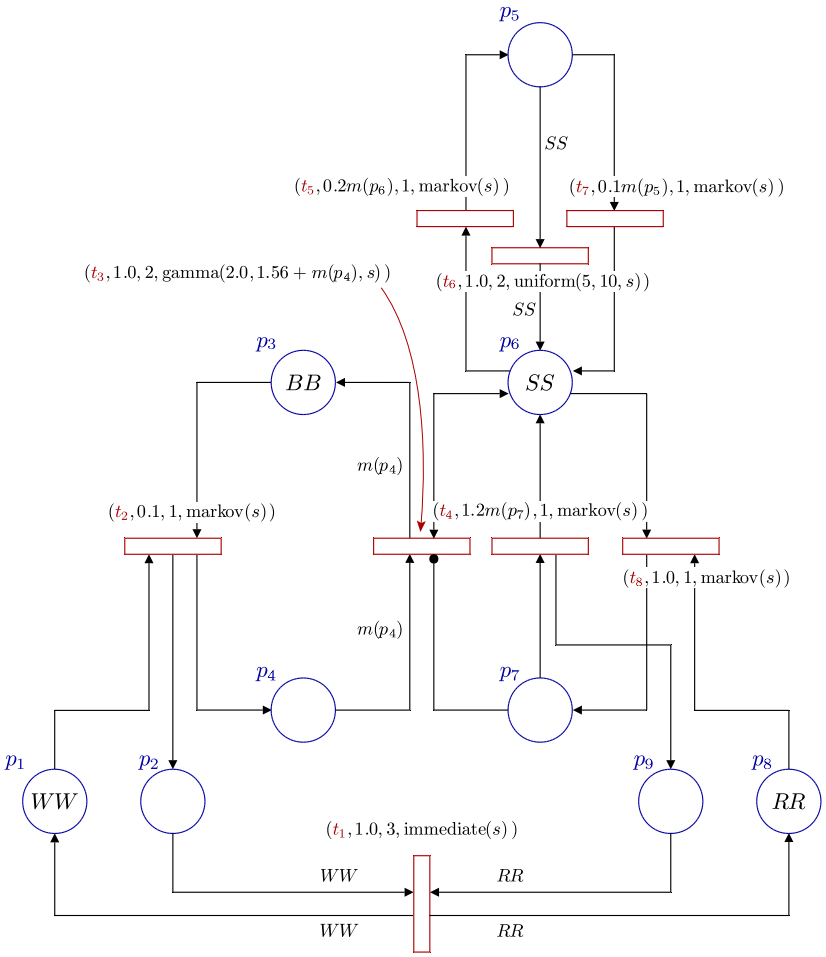
The SM-SPN described in sect. A.1 shows a model of a distributed voting system[9, 22]. The model has CC voters, MM polling units and NN central voting units that gather the votes from the polling units. Voters vote asynchronously, moving from p_1 to p_2 as they cast their

vote. To ensure that each voter can only cast one vote, transition t_9 is only enabled when all CC voters have casted their vote. Voting can only occur in abundance of a free polling unit in p_3. Having been used by a voter the polling unit sends the vote to one of the NN central voting units. If there is no such voting unit the polling unit waits in p_4. Once it has submitted its vote to the server the voting unit becomes operational again. When polling units fail they enter p_7 via transition t_3 where they remain until they have been repaired. Similarly broken central voting units wait in p_6. The passage time analysis conducted on this model in our study investigates the distribution of time needed for CC voters to cast their vote in a system with MM polling units and NN central servers.

CC	MM	NN	States	Transitions
22	7	4	4050	16128
22	12	4	10300	43608
60	25	4	106540	480000
100	30	4	249760	1140000
125	40	4	541280	2500000
175	45	5	1140050	5512500

Table A.1: Size of SMP generated by different configurations of the voting model.

A.2 Web-content authoring (web-server) model

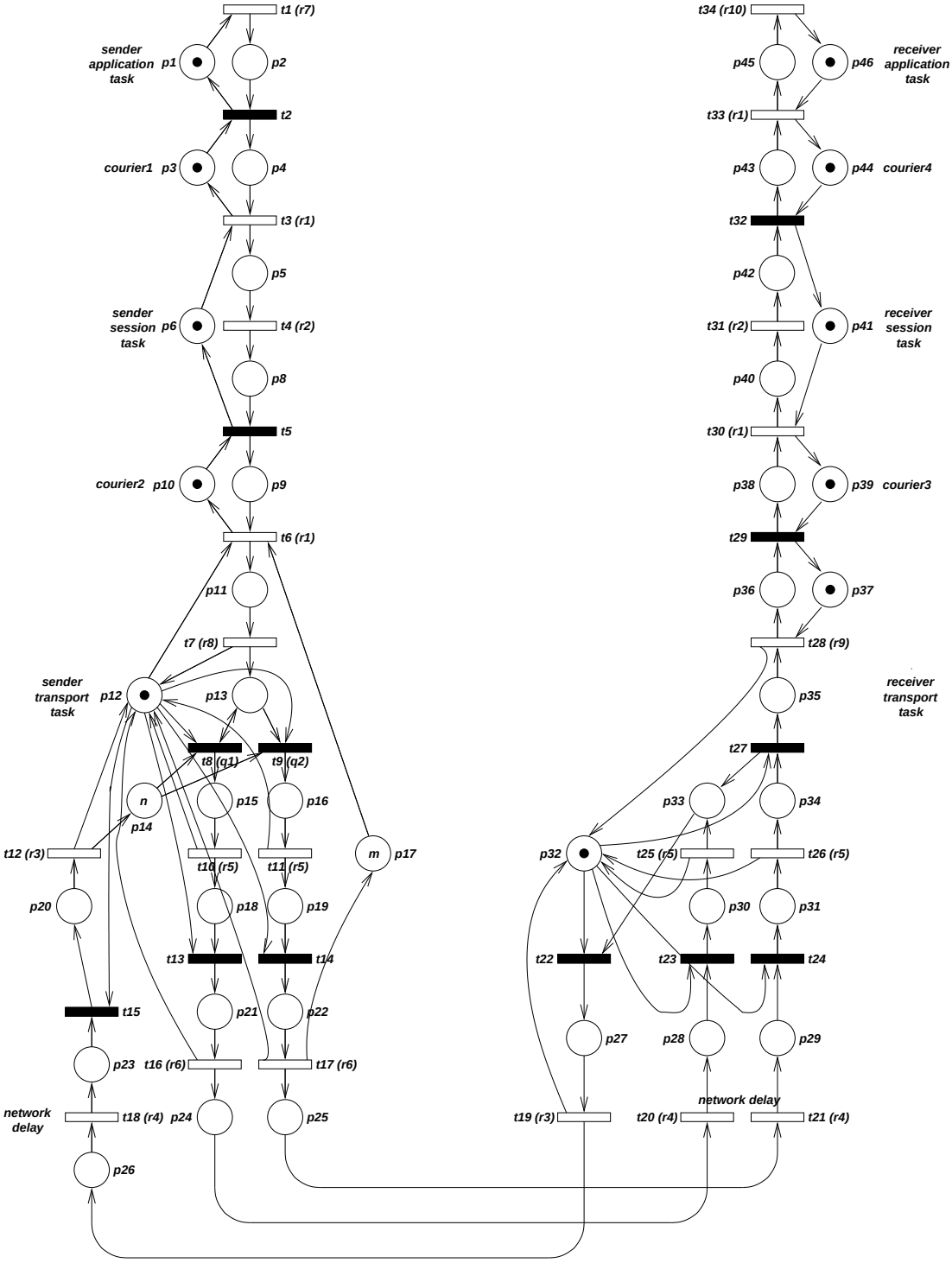


The SM-SPN described in sect. A.2 shows a model of a web-server with RR clients (readers), WW web content authors (writers), SS parallel web-servers and a write-buffer of size $BB[9, 22]$. Readers that request web-pages from one of the web-servers move from p_8 to p_7 and from p_7 to p_9 as they receive the requested content. Writers who have completed a new web-page submit it to the write buffer. This is represented by a token movement from p_1 to p_3 via p_4, which also requires them to use one of the servers in p_6. Write requests are granted if there is no read request in p_7. Once a write request has been processed it moves to place p_2. The web-servers in p_6 are liable to fail. If they do, they move to p_5 where they remain until they have been fixed. For this model the passage time analysis we measured in our experiments represents the probability distribution of the time needed until all RR read and WW write requests have been processed in a system with SS web-servers and a write buffer of size BB.

RR	WW	SS	BB	States	Transitions
45	22	4	8	107289	319164
66	33	4	8	249357	743272
94	45	4	8	498433	1487432
130	64	4	8	1002752	2994732

Table A.2: Size of SMP generated by different configurations of the web-server model.

A.3 Courier model



The GSPN model described in sect. A.3 represents the ISO Application, Session and Transport layers of the Courier sliding-window communication protocol. It was originally presented in [32]. For a detailed explanation see [22]. The model has 29010 states and 65640 transitions.

APPENDIX B

Additional diagrams for barrier partitioning discussion

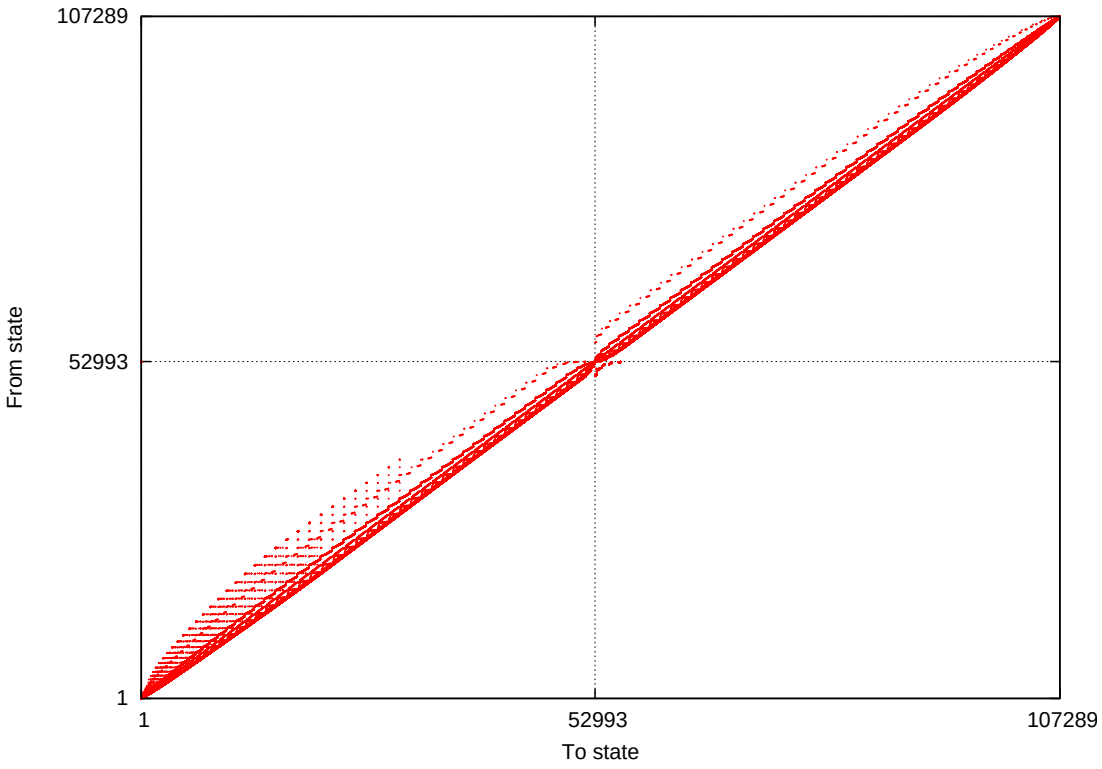


Figure B.1: Balanced barrier partitioning of 107289 states web-server model.

APPENDIX C

Additional diagrams for FPTA performance discussion

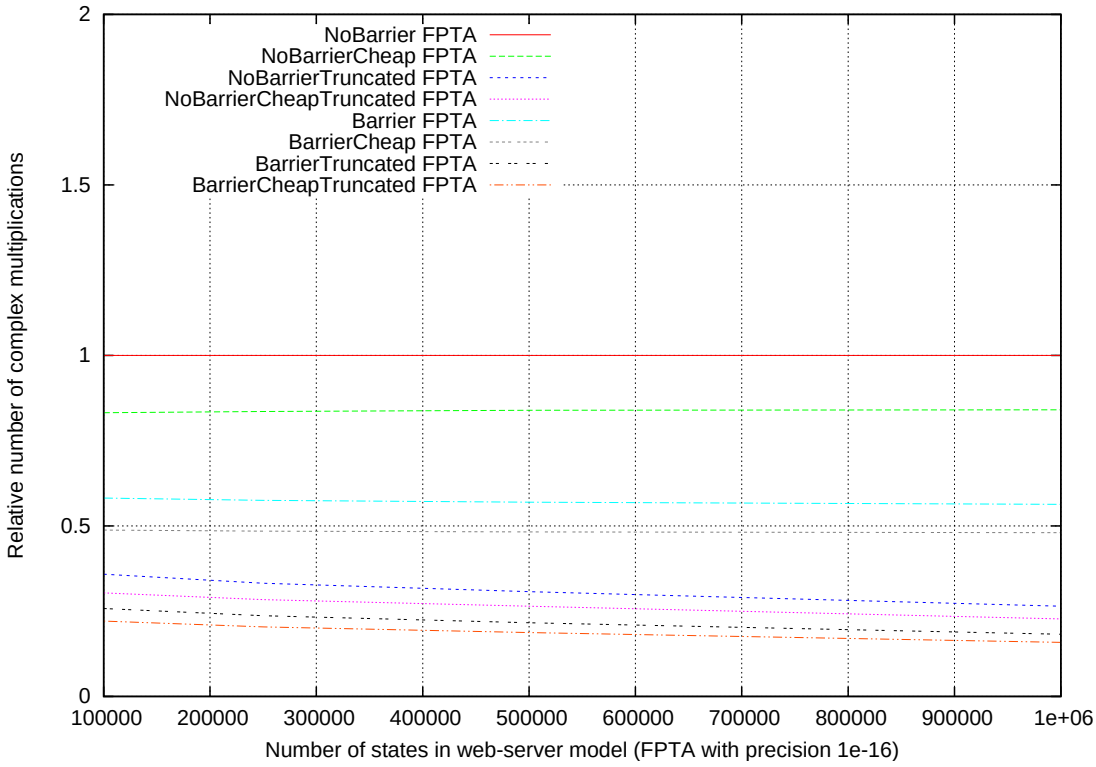


Figure C.1: The diagram shows how different combinations of aggregation and first-passage time analysis techniques perform relative to the standard iterative first-passage time technique on the web-server model. For each model size we divide the number of complex multiplications needed for the first-passage time calculation for a given technique by the number of complex multiplications needed by the standard technique on the unaggregated SMP transition matrix. The first-passage time calculates 165 Laplace transform samples that allow us to estimate a t-point near the mode of the distribution and 2 t-points to either side of that point.

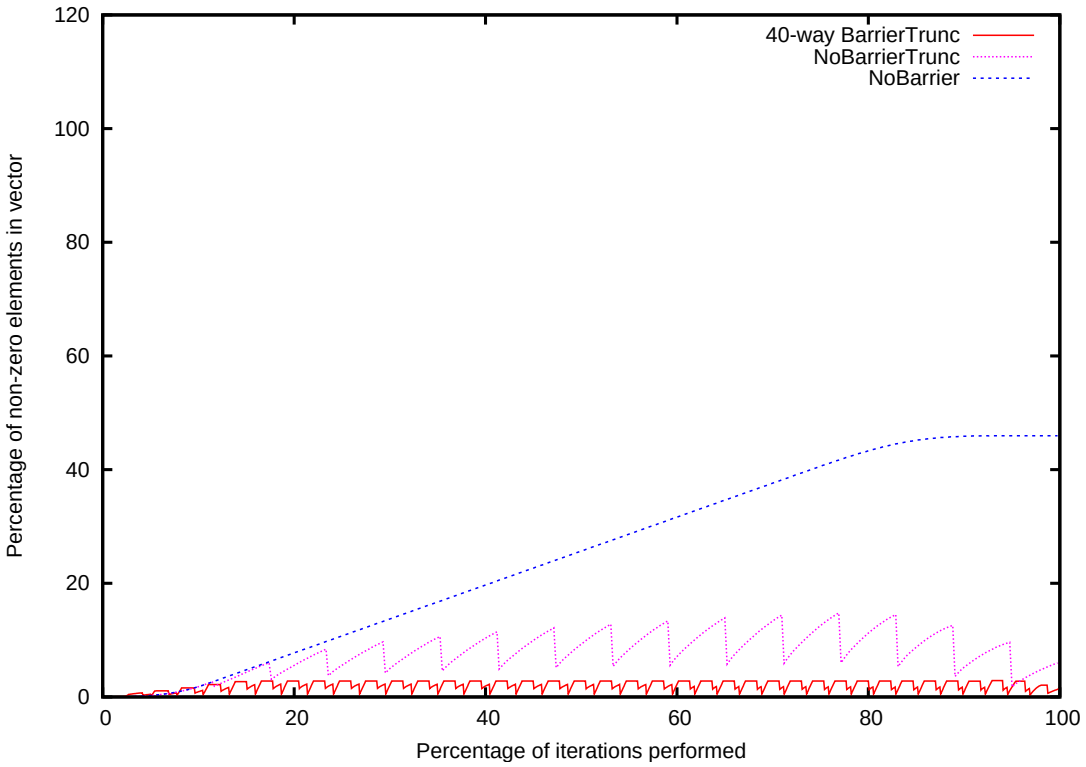


Figure C.2: Sparsity of ν_r vector during the iterative passage time analysis of the 1100000 state voting model. The data is based on the vector fill-in observed during the iterative passage time computation of a single s-point. Although the exact pattern differs slightly between different s-points, the general trends were the same. Therefore the data is representative of the behaviour of the ν_r vector in the large voting model.

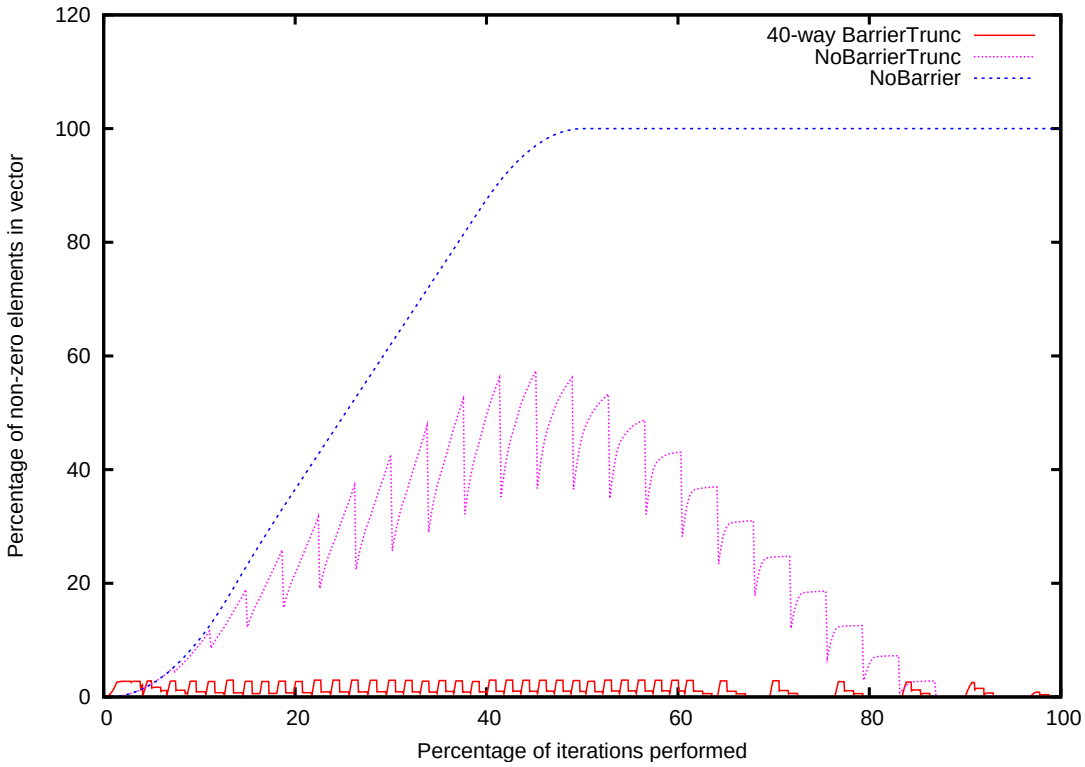


Figure C.3: Sparsity of ν_r vector during the iterative passage time analysis of the 1000000 state web-server model. The data is based on the vector fill-in observed during the iterative passage time computation of a single s-point. Although the exact pattern differs slightly between different s-points, the general trends were the same. Therefore the data is representative of the behaviour of the ν_r vector in the large web-server model.

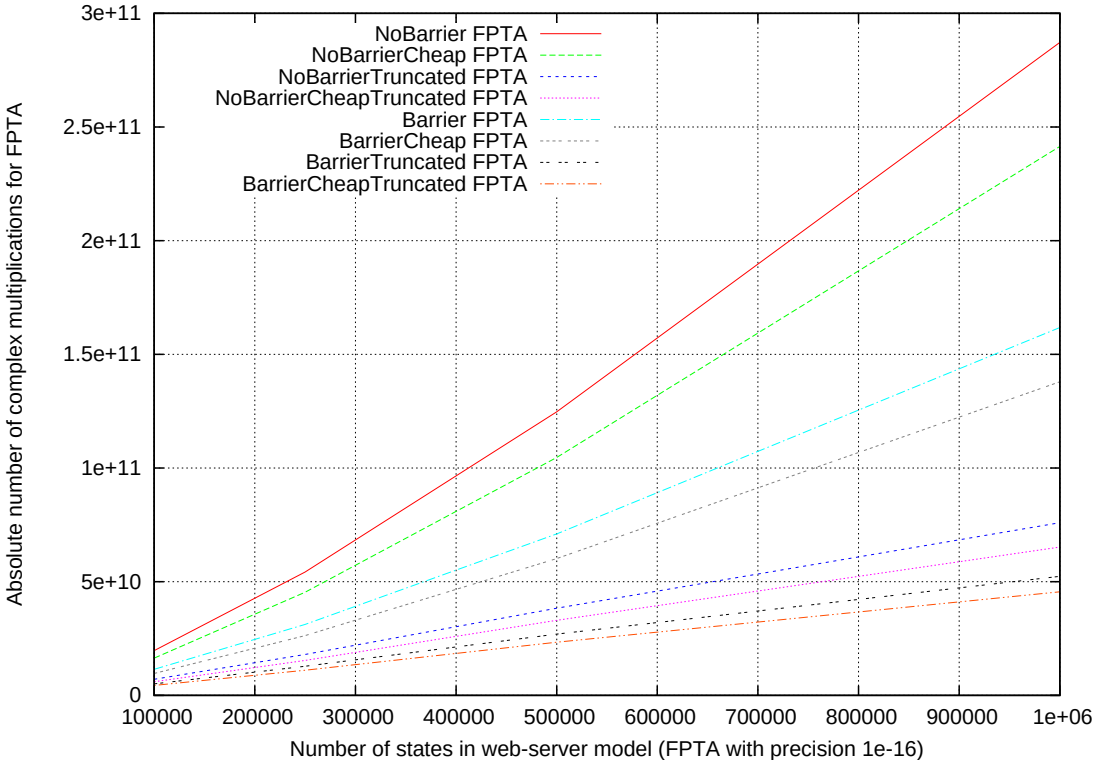


Figure C.4:

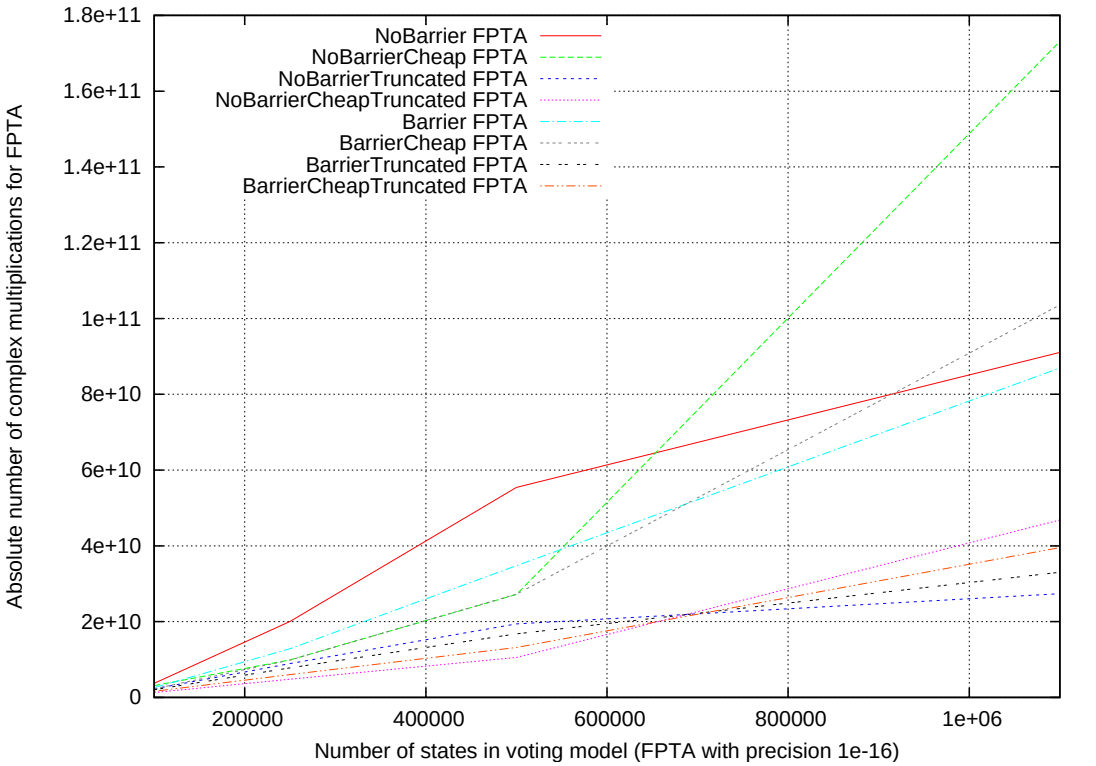


Figure C.5:

	Voting model							
	Untruncated				Truncated			
Number of states	NoBarrier	NoBarrierCheap	Barrier	BarrierCheap	NoBarrier	NoBarrierCheap	Barrier	BarrierCheap
100000	100%	84%	70%	61%	61%	35%	54%	44%
250000	100%	49%	64%	49%	45%	24%	39%	30%
500000	100%	49%	63%	49%	35%	19%	30%	24%
1100000	100%	190%	95%	113%	30%	51%	36%	43%

	Web-server model							
	Untruncated				Truncated			
Number of states	NoBarrier	NoBarrierCheap	Barrier	BarrierCheap	NoBarrier	NoBarrierCheap	Barrier	BarrierCheap
100000	100%	83%	58%	49%	36%	30%	26%	22%
250000	100%	84%	57%	48%	33%	28%	24%	20%
500000	100%	84%	56%	48%	31%	26%	22%	19%
1000000	100%	84%	56%	48%	26%	23%	18%	16%

Table C.1: Table contains numerical data used to plot figs. 6.1 and C.1. To get this data we simply divided the number of of complex multiplication needed for the FPTA with a particular technique by the number of complex multiplications needed for the *NoBarrier* method.

Bibliography

[1] J. T. Bradley, N. J. Dingle, W. J. Knottenbelt: Exact Aggregation Strategies for Semi-Markov Performance Models, SPECTS 2003, International Symposium on Performance Evaluation of Computer and Telecommunication Systems, Montreal, Canada, July 20-24 2003

[2] B. Hendrickson, T. G. Kolda: Graph partitioning models for parallel computing, Parallel Computing, v.26 n.12, p.1519-1534, Nov. 2000

[3] B. Hendrickson: Graph Partitioning and Parallel Solvers: Has the Emperor No Clothes? (Extended Abstract), Proceedings of the 5th International Symposium on Solving Irregularly Structured Problems in Parallel, p.218-225, August 09-11, 1998

[4] U. Catalyurek, C. Aykanat: Hypergraph-Partitioning-Based Decomposition for Parallel Sparse-Matrix Vector Multiplication, IEEE Transactions on Parallel and Distributed Systems, v.10 n.7, p.673-693, July 1999

[5] B. Ucar, C. Aykanat: Revisiting Hypergraph Models for Sparse Matrix Partitioning, SIAM Rev. Volume 49, Issue 4, pp. 595-603, Nov. 2007

[6] U. Catalyurek, C. Aykanat: A Fine-Grain Hypergraph Model for 2D Decomposition of Sparse Matrices, Proceedings of the 15th International Parallel & Distributed Processing Symposium, p.118, April 23-27, 2001

[7] A. Trifunovic, W. J. Knottenbelt: Parallel multilevel algorithms for hypergraph partitioning, May 2008, Journal of Parallel and Distributed Computing, Volume 68 Issue 5, Publisher: Academic Press, Inc.

[8] P. G. Harrison, W. J. Knottenbelt: Passage time distributions in large Markov chains, Proceedings of the 2002 ACM SIGMETRICS international conference on Measurement and modeling of computer systems, June 15-19, 2002, Marina Del Rey, California

[9] J. T. Bradley, N. J. Dingle, W. J. Knottenbelt, H. J. Wilson: Hypergraph-based parallel computation of passage time densities in large semi-Markov models, J. Linear Algebra Appl. 386 (2004) 311-334.

[10] B. Vastenhouw, R. H. Bisseling: A Two-Dimensional Data Distribution Method for Parallel Sparse Matrix-Vector Multiplication, SIAM Review, v.47 n.1, p.67-95, 2005

[11] J. T. Bradley, D. V. de Jager, W. J. Knottenbelt, A. Trifunovic: Hypergraph partitioning for faster parallel PageRank computation, LECT NOTES COMPUT SC, 2005, Vol: 3670, Pages: 155 - 171, ISSN: 0302-9743

[12] G. Karypis , V. Kumar: Multilevel k-way hypergraph partitioning, Proceedings of the 36th ACM/IEEE conference on Design automation, p.343-348, June 21-25, 1999, New Orleans, Louisiana, United States

[13] C. M. Fiduccia, R. M. Mattheyses: A linear-time heuristic for improving network partitions, Proceedings of the 19th conference on Design automation, p.175-181, January 1982

[14] A. Trifunovic, W. J. Knottenbelt: Parkway 2.0: a parallel multilevel hypergraph partitioning tool. In: Proceedings of the 19th International Symposium on Computer and Information Sciences, Lecture Notes in Computer Science, vol. 3280. Springer, Berlin. pp. 789-800.

[15] J. T. Bradley, N. J. Dingle, P. G. Harrison, W. J. Knottenbelt: Distributed Computation of Passage Time Quantiles and Transient State Distributions in Large Semi-Markov Models, Proceedings of the 17th International Symposium on Parallel and Distributed Processing, p.281.1, April 22-26, 2003

[16] C. J. Alpert, A. B. Kahng: Recent directions in netlist partitioning: a survey, Integration, the VLSI Journal, v.19 n.1-2, p.1-81, Aug. 1995

[17] C. J. Alpert, A. B. Kahng: Multi-way partitioning via spacefilling curves and dynamic programming, Proceedings of the 31st annual conference on Design automation, p.652-657, June 06-10, 1994, San Diego, California, United States

[18] G. Karypis, R. Aggarwal, V. Kumar, S. Shekhar: Multilevel hypergraph partitioning: application in VLSI domain, Proceedings of the 34th annual conference on Design automation, p.526-529, June 09-13, 1997, Anaheim, California, United States

[19] C. J. Alpert, J. Huang, A. B. Kahng: Multilevel circuit partitioning, Proceedings of the 34th annual conference on Design automation, p.530-533, June 09-13, 1997, Anaheim, California, United States

[20] P. P. G. Dyke: An introduction to Laplace transforms and Fourier series, Springer Verlag London Limited 2001, 2nd printing 2001

[21] DNAMaca: http://www.doc.ic.ac.uk/ipc/, accessed on the 3$^{\text{rd}}$ of November 2008

[22] N. J. Dingle: Parallel Computation of Response Time Densities and Quantiles in Large Markov and Semi-Markov Models, PhD thesis, Imperial College, London, United Kingdom, February 2004.

[23] W. J. Knottenbelt: Parallel Performance Analysis of Large Markov Models, PhD thesis, Imperial College, London, United Kingdom, February 2000.

[24] MeTiS/ParMeTiS graph partitioners and hMeTiS hypergraph partitioner `http://www.cs.umn.edu/~karypis/metis`, accessed on the 7$^{\text{th}}$ of January 2009

[25] PaToH hypergraph partitioning software `http://bmi.osu.edu/~umit/software.html`, accessed on the 28$^{\text{th}}$ of December 2008

[26] R. Neapolitan: Probabilistic Reasoning in Expert Systems, John Wiley 1990

[27] M. C. Guenther, N. J. Dingle, J. T. Bradley, W. J. Knottenbelt: Aggregation Strategies for Large Semi-Markov Processes, III International Symposium on Semi-Markov Models: Theory & applications, June 2009

[28] M. C. Guenther, N. J. Dingle, J. T. Bradley, W. J. Knottenbelt: Truncation of Passage Time Calculations in Large Semi-Markov models, 25th UK Performance Engineering Workshop, to appear in July 2009

[29] http://en.wikipedia.org/wiki/Kolmogorov-Smirnov, accessed on 04/06/2009 at 4.30pm

[30] http://www.sgi.com/tech/stl/, accessed on 05/06/2009 at 12pm

[31] U.S. Fire Administration/National Fire Data Center: Structure Fire Response Times, Topical Fire Research Series, Volume 5 Issue 7 January 2006 / Revised August 2006, http://www.usfa.dhs.gov/downloads/pdf/tfrs/v5i7.pdf, accessed on 06/06/2009 at 11pm

[32] C. M. Woodside, Y. Li: Performance Petri net analysis of communcation protocol software by delay-equivalent aggregation. In Proceedings of the 4th International Workshop on Petri nets and Performance Models(PnPM'91), pages 64-73, Melbourne, Australia, 2-5 December 1991, IEEE Computer Society Press

[33] N. J. Dingle, P. G. Harrison, W. J. Knottenbelt: HYDRA: HYpergraph-based Distributed Response-time Analyser. In Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'03), pages 215-219, Las Vegas NV, USA, June 23rd-26th 2003

[34] J. T. Bradley, H. J. Wilson: Iterative convergence of passage-time densities in semi-Markov performance models, Performance Evaluation, Volume 60, Issues 1-4, Performance Modeling and Evaluation of High-Performance Parallel and Distributed Systems, May 2005, Pages 237-254,

[35] R. Mehmood: Disk-based techniques for efficient solution of large Markov chains, Ph.D. thesis, University of Birmingham., October 2004

Index