Tackling Large State Spaces in Performance Modelling

William Knottenbelt, Jeremy Bradley, Nicholas Dingle,
Peter Harrison, Aleksandar Trifunovic

{wjk,jb,njd200,pgh,at701}@doc.ic.ac.uk
Department of Computing
Imperial College London
United Kingdom

SFM-07 Bertinoro, Italy 1 June 2007

The rise of system complexity

- Systems are no longer controlled by single-flow-of-control programs running on single machines
- Increasing use of technologies like multithreading and parallel and distributed programming
- Modern systems are complex webs of cooperating subsystems implying:
  - millions or billions of possible system configurations
  - massive potential for bugs caused by subtle interactions
- Challenge: How can we rigorously guarantee satisfactory system operation in the face of this **state explosion**?

The triumph of formal methods over ad hoc testing

- Thorough code reviews and ad hoc testing are not enough
- e.g. You might think file systems are highly reliable and thoroughly tested, right?

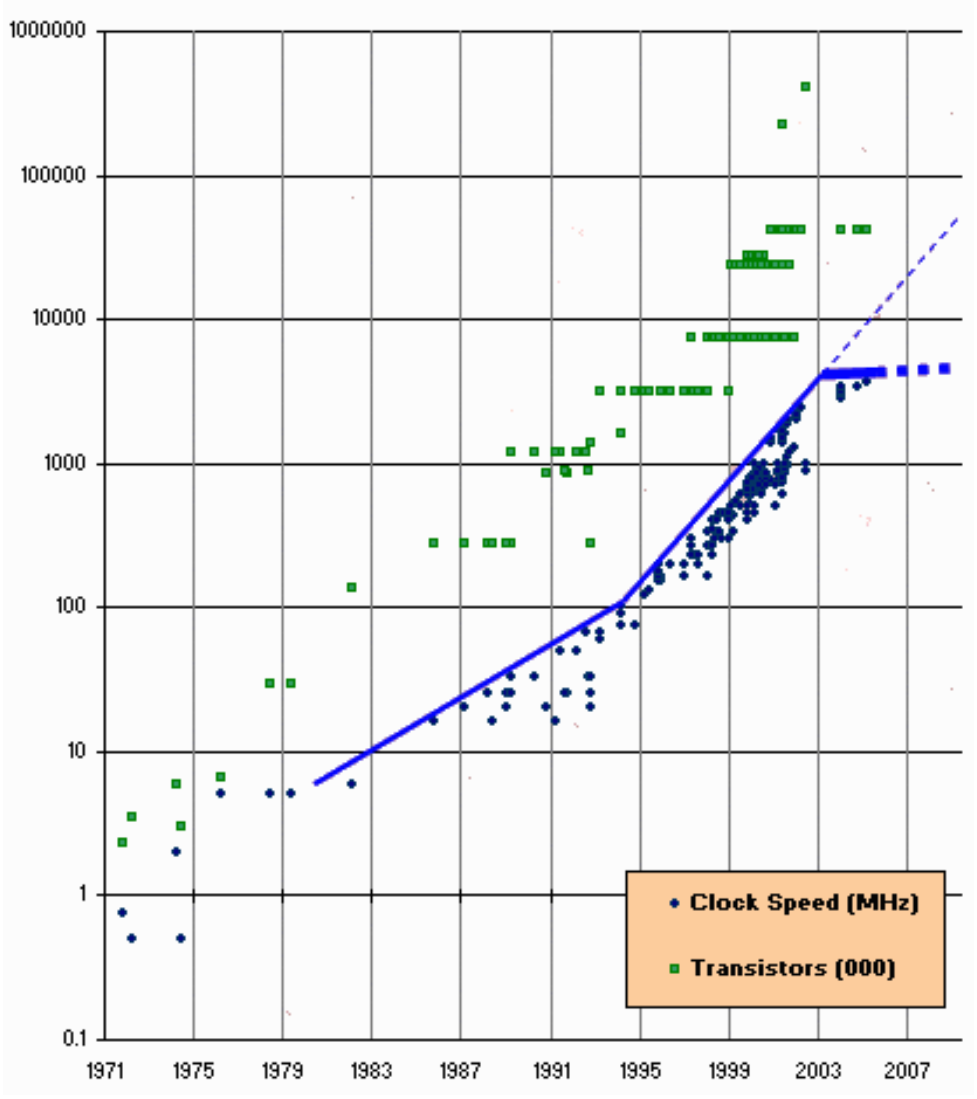
The triumph of formal methods over ad hoc testing

- Thorough code reviews and ad hoc testing are not enough
- e.g. You might think file systems are highly reliable and thoroughly tested, right?
- Recent work on model-checking of file system code by Engler et al (OSDI '06) uses simple BFS exploration of all possible execution paths and failure points
- 18 (serious and hitherto undiscovered) errors uncovered in the 10 widely-used file systems tested
- Formal methods can also be brought to bear on problem of ensuring adequate system performance

Performance requirements are getting more sophisticated

- No longer primarily concerned with simple steady-state resource-based measures like mean utilization
- Service Level Agreements featuring more sophisticated performance requirements based on passage times and transients abound
- Used by hospitals, emergency services, research councils, postal services, industry-benchmarks etc.

CPU Speed


1000000
100000
10000
1000
100
10
1
0.1
1971
1975
1979
1983
1987
1991
1995
1999
2003
2007
Clock Speed (MHz)
Transistors (000)

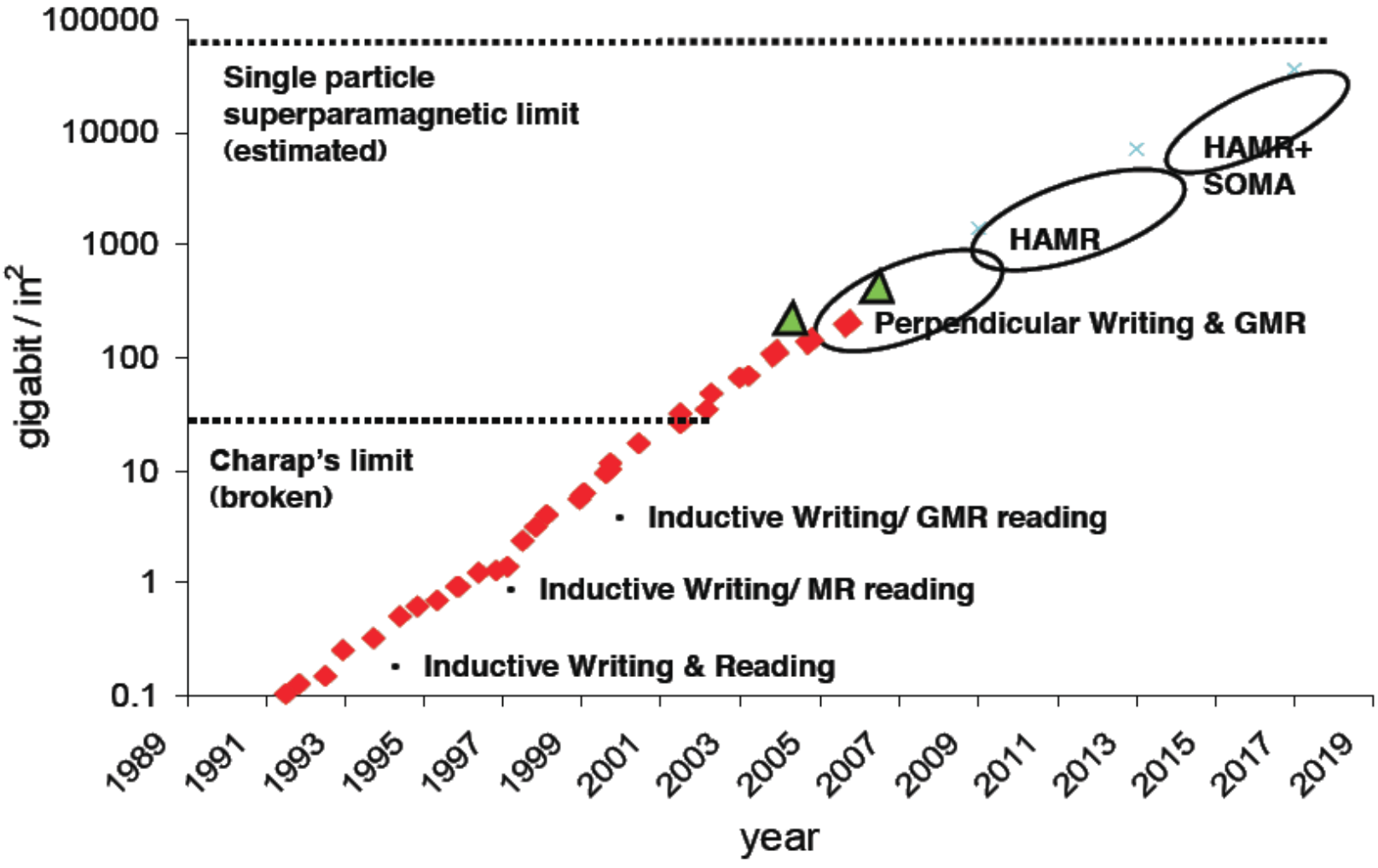
CPU Speed

- “The free performance lunch is over” thanks to physical barriers
- Multi-core processors widely available:
  - Dual-core processors common
  - IBM/Sony Cell, Sun Niagara have 8 cores
  - Intel has released 80-core prototype

CPU Speed

- “The free performance lunch is over” thanks to physical barriers
- Multi-core processors widely available:
  - Dual-core processors common
  - IBM/Sony Cell, Sun Niagara have 8 cores
  - Intel has released 80-core prototype
- Concurrent/parallel programming is the next programming revolution

Drive capacity (Seagate projections)

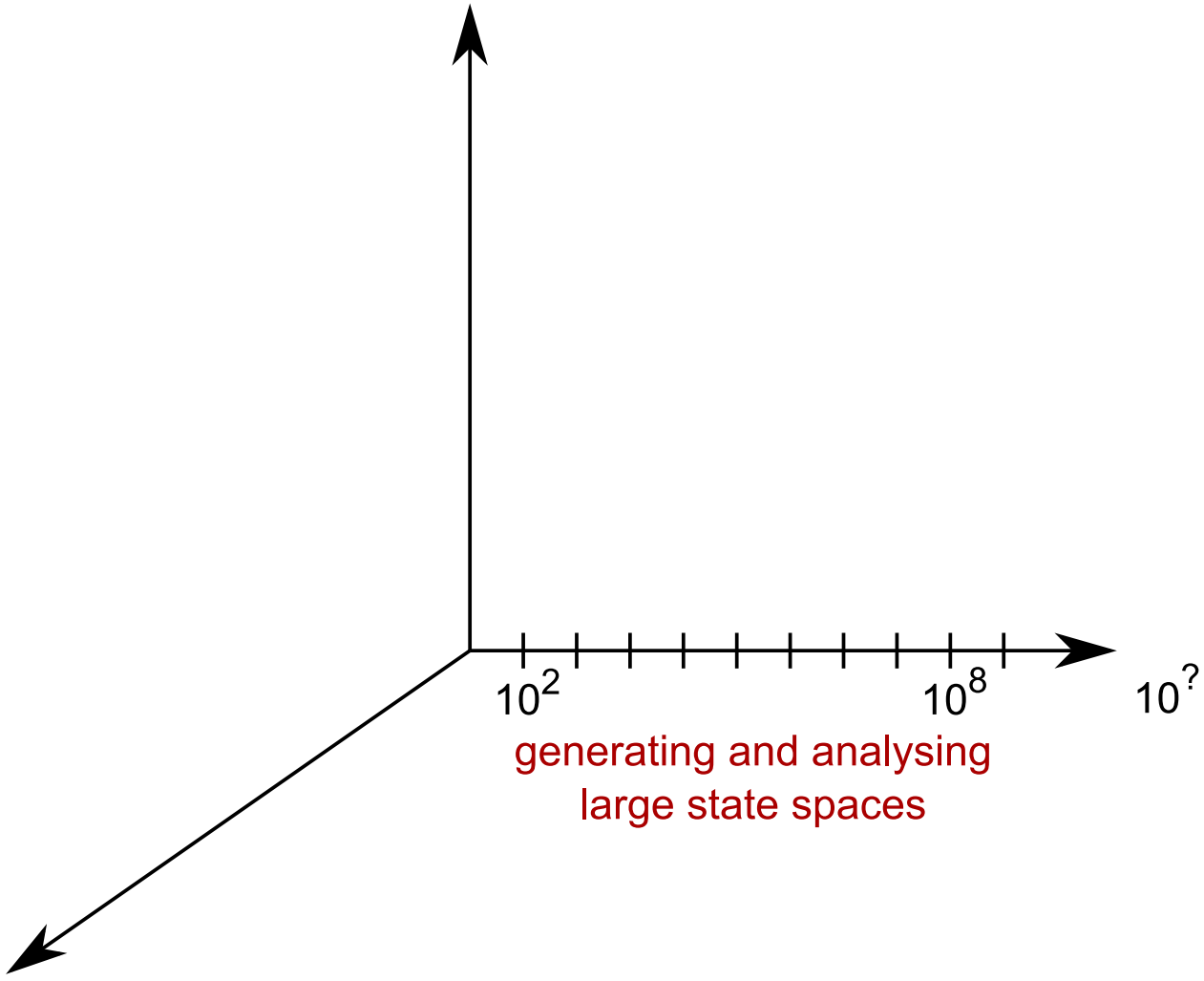


Drive capacity (Seagate projections)

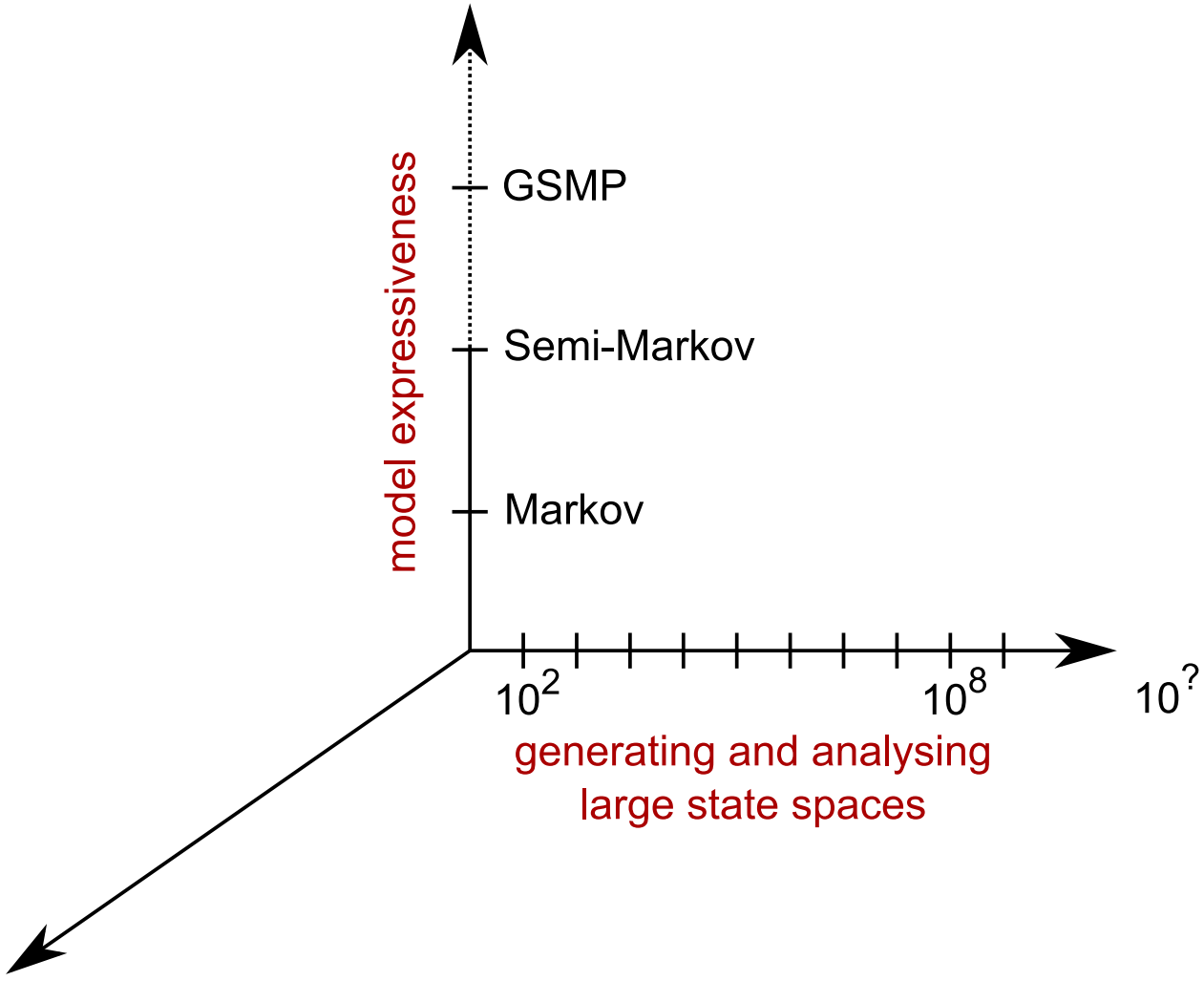
	2006	2009	2013
Capacity (GB)	750	2000	8000
Areal Density (GB/sq in)	133	500	1800
Transfer Rate (Mb/s)	930	2000	5000
RPM	7200	7200	10000
read seek time (ms)	8	7.2	6.5

Source: Dave Anderson, Seagate Techologies, 2007

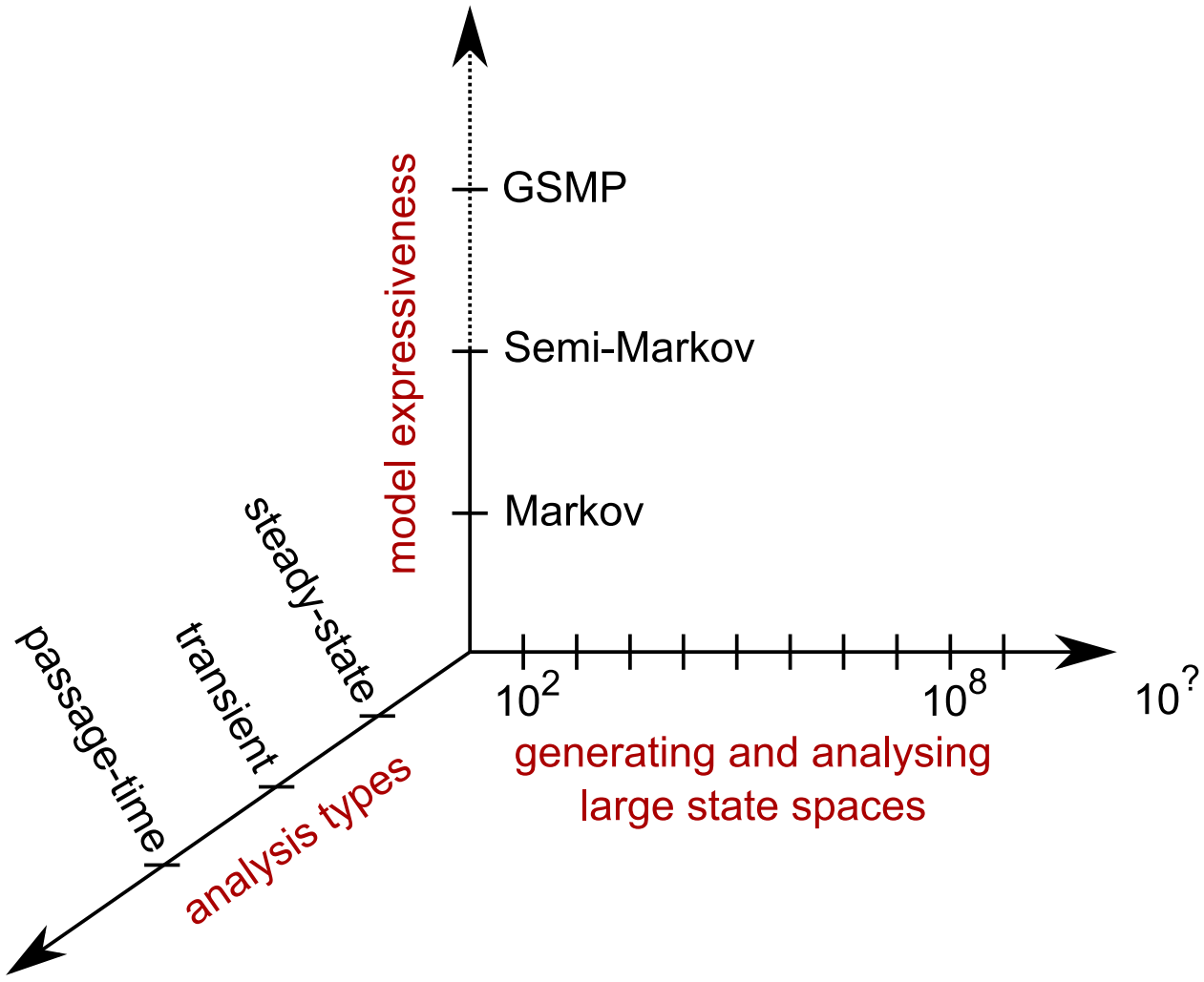
Talk Overview



Talk Overview



Talk Overview



CTMC Introduction

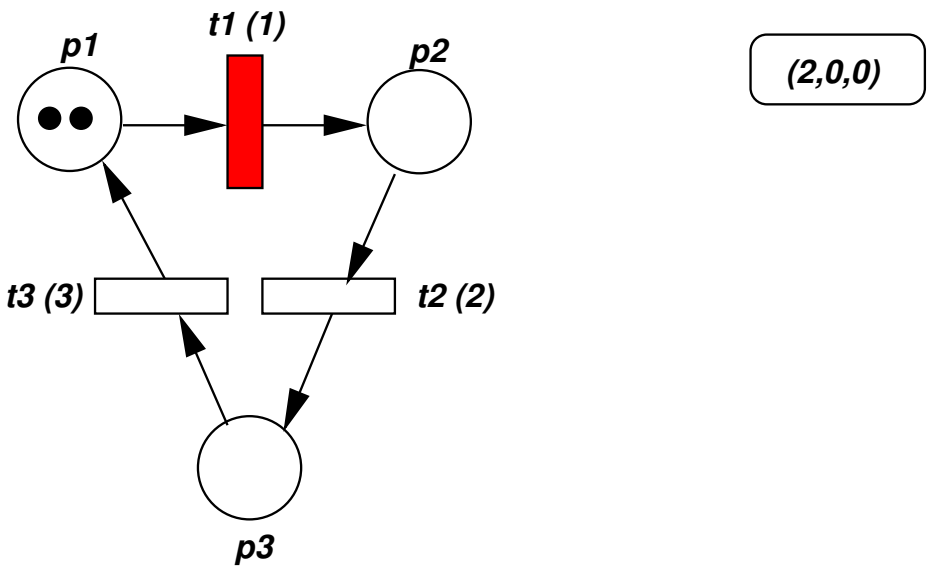
- Model systems by identifying all possible states system can enter (the **state space**) and ways in which system can move between states (the **state graph**)
- State transitions describe a stochastic process $\chi(t)$
- Focus on time-homogeneous Markov processes with discrete, finite state spaces that evolve in continuous time

CTMC Definition

- Evolution of an homogenous N-state CTMC described by $N \times N$ generator matrix $\mathbf{Q}$
- q_{ij} is the infinitesimal rate of moving from state i to state j ($i \neq j$)
- $q_{ii} = -\sum_{j \neq i} q_{ij}$
- Sojourn time in state i is exponentially distributed with rate parameter $\mu_i = -q_{ii}$
- Steady state distribution $\boldsymbol{\pi}\mathbf{Q} = \mathbf{0}$ with $\sum \pi_i = 1$

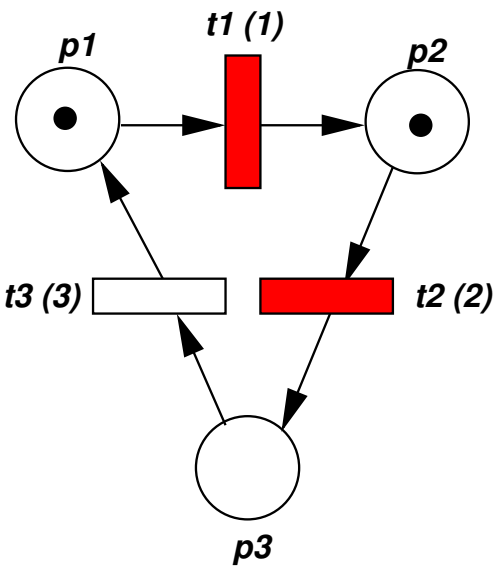
Generating CTMCs

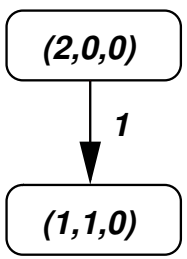
- CTMCs can be automatically derived from several higher-level formalisms e.g. Stochastic Petri nets, Stochastic Process Algebras
- Example:



Generating CTMCs

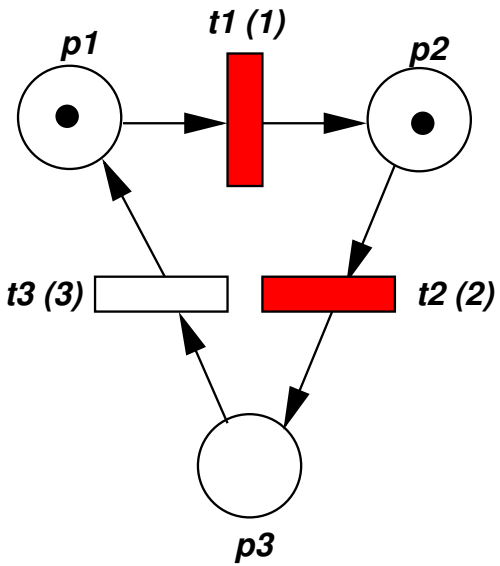
- CTMCs can be automatically derived from several higher-level formalisms e.g. Stochastic Petri nets, Stochastic Process Algebras
- Example:

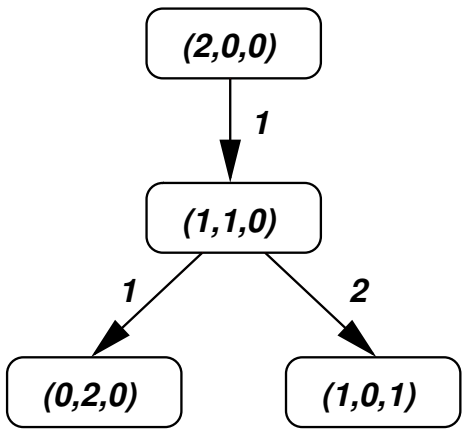




Generating CTMCs

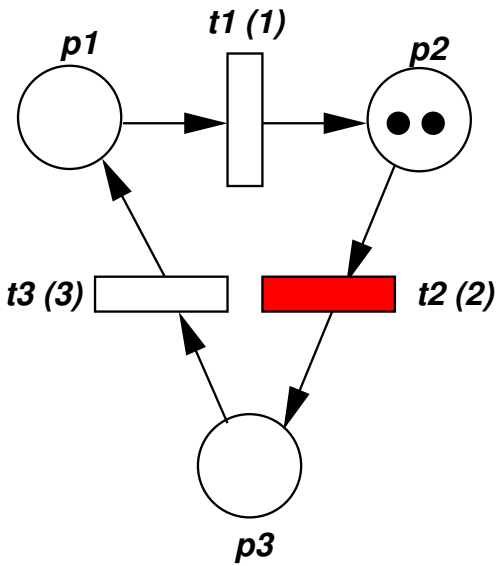
- CTMCs can be automatically derived from several higher-level formalisms e.g. Stochastic Petri nets, Stochastic Process Algebras
- Example:

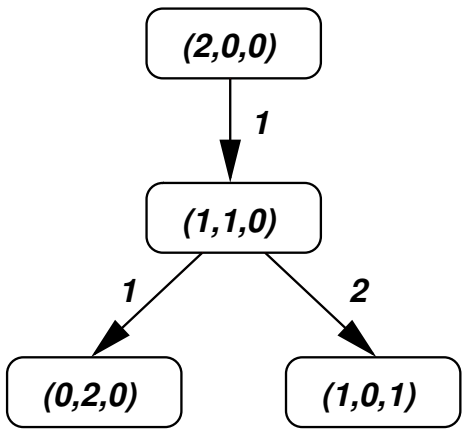




Generating CTMCs

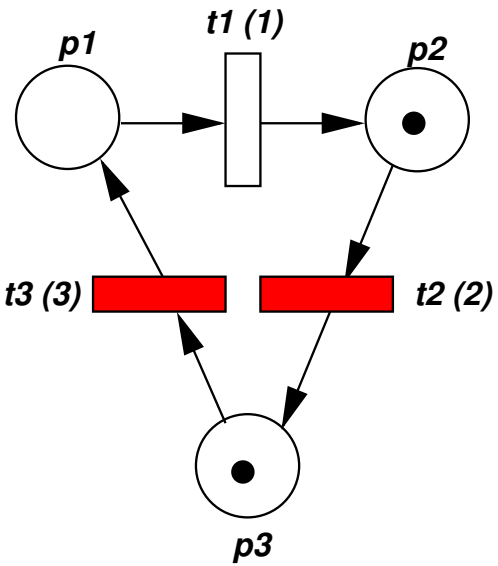
- CTMCs can be automatically derived from several higher-level formalisms e.g. Stochastic Petri nets, Stochastic Process Algebras
- Example:

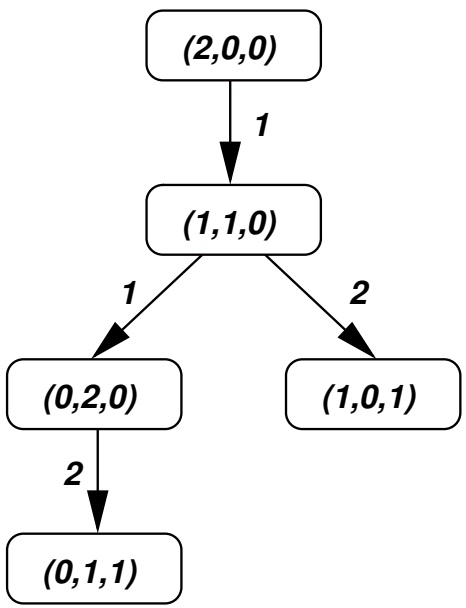




Generating CTMCs

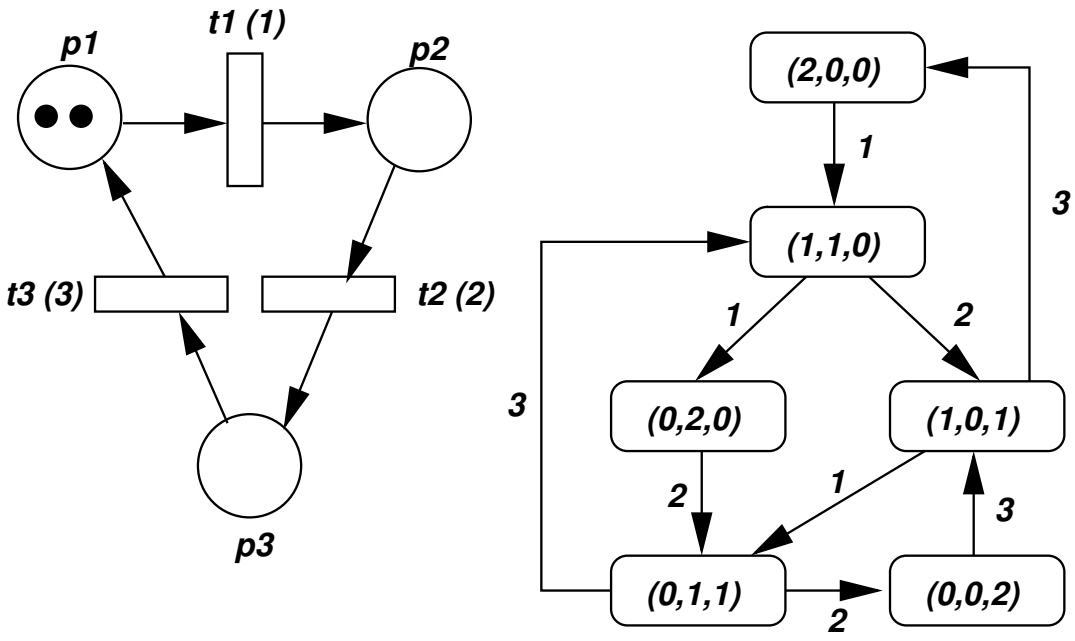
- CTMCs can be automatically derived from several higher-level formalisms e.g. Stochastic Petri nets, Stochastic Process Algebras
- Example:





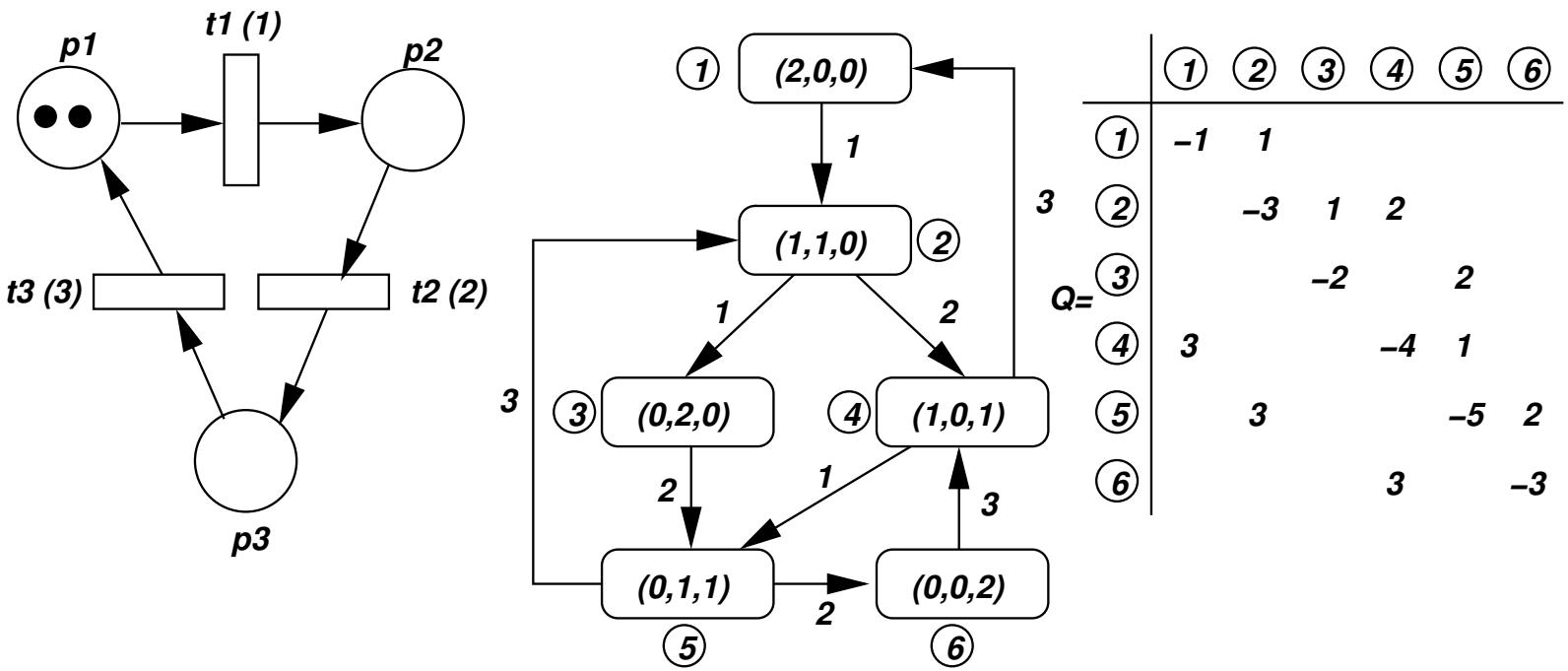
Generating CTMCs

- CTMCs can be automatically derived from several higher-level formalisms e.g. Stochastic Petri nets, Stochastic Process Algebras
- Example:



Generating CTMCs

- CTMCs can be automatically derived from several higher-level formalisms e.g. Stochastic Petri nets, Stochastic Process Algebras
- Example:



Response Time Analysis with Uniformisation

- CTMCs can be "uniformised" for purposes of first passage time analysis:

$$\mathbf{P} = \mathbf{Q}/q + \mathbf{I}$$

  where the rate $q > \max_i |q_{ii}|$ ensures that $\mathbf{P}$ is aperiodic
- $\mathbf{P}$ is the one-step transition matrix of a DTMC; has equivalent behaviour to CTMC when number of transitions is given by an associated Poisson process of rate q.
- Modify $\mathbf{P}$ to produce $\mathbf{P}'$ by making all target states absorbing

Response Time Analysis with Uniformisation

Density of time between states can be found by convolving state holding-time densities along all paths between states

$$
\begin{aligned}
f_{\vec{i}\vec{j}}(t) &= \sum_{n=1}^{\infty} \left(\frac{q^n t^{n-1} e^{-qt}}{(n-1)!} \sum_{k \in \vec{j}} \pi_k^{(n)} \right) \\
&\simeq \sum_{n=1}^{m} \left(\frac{q^n t^{n-1} e^{-qt}}{(n-1)!} \sum_{k \in \vec{j}} \pi_k^{(n)} \right)
\end{aligned}
$$

where:

$$
\boldsymbol{\pi}^{(n+1)} = \boldsymbol{\pi}^{(n)} \mathbf{P}' \qquad \text{for } n \geq 0
$$

with:

$$
\pi_k^{(0)} = \begin{cases} 0 & \text{for } k \notin \vec{i} \\ \pi_k / \sum_{j \in \vec{i}} \pi_j & \text{for } k \in \vec{i} \end{cases}
$$

SMP Introduction

- Extensions of Markov processes that allow for generally distributed sojourn times
- Markov property holds at transition instants
- SMP characterised by two matrices **P** and **H** with elements p_{ij} and $H_{ij}(t)$ respectively
- $p_{ij} = \mathbb{P}(\chi_{n+1} = j \mid \chi_n = i)$ is the state transition probability between states i and j (embedded DTMC); χ_n is the state at transition n
- $H_{ij}(t) = \mathbb{P}(T_{n+1} - T_n \leq t \mid \chi_{n+1} = j, \chi_n = i)$ is the sojourn time distribution in state i when the next state is j; T_n is the time of transition n

Generating SMPs

SMPs can be generated from various higher-level formalisms (e.g. SM-SPN, SM-PEPA)

Definition

An SM-SPN consists of a 4-tuple, $(PN, \mathcal{P}, \mathcal{W}, \mathcal{D})$, where:

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

Steady State Analysis of SMPs

- Solve for steady state distribution of embedded DTMC:

$$\boldsymbol{\pi} = \boldsymbol{\pi}\mathbf{P}$$

- Calculate average sojourn time in state i:

$$\mathbb{E}[\tau_i] = \sum_j p_{ij}\mathbb{E}[\tau_{ij}]$$

  where $\mathbb{E}[\tau_{ij}]$ is the expected sojourn time in state i when moving to state j.
- Then the steady-state probability of being in state i is:

$$\phi_i = \frac{\pi_i \mathbb{E}[\tau_i]}{\sum_{m=1}^{N} \pi_m \mathbb{E}[\tau_m]}$$

Response Time Analysis with Laplace Transforms

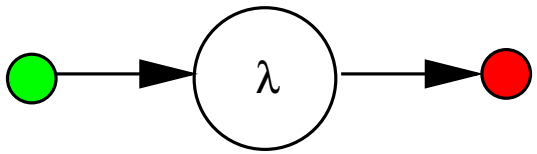
- We calculate the Laplace Transform (LT) of required response time density and then invert it numerically.
- Laplace Transform of $f(t)$ is

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

- For $f(t) = \lambda e^{-\lambda t}$

$$\begin{aligned} L(s) &= \int_0^\infty \lambda e^{-(\lambda+s)t} dt \\ &= \lambda/(\lambda+s) \int_0^\infty (\lambda+s) e^{-(\lambda+s)t} dt \\ &= \lambda/(\lambda+s) \end{aligned}$$

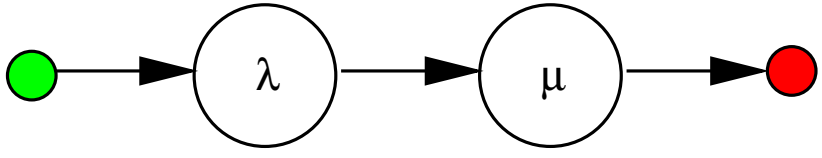
Why the Laplace Domain?



$$L(s) = \left(\frac{\lambda}{\lambda + s}\right) \Leftrightarrow f(t) = \lambda e^{-\lambda t}$$

Uniqueness

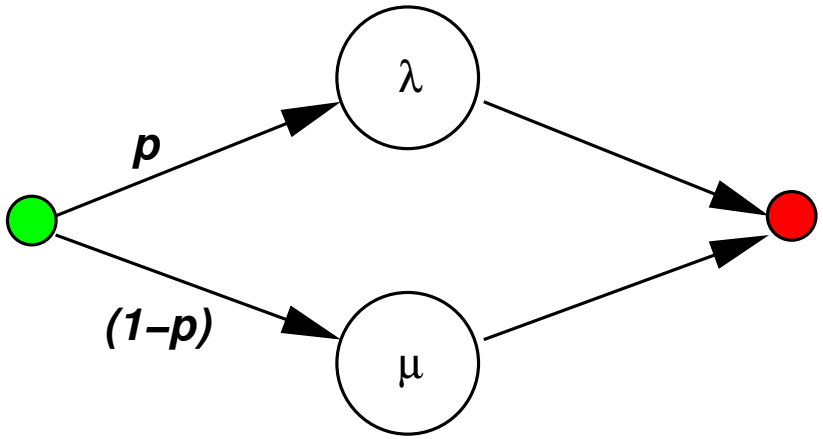
Why the Laplace Domain?



$$L(s) = \left(\frac{\lambda}{\lambda + s}\right)\left(\frac{\mu}{\mu + s}\right)$$

Convolution (easier than $h(t) = f(t) * g(t) = \int_0^t f(\alpha)g(t-\alpha)d\alpha$)

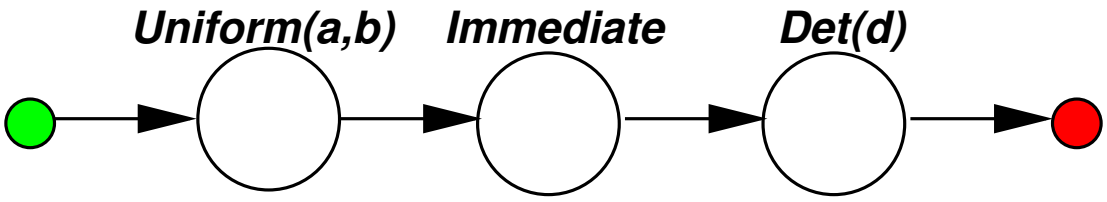
Why the Laplace Domain?



$$L(s) = p\left(\frac{\lambda}{\lambda + s}\right) + (1 - p)\left(\frac{\mu}{\mu + s}\right)$$

Linearity

Why the Laplace Domain?



$$L(s) = \left(\frac{e^{-as} - e^{-bs}}{(b-a)s} \right) (1) \left(e^{-ds} \right)$$

Extensible to non-exponential time delays (SMPs)

Why the Laplace Domain?

- Integration in t domain corresponds to division in Laplace domain.
- By inverting $L(s)/s$ instead of $L(s)$ we obtain cumulative density function (cdf) cheaply.

$$\begin{aligned} L_{f(t)}(s) &= \int_0^\infty e^{-st} f(t)dt = e^{-st}F(t)\Big|_0^\infty + s\int_0^\infty e^{-st}F(t)dt \\ &= sL_{F(t)}(s) \end{aligned}$$

- Quantiles (percentiles) can be easily calculated from cdf.

Why the Laplace Domain?

- Calculating (raw) moments is straightforward:

$$\begin{aligned} L'(s) &= \int_0^\infty -te^{-st}f(t)dt \\ L'(0) &= -\int_0^\infty tf(t)dt \\ &= -E[T] \end{aligned}$$

- Higher moments can be found via $L''(0)$, $L'''(0)$ etc.
- i.e. From the Laplace Transform, we can easily find the mean, variance, skewness, kurtosis etc. of $f(t)$

Response Time Analysis with Laplace Transforms

- First step analysis (source state i, target states $\vec{j}$):

$$L_{i\vec{j}}(s) = \sum_{k \notin \vec{j}} p_{ik} h^*_{ik}(s) L_{k\vec{j}}(s) + \sum_{k \in \vec{j}} p_{ik} h^*_{ik}(s)$$

- For CTMCs:

$$L_{i\vec{j}}(s) = \sum_{k \notin \vec{j}} \frac{q_{ik}}{s - q_{ii}} L_{k\vec{j}}(s) + \sum_{k \in \vec{j}} \frac{q_{ik}}{s - q_{ii}}$$

- With multiple source states:

$$L_{\vec{i}\vec{j}}(s) = \sum_{k \in \vec{i}} \alpha_k L_{k\vec{j}}(s) \quad \text{where} \quad \alpha_k = \frac{\tilde{\pi}_k}{\sum_{j \in \vec{i}} \tilde{\pi}_j}$$

Response Time Analysis with Laplace Transforms

- As a system of n linear equations in $\mathbf{Ax} = \mathbf{b}$ form ($\vec{j} = 1$):

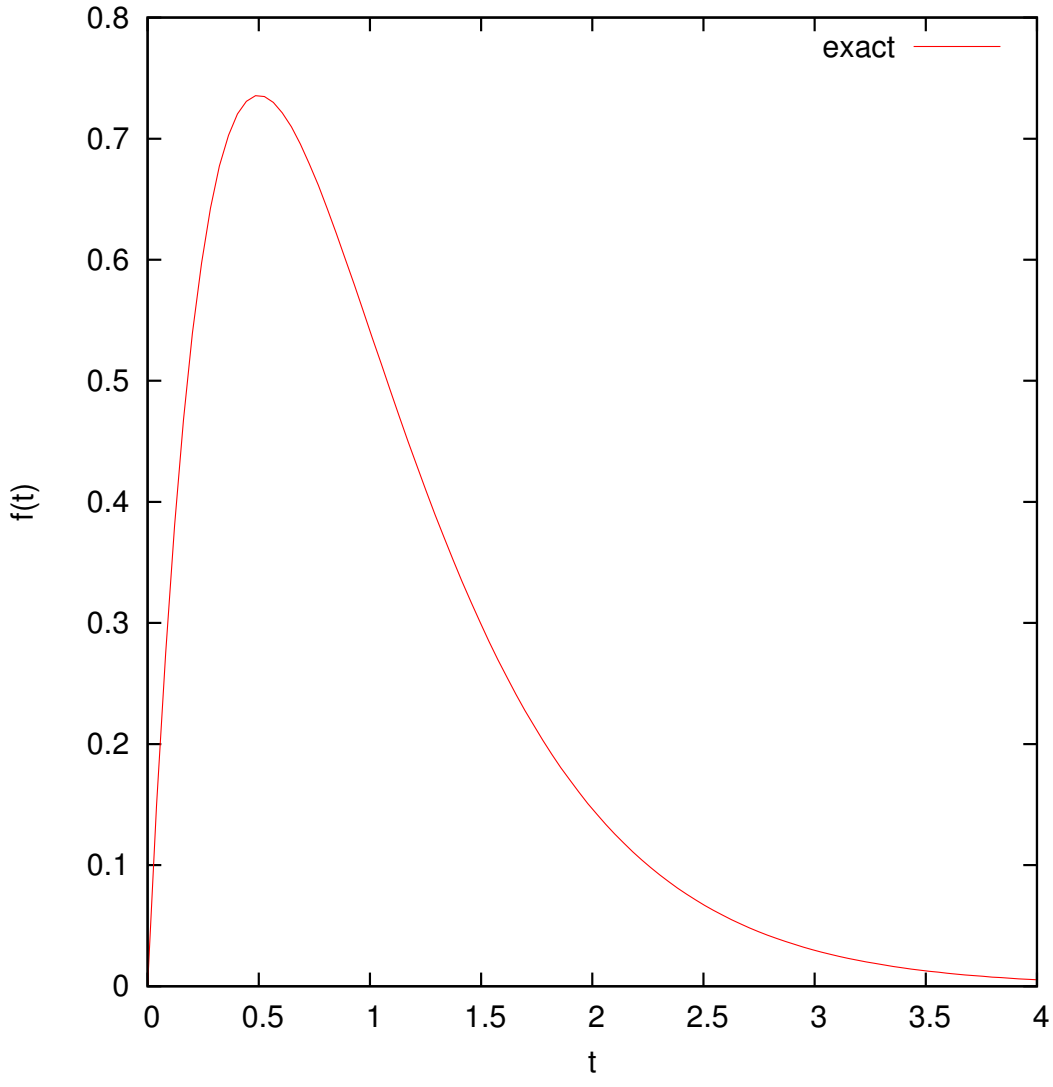
$$\begin{pmatrix} 1 & -r^*_{12}(s) & \cdots & -r^*_{1n}(s) \\ 0 & 1 - r^*_{22}(s) & \cdots & -r^*_{2n}(s) \\ 0 & -r^*_{32}(s) & \cdots & -r^*_{3n}(s) \\ 0 & \vdots & \ddots & \vdots \\ 0 & -r^*_{n2}(s) & \cdots & 1 - r^*_{nn}(s) \end{pmatrix} \begin{pmatrix} L_{1\vec{j}}(s) \\ L_{2\vec{j}}(s) \\ L_{3\vec{j}}(s) \\ \vdots \\ L_{n\vec{j}}(s) \end{pmatrix} = \begin{pmatrix} r^*_{11}(s) \\ r^*_{21}(s) \\ r^*_{31}(s) \\ \vdots \\ r^*_{n1}(s) \end{pmatrix}$$

  where $r^*_{ij}(s) = p_{ij} h^*_{ij}(s)$
- Symbolic solution infeasible for large n.
- Instead solve for particular values of s based on evaluation demands of numerical LT inversion algorithm.

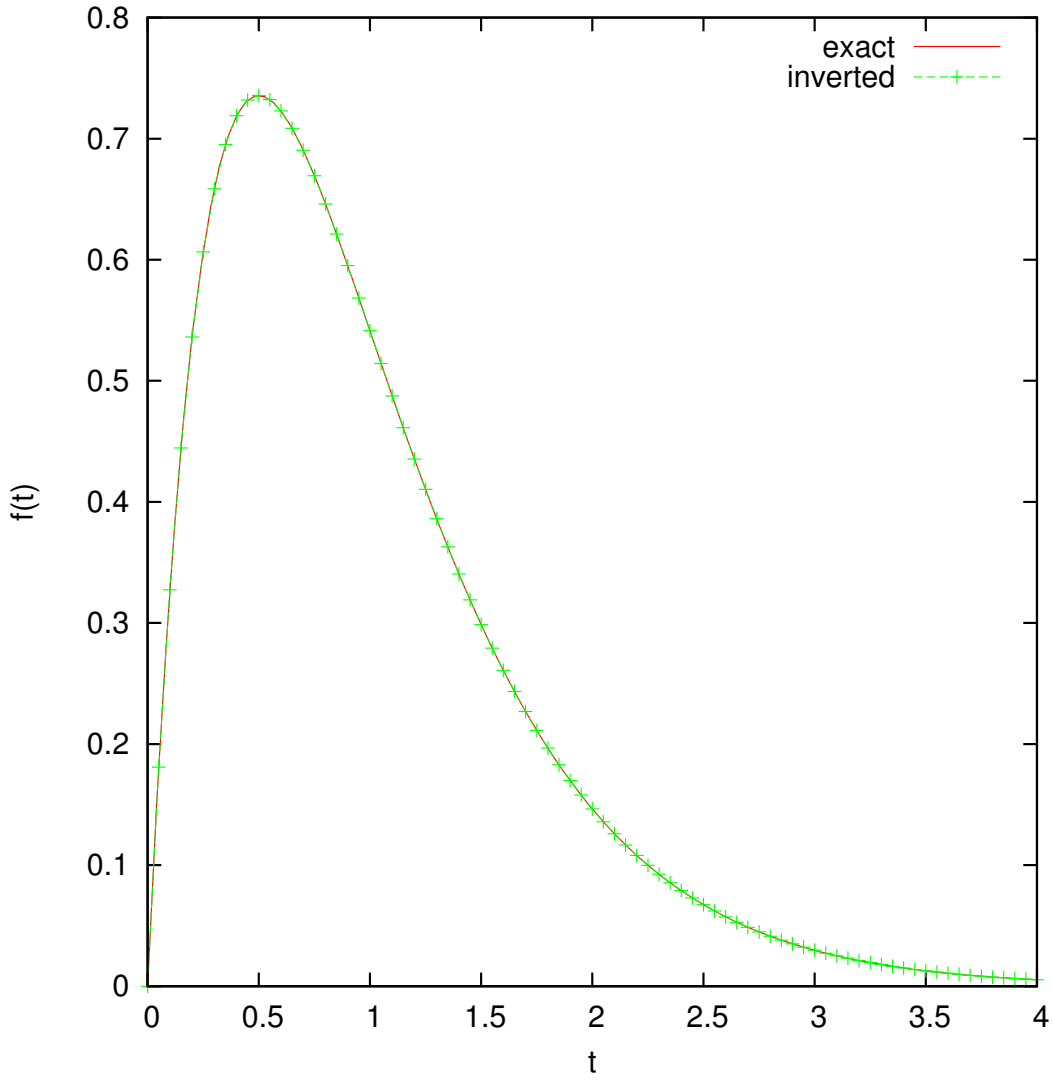
Response Time Analysis with Laplace Transforms

- General principle:
  - Input: values of t
  - Inverter demands: value of $L(s)$ set for several s
  - Output: values of $f(t)$
- Computational cost (values of $L(s)$ required):
  - Euler: $\approx 30|t|$, Laguerre (modified): 400
- Euler inversion on a 15 000 000 state model and 50 t points requires solving 1 500 systems of $15\,000\,000 \times 15\,000\,000$ (complex) sparse linear equations!

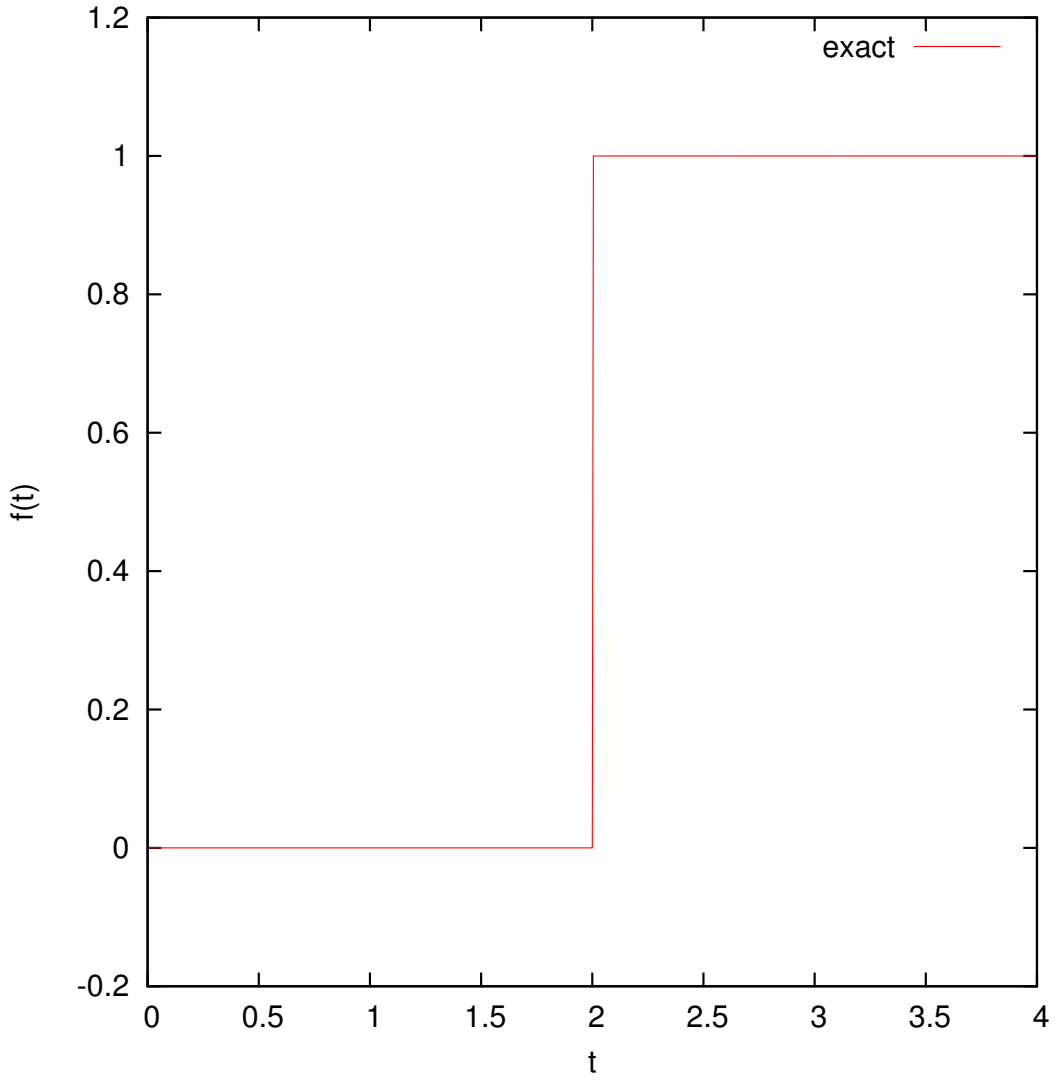
Numerical Laplace Transform Inversion (Erlang-2, $\lambda = 2$)



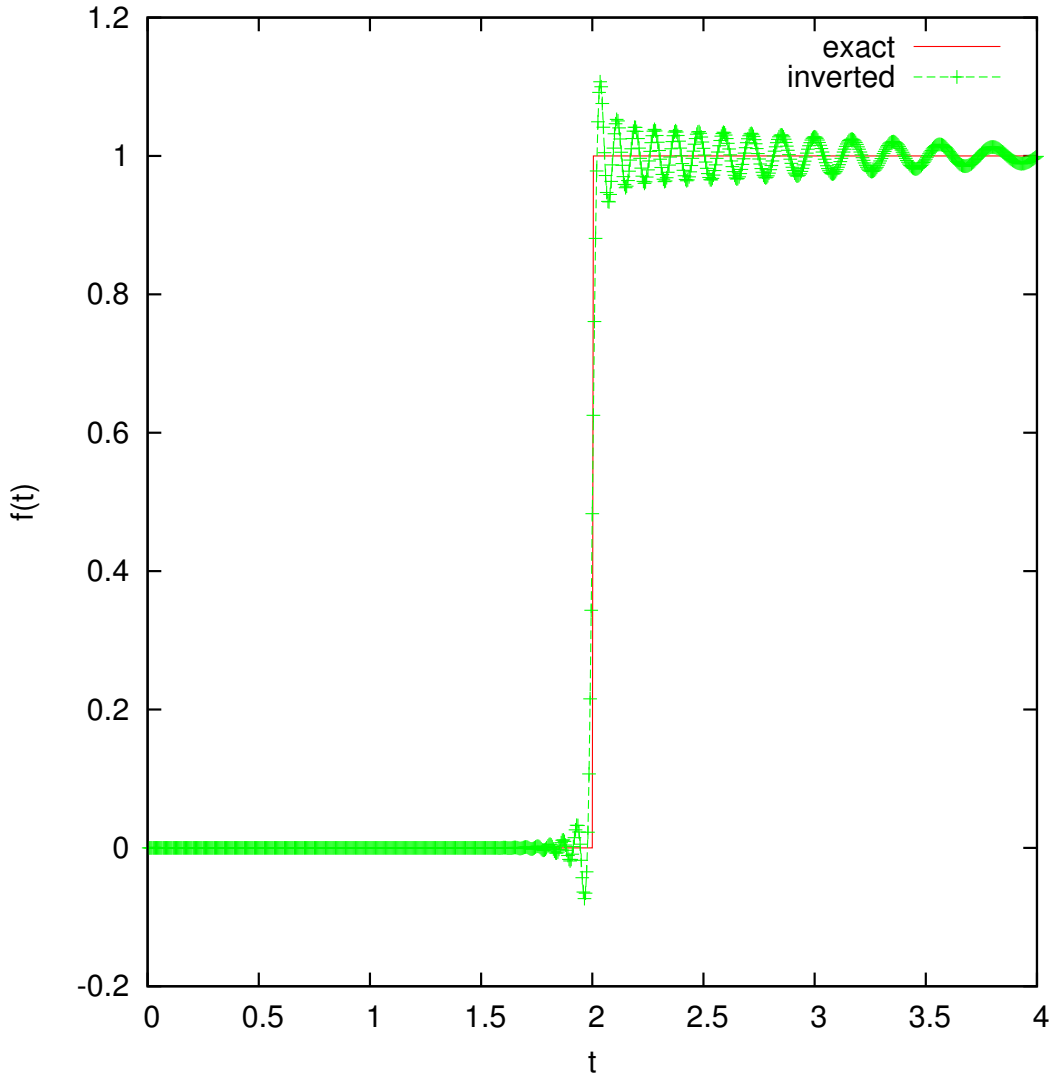
Numerical Laplace Transform Inversion (Erlang-2, $\lambda = 2$)



Numerical Laplace Transform Inversion (Heavyside-2, $\lambda = 2$)



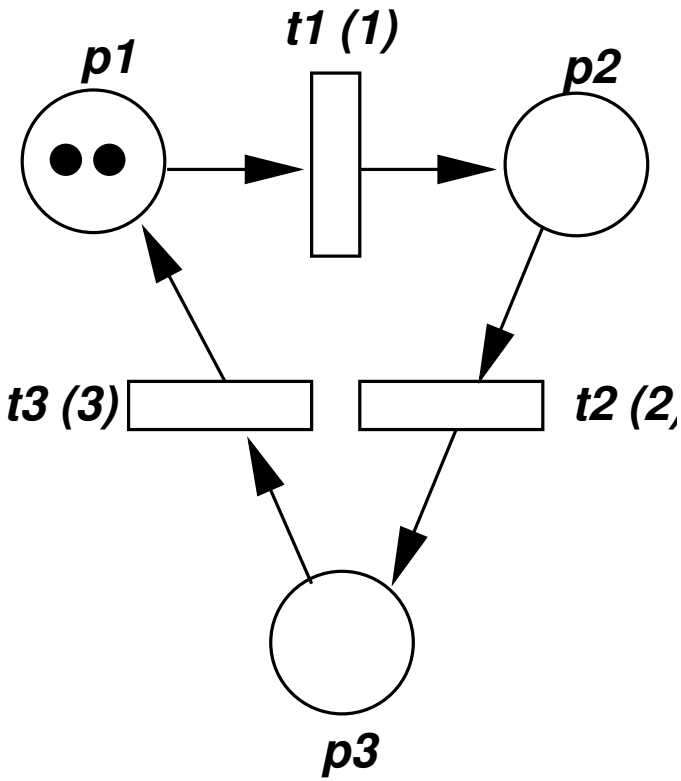
Numerical Laplace Transform Inversion (Heavyside-2, $\lambda = 2$)



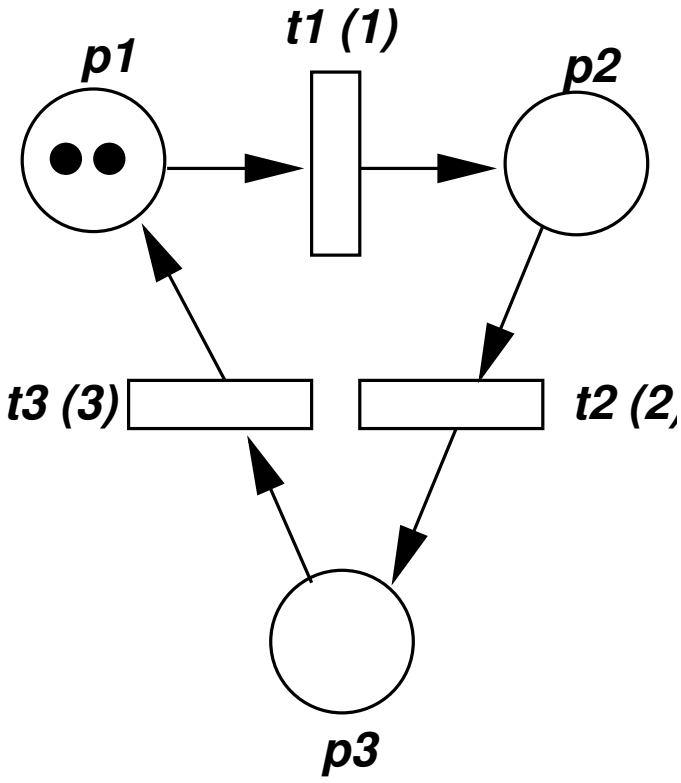
The Challenge

- Generate representation of state space and state graph in face of:
  - huge number of states
  - large state descriptors
- Extract performance measures efficiently (in terms of both time and space)
- In both cases approaches should ideally be scalable

The Challenge

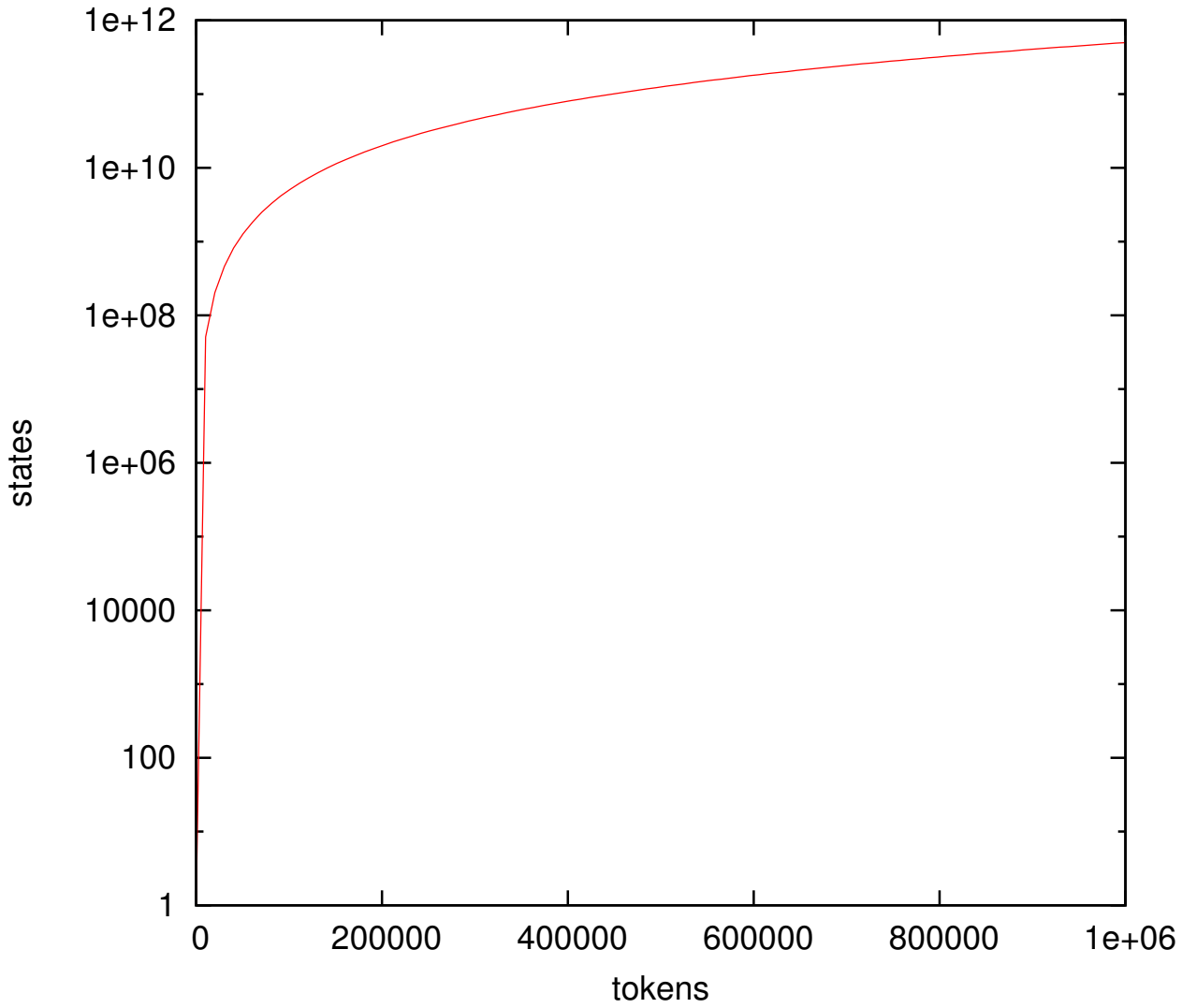


The Challenge

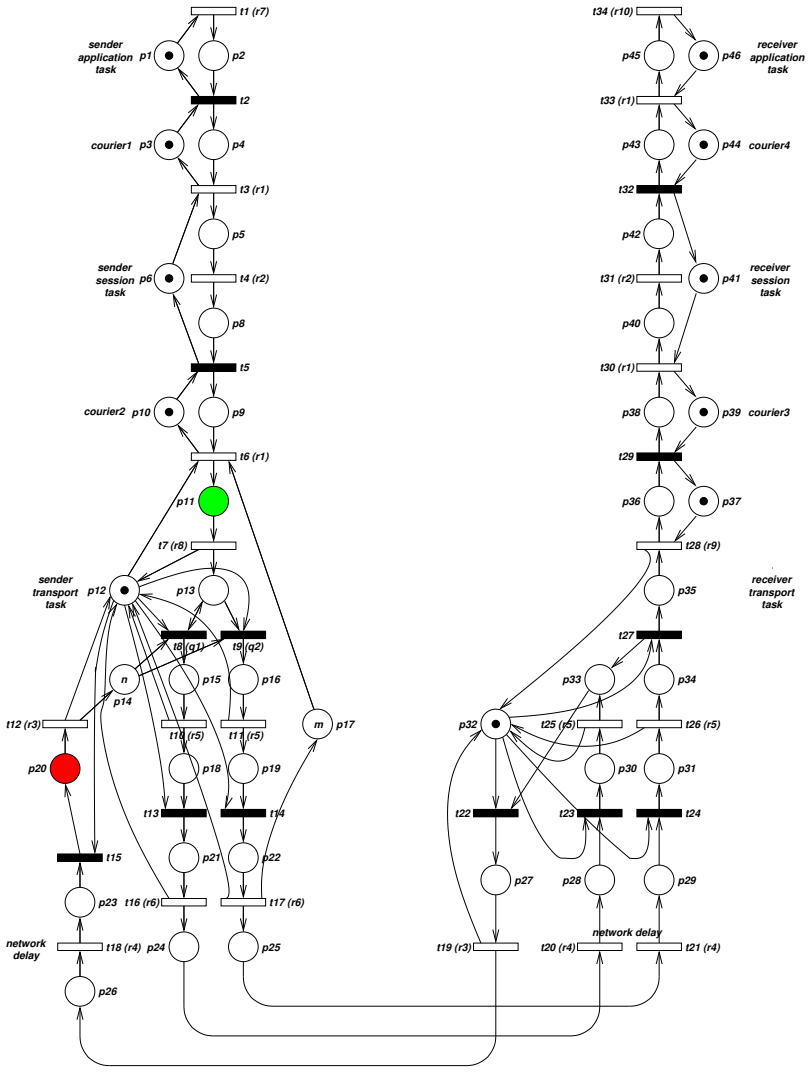


Looks innocent doesn't it!

The Challenge



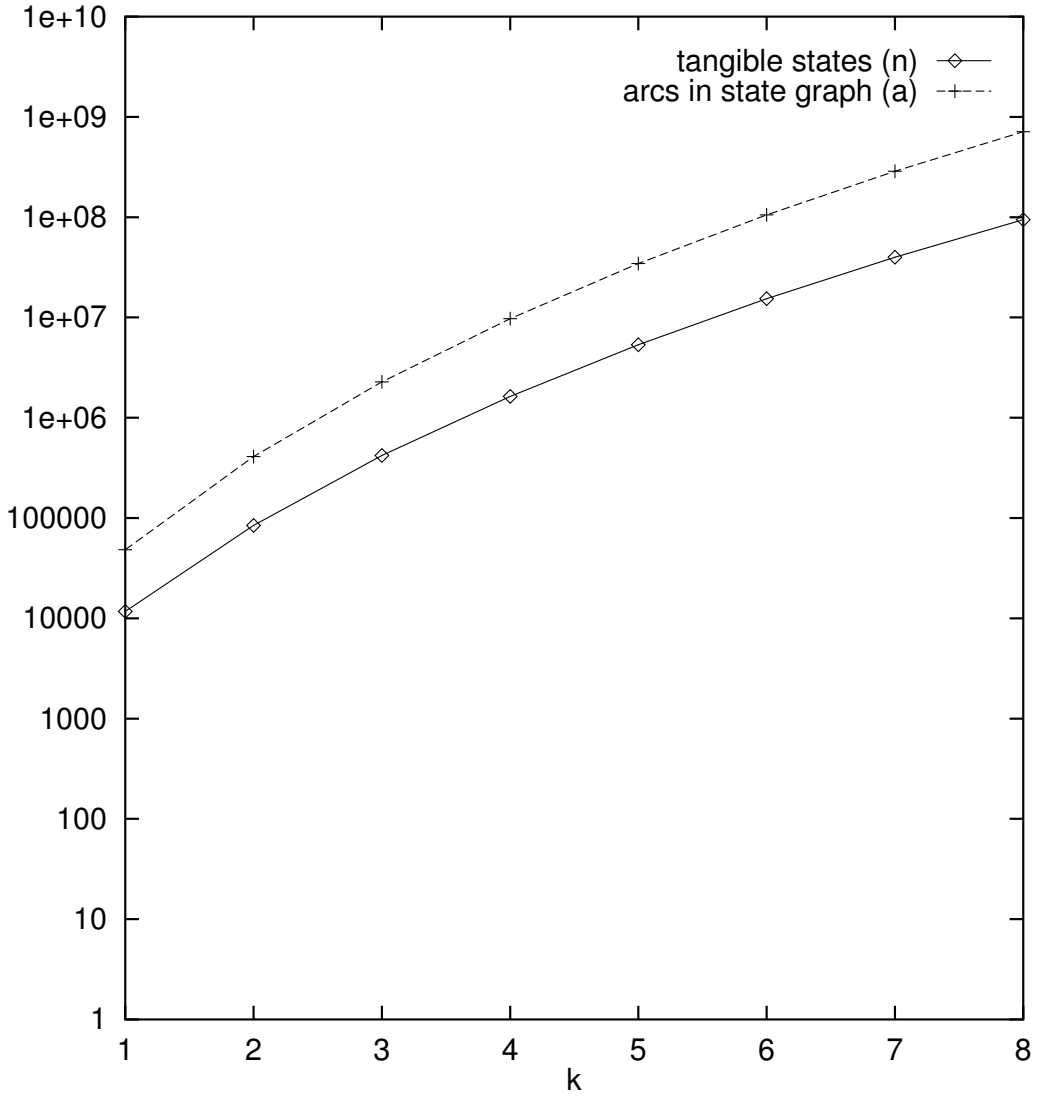
Courier Model



Courier State Space

k	n	a
1	11 700	48 330
2	84 600	410 160
3	419 400	2 281 620
4	1 632 600	9 732 330
5	5 358 600	34 424 280
6	15 410 250	105 345 900
7	39 836 700	286 938 630
8	94 322 250	710 223 930
9	207 498 600	1 623 000 330

Courier State Space



Algorithm

begin
  $E = \{s_0\}$
  F.add(s_0)
  $A = \emptyset$
  while (F not empty) **do begin**
    F.remove(s)
    for each $s' \in$ succ(s) **do begin**
      if $s' \notin E$ **do begin**
        F.add(s')
        $E = E \cup \{s'\}$
      end
      $A = A \cup \{\text{id}(s) \rightarrow \text{id}(s')\}$
    end
  end
end

Binary Decision Diagrams

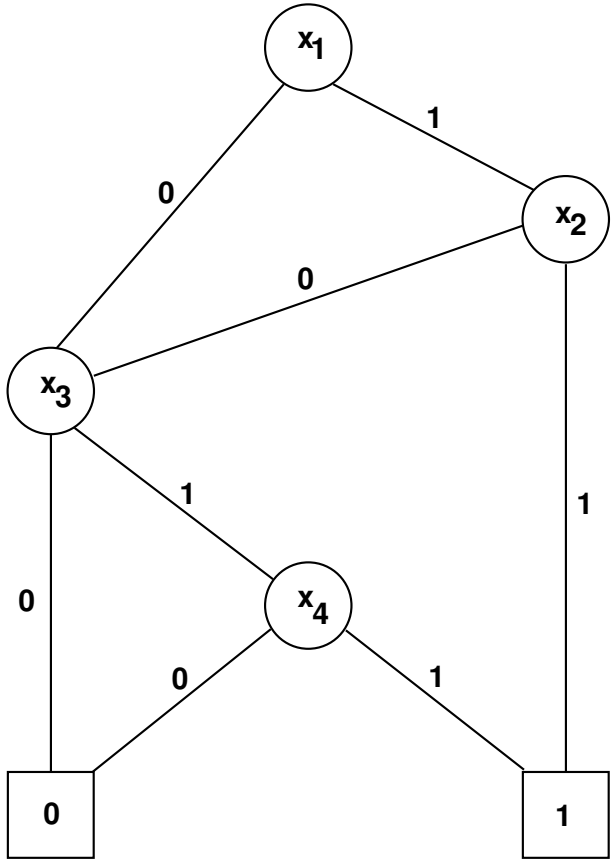


Figure: A binary decision diagram corresponding to the boolean function $(x_1 \wedge x_2) \vee (x_3 \wedge x_4)$, or, equivalently, the state space $\{0011, 0111, 1011, 1100, 1101, 1110, 1111\}$

Symbolic State Space Generation

begin
$E = F = \{s_0\}$
repeat
$T = \text{succ}(F)$
$N = T - E$
$F = N$
$E = E \cup N$
until $(N = \emptyset)$
end

Figure: Algorithm for symbolic state space generation. E, T, F, N are BDDs representing “explored”, “to”, “from”, “new” states respectively.

Multi-Terminal Binary Decision Diagrams

$$R = \begin{pmatrix} 0 & 4 & 0 & 0 \\ 3 & 0 & 3 & 0 \\ 0 & 2 & 0 & 2 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

transition	r_1	c_1	r_2	c_2	value
$0 \xrightarrow{4} 1$	0	0	0	1	4
$1 \xrightarrow{3} 0$	0	0	1	0	3
$1 \xrightarrow{3} 2$	0	1	1	0	3
$2 \xrightarrow{2} 1$	1	0	0	1	2
$2 \xrightarrow{2} 3$	1	1	0	1	2
$3 \xrightarrow{1} 2$	1	1	1	0	1
	otherwise				0

Multi-Terminal Binary Decision Diagrams

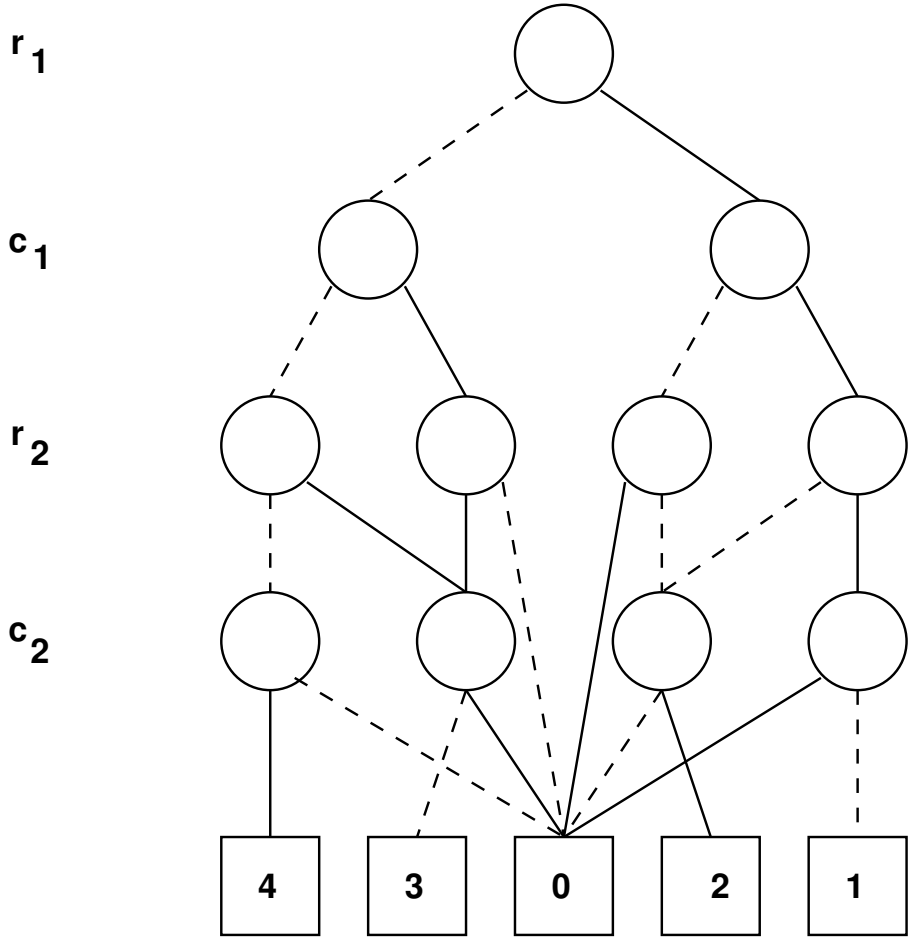


Figure: A multi-terminal binary decision diagram for encoding the matrix

Pros and Cons of (MT)BDDs

- Potential for exceeding bounds of explicit state enumeration methods by several orders of magnitude given sufficiently regular state space
- Sensitive to variable ordering
- Peak BDD size often an order of magnitude larger than final BDD size
- Completely symbolic numerical solution using MTBDDs has not (yet) met with success
- Other symbolic techniques (notably MDDs) will be described in this afternoon's lecture; we will focus here on explicit techniques

The Case for Probabilistic Algorithms

- Can be useful to relax the requirement that the solution produces the correct answer
- Such *probabilistic* or *randomised* approaches can lead to dramatic memory and time savings
- Risk of wrong result must be quantified and kept small
- Algorithms can be often made to be more reliable than the hardware they are running on!

The Case for Probabilistic Algorithms

Example

Miller-Rabin primality test of number n is based on three facts:

- If n is composite (i.e. not prime) then at least three quarters of the natural numbers less than n are *witnesses* to the compositeness of n
- If n is prime then there is no natural number less than n that is witness to the compositeness of n.
- Given number natural numbers m and n with $m < n$, there is an efficient algorithm which ascertains whether or not m is a witness to the compositeness of n.

We perform k witness tests using randomly chosen natural numbers less than n. If all fail assume n is prime. So chance of failing to find a witness for a composite number is 2^{-2k} and can be made arbitrarily small at logarithmic runtime cost

Application to State Space Generation

- Key idea: use one-way hashing to reduce memory requirements of the explored state table
- This reduces the amount of memory required to store states dramatically
- But introduces the risk that two distinct states will have the same hashed representation $\Rightarrow$ misidentification and omission of states in state graph
- Need to quantify this risk and keep it low.

Holzmann's Bit-state Hashing

- Attempts to maximise state coverage in face of limited memory
- Based on Bloom filters (devised in 1970!)
- Explored state table is a large bit vector T
- Initially all bits in T are set to zero
- Hash function $h(s)$ maps states onto bit vector positions, so when s is encountered, $T[h(s)]$ is set to 1
- To check if s is present, examine $T[h(s)]$. If zero, s has not been encountered previously; if one, assume (possibly erroneously) that s has already been encountered.

Holzmann's Bit-state Hashing

- Probability of no hash collisions when inserting n states into a bit vector of t bits is:

$$p = \frac{t!}{(t-n)!t^n} = \prod_{i=0}^{n-1} \frac{(t-i)}{t} = \prod_{i=0}^{n-1} \left(1 - \frac{i}{t}\right)$$

- Assuming $n \ll t$

$$p \approx \prod_{i=0}^{n-1} e^{-i/t} = e^{\sum_{i=0}^{n-1} -i/t} = e^{-\frac{n(n-1)}{2t}} = e^{\frac{n-n^2}{2t}} \approx e^{-\frac{n^2}{2t}}$$

- Corresponding probability of state omission is $q = 1 - p$
- Problem: To obtain $q = 0.1\%$ when inserting $n = 10^6$ states requires the allocation of a bit vector of 125TB!

Improving Holzmann's Method

- Why not use two hash functions $h_1(s)$ and $h_2(s)$?
- Then when we set $T[h_1(s)] = 1$ and $T[h_2(s)] = 1$ when we encounter a state
- And only conclude a state is explored if both $h_1(s) = 1$ and $h_2(s) = 1$
- Now

$$p \approx e^{-\frac{4n^3}{t^2}}.$$

- But this still requires an infeasibly large bit vector to achieve low state omission probabilities
- Actually optimal number of hash functions is around 20 (determined by Wolper and Leroy).

Interpreting the State Omission Probability

- The state omission probability does NOT indicate a fraction of states that are omitted or misidentified
- So if we generate a state space of 10^8 states with an omission probability of 0.1% this does NOT imply that 100000 states are incorrect
- Rather there is a 99.9% chance that the entire state space has been generated correctly

Wolper and Leroy's Hash Compaction

- Holzmann's method only gives good complete state coverage probability when most of bit vector is empty
- Wolper and Leroy proposed to store only which bit positions are occupied
- Done by hashing states onto compressed keys of b bits, which can then be stored in a smaller (fixed size) hash table which supports a collision resolution scheme
- Since approach is like Holzmann's method with a table of size 2^b, we have

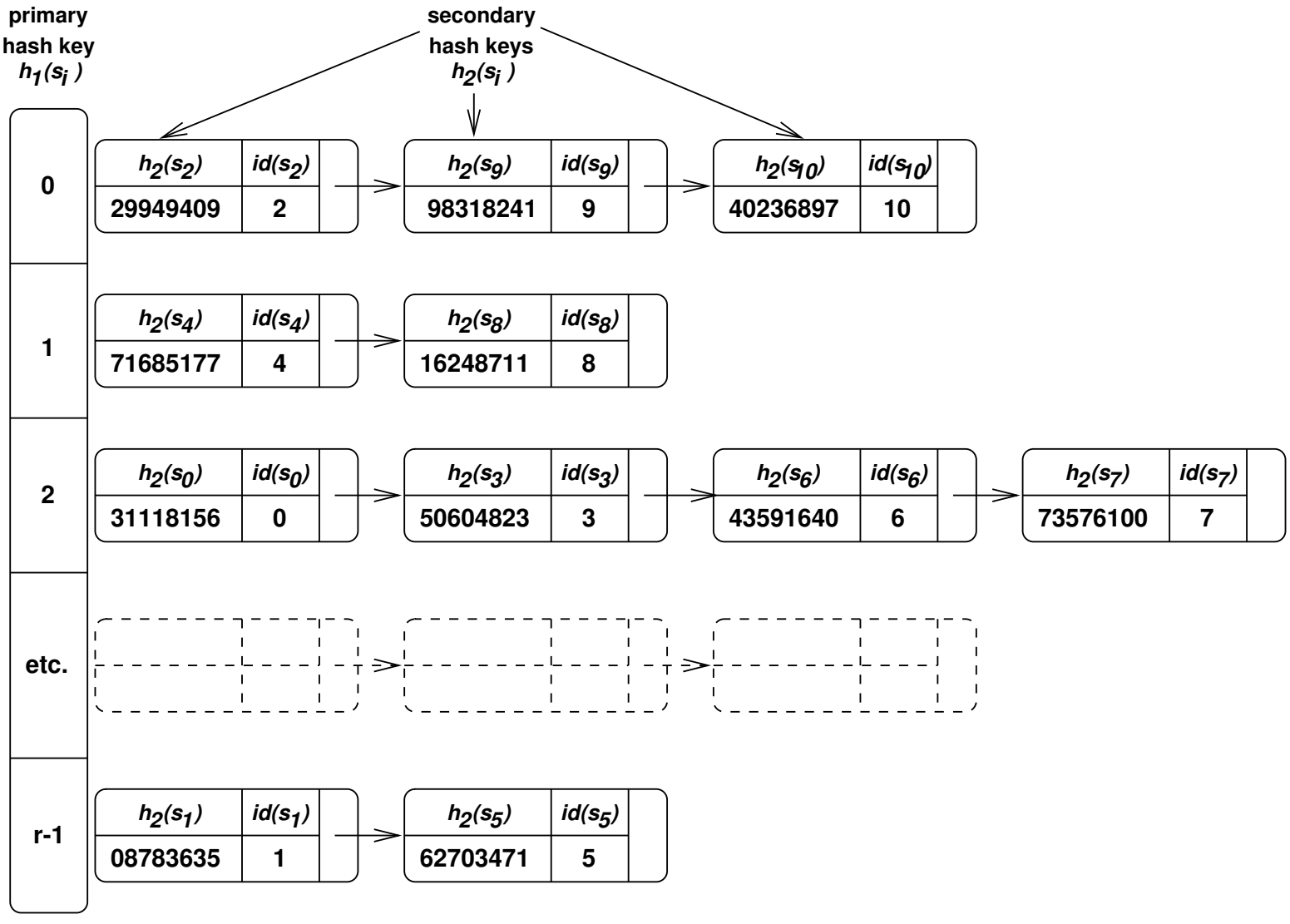
$$p \approx e^{-\frac{n^2}{2^{b+1}}}$$

Stern and Dill's Improved Hash Compaction

- Uses large static hash table
- Determines where to store states by using a separate independent hash function
- Also uses double hashing to keep number of probes per insertion low
- For a full table with n slots:

$$q \approx \frac{1}{2^b} n(\ln n - 1)$$

Dynamic Probabilistic State Space Generation



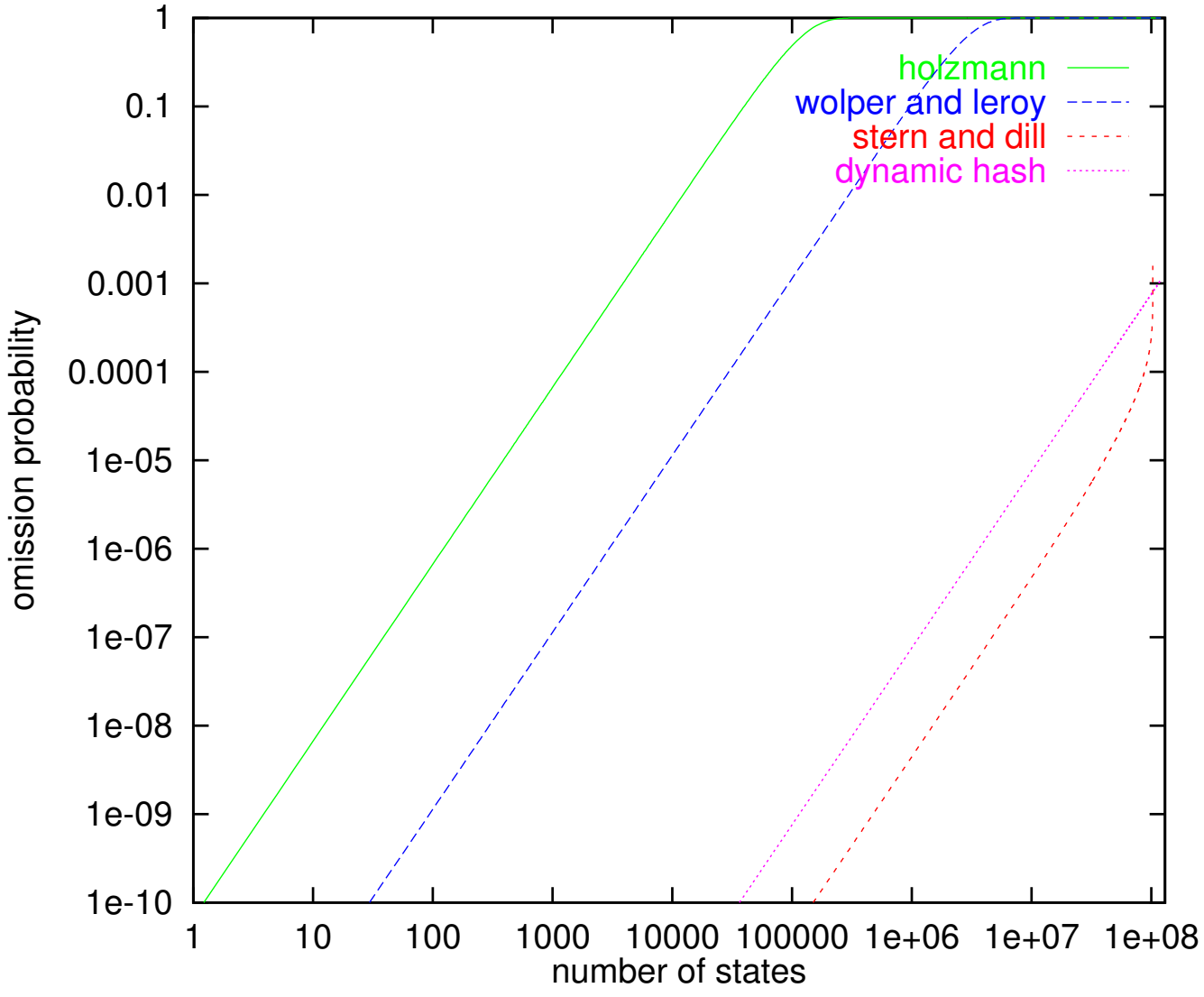
Dynamic Probabilistic State Space Generation

- Effectively we have r independent Wolper and Leroy-type hash tables with 2^b potential entries each:

$$p \approx e^{-\frac{n^2}{2^{b+1}r}} \approx 1 - \frac{n^2}{2^{b+1}r}$$

- This is in fact a lower bound for p (i.e. provides a conservative estimate for the probability of complete state coverage)

Comparison of State Omission Probabilities



Parallel Dynamic Probabilistic State Space Generation

- Use a partitioning hash function to assign states to processors

$$h_0(s) \rightarrow (0, \ldots, N-1)$$

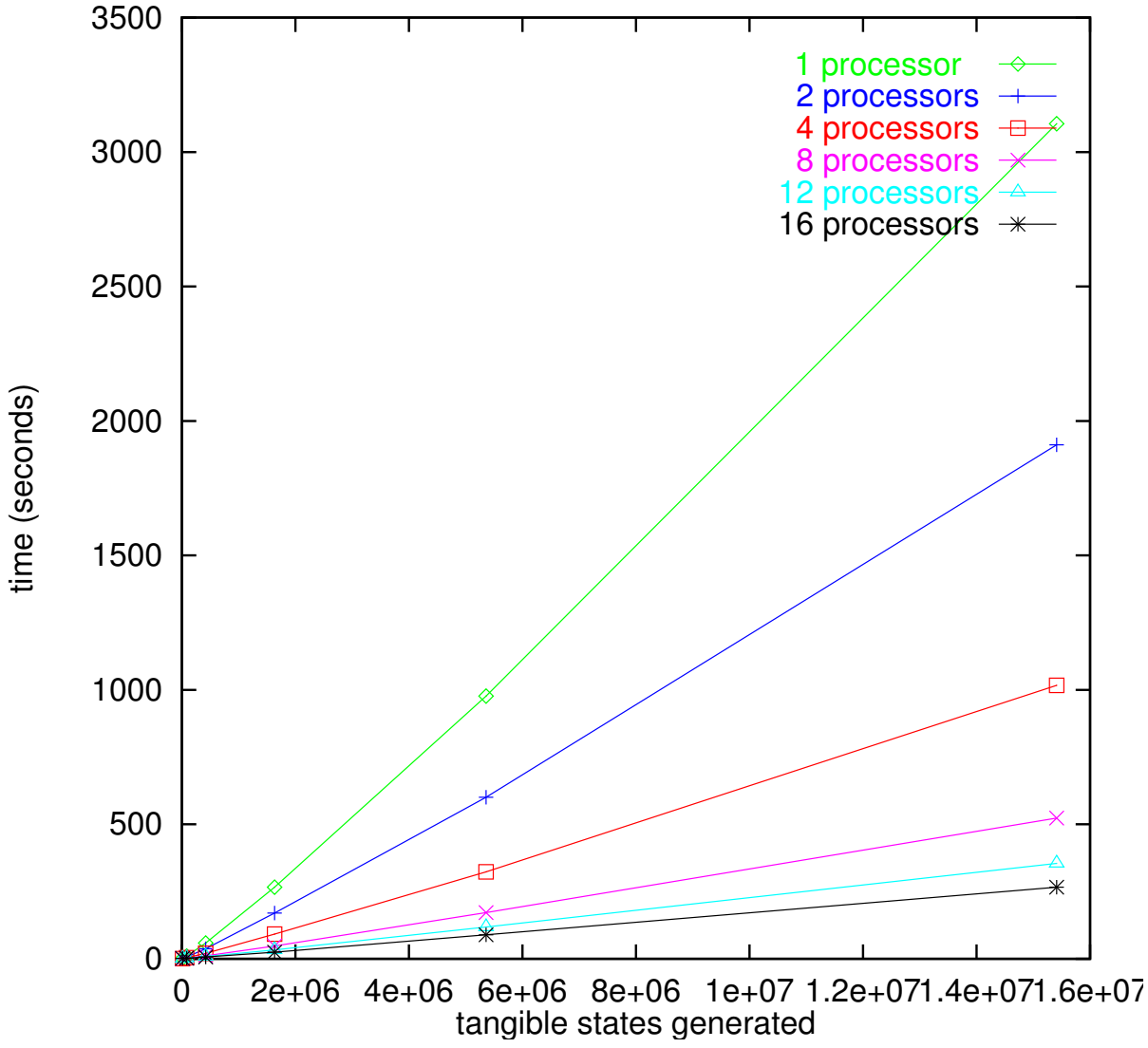
- Processor i constructs

$$\begin{aligned} E_i &= \{s : h_0(s) = i\} \\ A_i &= \{(s_1 \rightarrow s_2) : h_0(s_1) = i\} \end{aligned}$$

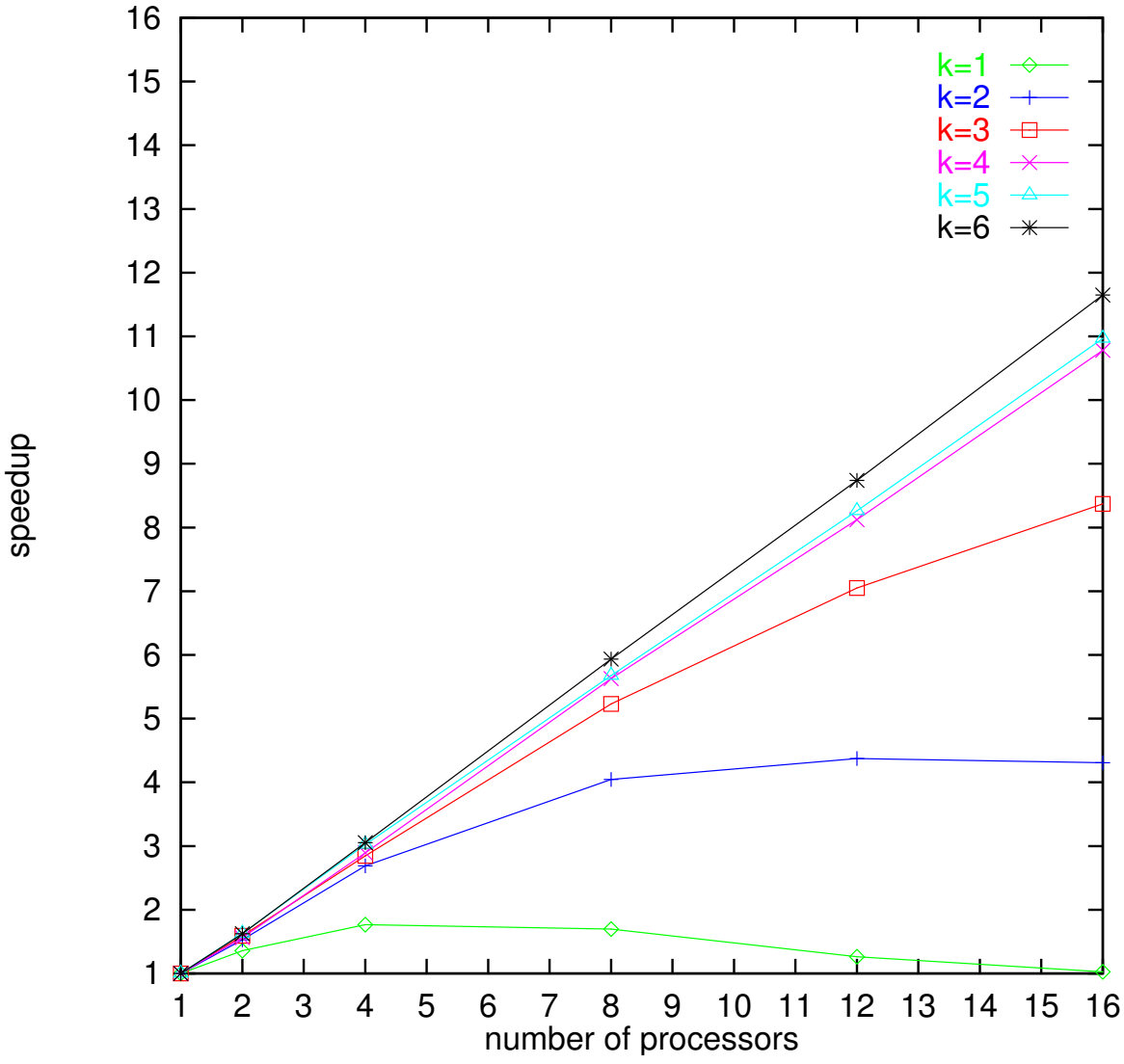
- Processor i has local FIFO queue F_i and hash table H_i
- Processors send states to owner processors (and receive identities of target states)
- Now we have:

$$p \approx e^{-\frac{n^2}{2^{b+1}Nr}} \approx 1 - \frac{n^2}{2^{b+1}Nr}$$

Courier Parallel State Space Generation Time



Courier Parallel State Space Generation Speedup



The Challenge

- Need to solve

$$\pi \mathbf{Q} = \mathbf{0}$$

  for very large, sparse $\mathbf{Q}$
- In doing so need to consider how to:
  - Represent $\mathbf{Q}$ (Kronecker, on-the-fly, MDD etc.)
  - Access elements of $\mathbf{Q}$
  - Store π
  - Scale solution

Numerical Methods

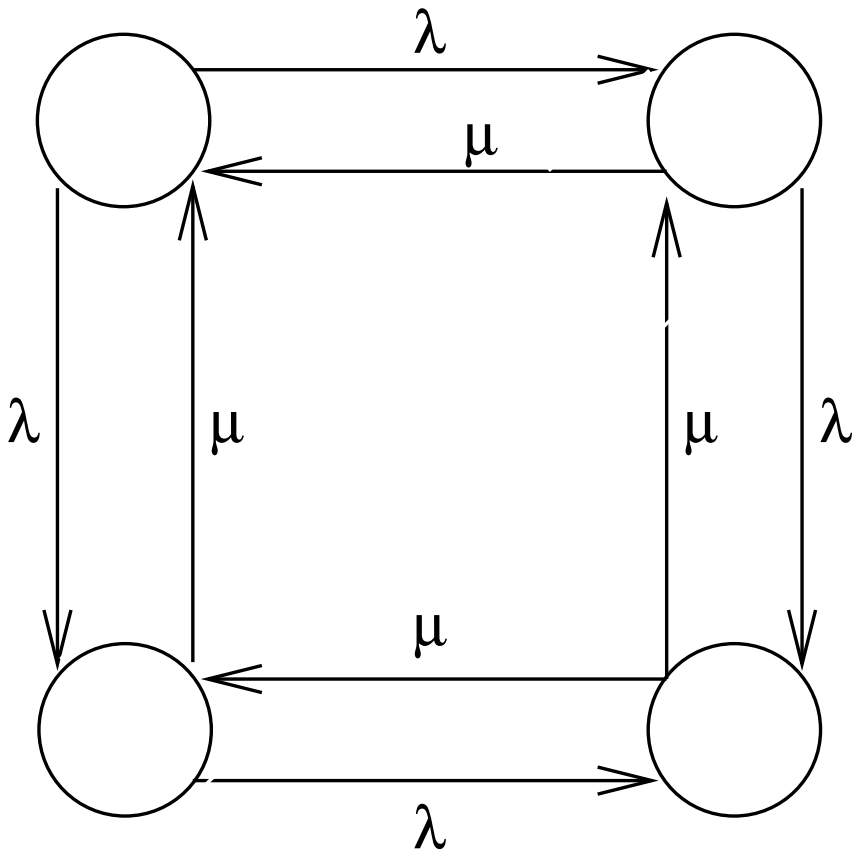
- Direct vs. iterative methods
- Classical Iterative Methods
  - Jacobi
  - Gauss-Seidel
  - SOR
- Scalable Iterative Methods
  - Jacobi Method
  - Krylov Subspace Techniques (e.g. CGS)

Lumpability

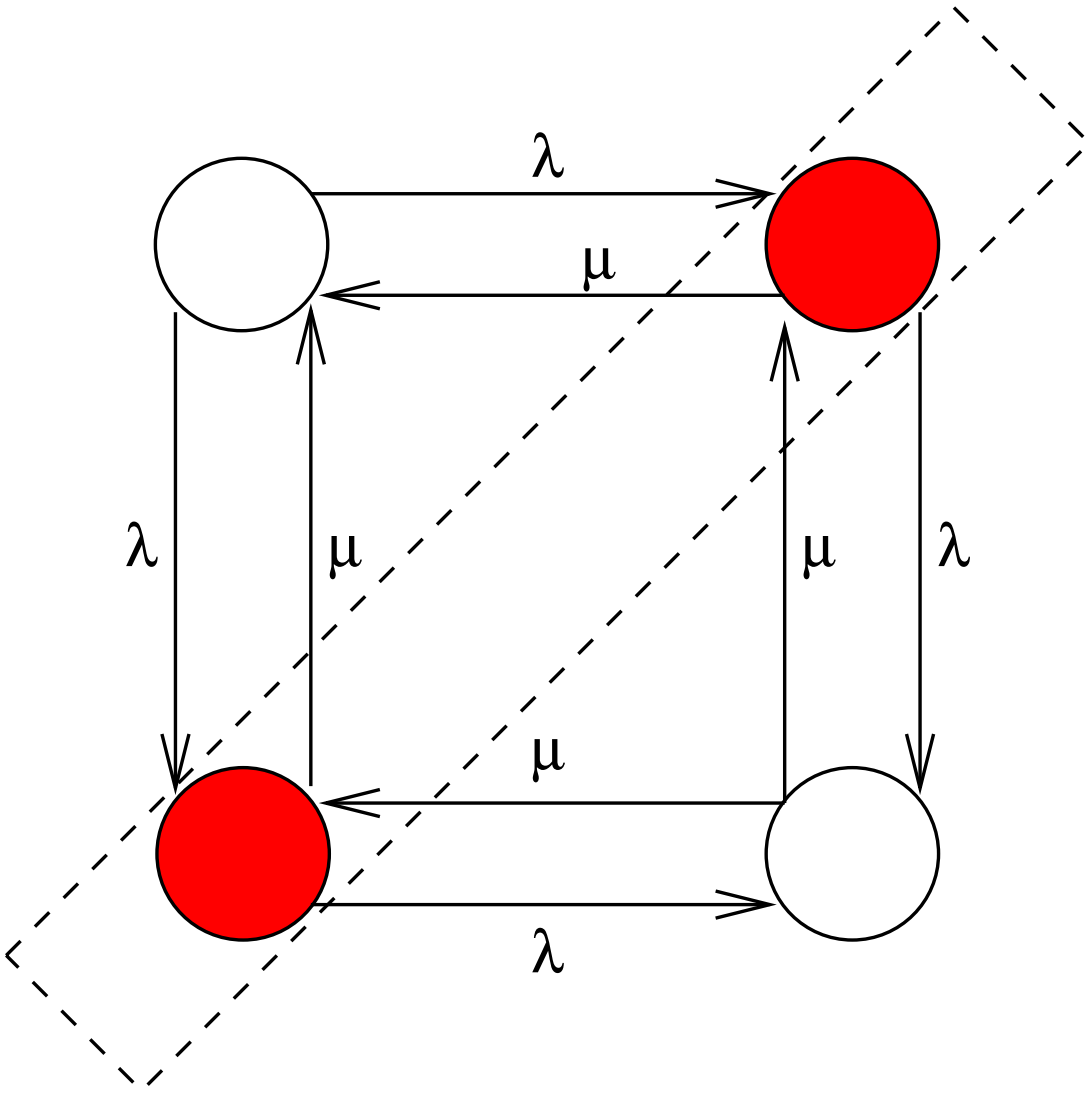
- Allows us to aggregate states to form smaller “quotient” chain
- Can obtain steady-state and transient results for original chain from quotient chain without loss of accuracy
- Symmetry is main source of lumpability
- Given a partition $P = P_1, P_2, \ldots, P_k$ of a CTMC with generator matrix $\mathbf{Q}$, the CTMC is ordinarily lumpable with respect to P iff

$$\forall B, B' \in P, s_1, s_2 \in B : Q(s_1, B') = Q(s_2, B')$$

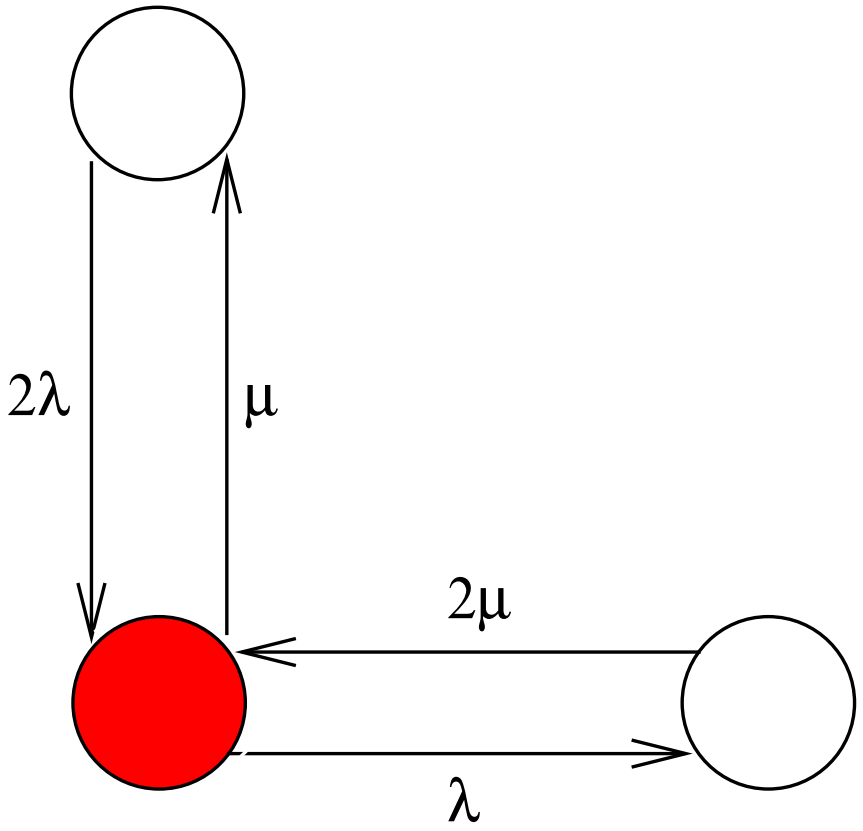
Lumpability



Lumpability



Lumpability



Disk-based Steady-State Solution

- Storing **Q** (and parts of π) from disk and reading elements as needed can yield surprisingly effective solution techniques, esp. when combined with clever block-based numerical methods that can re-use read data
- Effective data production rate often higher than storing **Q** in-core using some compact encoding (e.g. Kronecker, on-the-fly)
- Imposes no structural restrictions
- Deavours and Sanders pioneered disk-based CTMC solution

Disk-based Steady-State Solution (Mehmood and Kwiatkowska)

Integer constant: B (*number of blocks*)
Semaphores: S_1, S_2: occupied
Shared variables: R_0, R_1 *(To read matrix Q blocks into RAM)*
Shared variables: Πbox_0, Πbox_1 *(To read solution vector π blocks into RAM)*

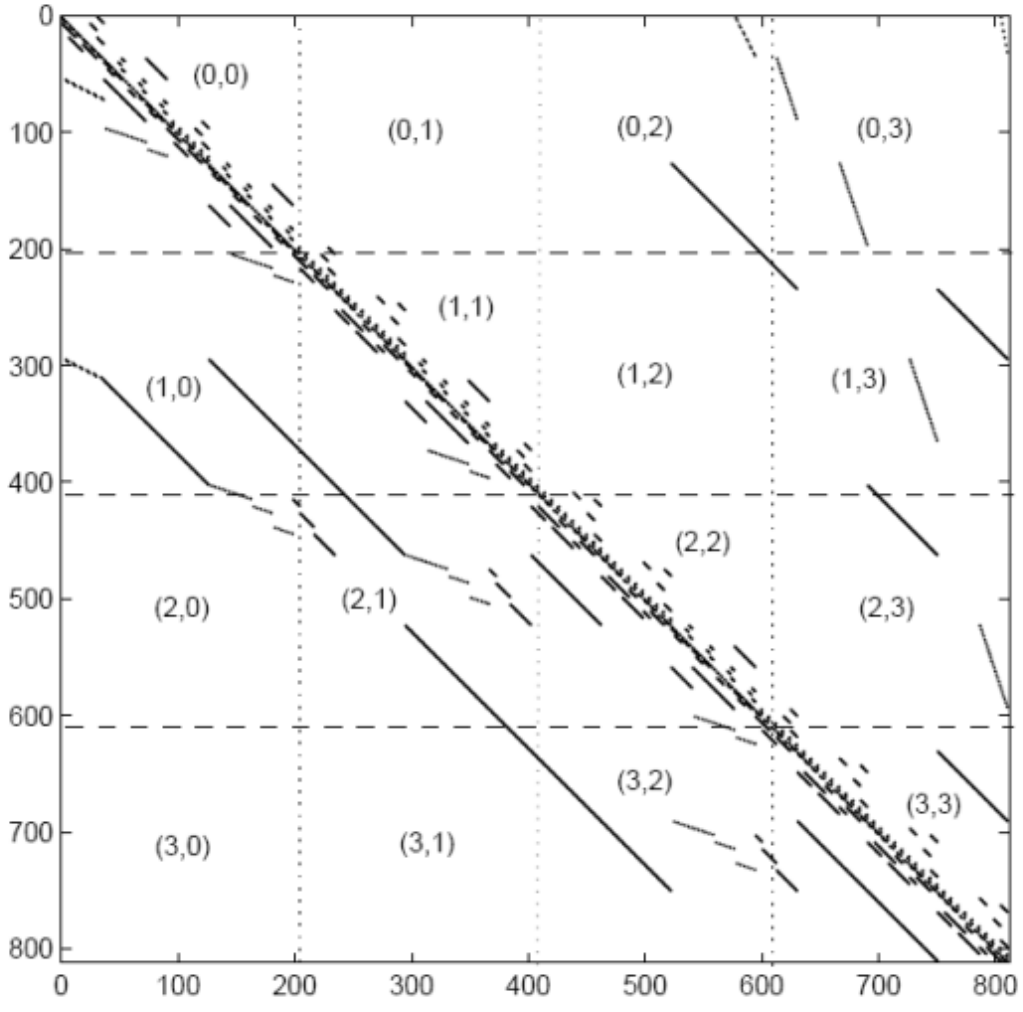
Disk-IO Process

1. Local variable: h, i, j, k
2. $k \leftarrow B-1$
3. **while** not converged
4. **for** $i \leftarrow 0$ to $B-1$
5. **if** $i = 0$ **then** $j \leftarrow B-1$
6. **else** $j \leftarrow i-1$
7. **for** $h \leftarrow 0$ to $B-1$
8. **if** not an *empty* block
9. read Q_{ij} from disk
10. **if** $h \neq 0$
11. read Π_j from disk
12. Signal(S_1)
13. Wait(S_2)
14. **if** $h = 0$
15. write Π_k to disk
16. $k \leftarrow k+1 \mod B$
17. **if** $j = 0$ **then** $j \leftarrow B-1$
18. **else** $j \leftarrow j-1$

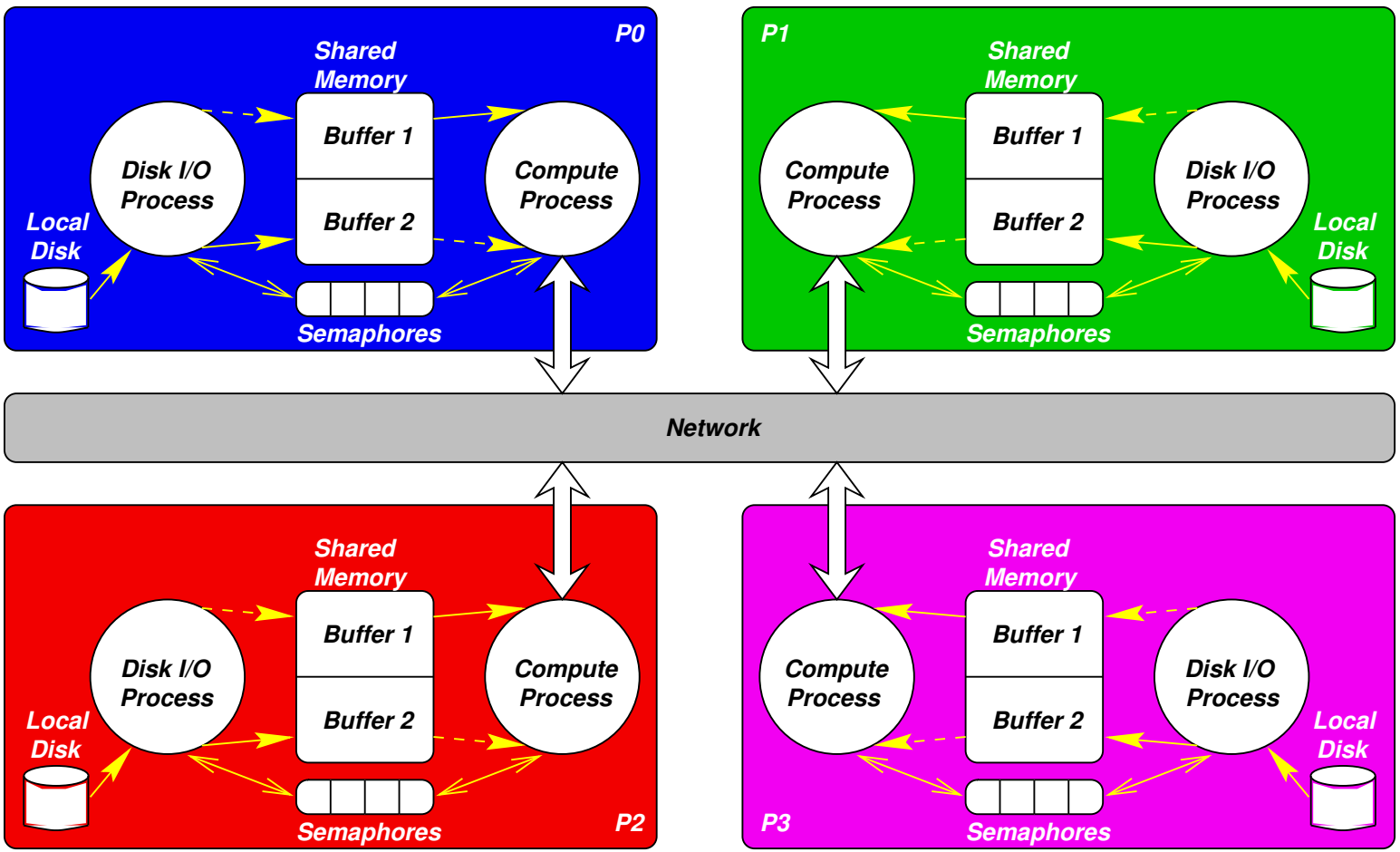
Compute Process

1. Local variable: i, j
2. **while** not converged
3. **for** $i \leftarrow 0$ to $B-1$
4. **if** $i = 0$ **then** $j \leftarrow B-1$
5. **else** $j \leftarrow i-1$
6. **for** 0 to $B-1$
7. Wait(S_1)
8. Signal(S_2)
9. **if** $j \neq i$
10. **if** not an *empty* block
11. Accumulate $Q_{ij}\Pi_j$
12. **else**
13. Update Π_i
14. check for convergence
15. **if** $j = 0$ **then** $j \leftarrow B-1$
16. **else** $j \leftarrow j-1$

Disk-based Steady-State Solution (Mehmood and Kwiatkowska)



Parallel Steady-State Disk-based Solution

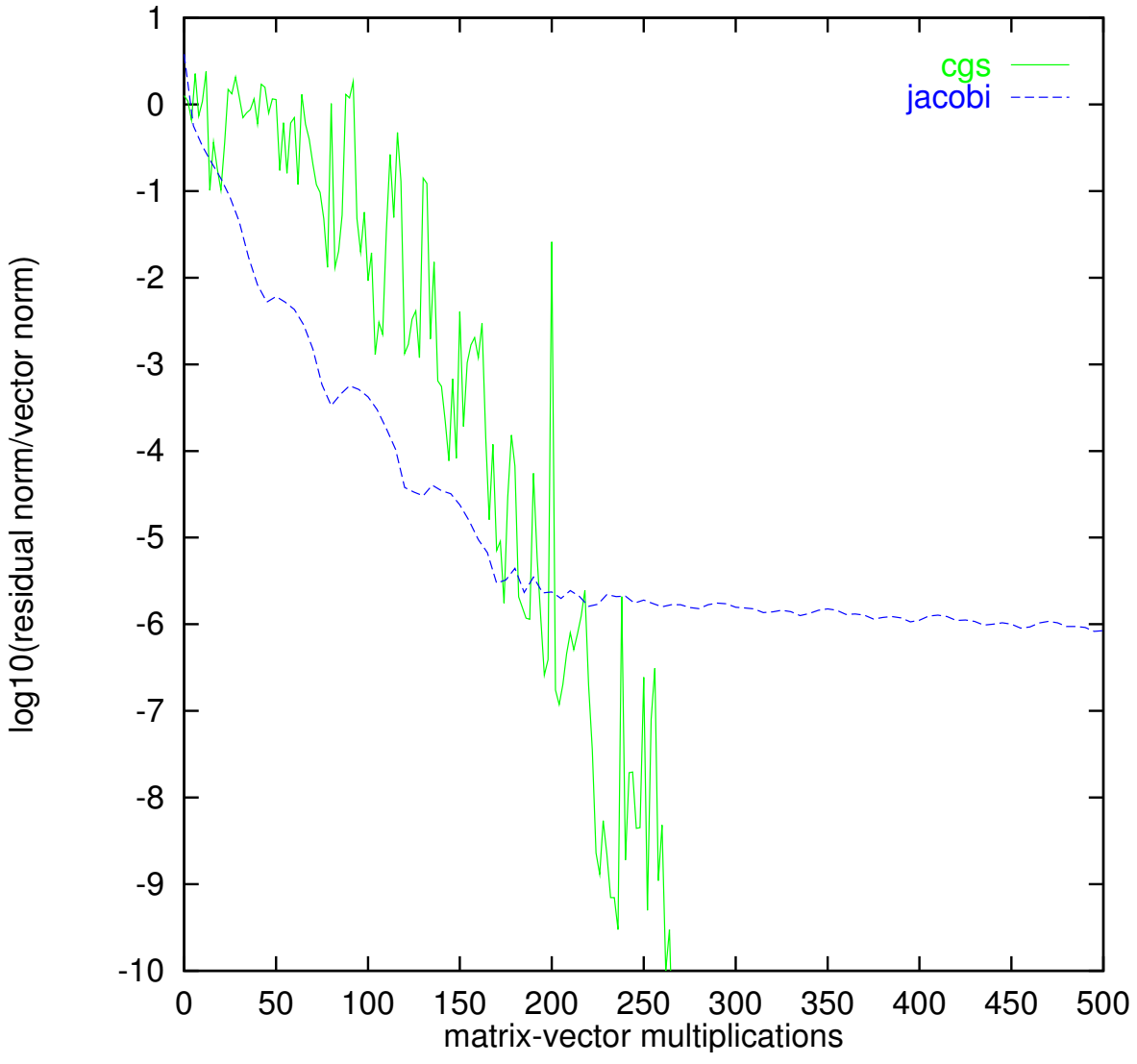


750 million states achieved by Bell and Haverkort (2001)

Courier Disk-based Parallel Steady-State Solution

		k=1	k=2	k=3	k=4	k=5	k=6	k=7	k=8
$p = 1$	Jac time	33.647	278.18	1506.4	5550.3				
	Jac its	4925	4380	4060	3655				
	CGS time	2.1994	21.622	163.87	934.27	29134			
	CGS its	60	81	106	129	157			
	MB/node	20.3	22.1	30.5	60.8	154.0			
$p = 2$	Jac time	29.655	176.62	1105.7	4313.6				
	Jac its	4925	4380	4060	3655				
	CGS time	1.6816	13.119	93.28	509.90	7936.9			
	CGS its	57	84	107	131	148			
	MB/node	20.2	21.1	25.45	41.2	89.7			
$p = 4$	Jac time	25.294	148.45	627.96	3328.3				
	Jac its	4925	4380	4060	3655				
	CGS time	1.2647	8.4109	58.302	322.50	1480.5			
	CGS its	60	80	108	133	159			
	MB/node	20.1	20.6	22.9	31.4	57.5			
$p = 8$	Jac time	38.958	140.06	477.02	1780.9	6585.4			
	Jac its	4925	4380	4060	3655	3235			
	CGS time	1.4074	6.0976	39.999	204.46	934.76	4258.7		
	CGS its	61	82	109	132	155	171		
	MB/node	20.0	20.3	21.7	26.5	41.4	81.6		
$p = 16$	Jac time	41.831	125.68	506.31	1547.9	5703.4	11683	32329	
	Jac its	4925	4380	4060	3650	3235	2325	2190	
	CGS time	3.3505	7.1101	31.322	134.48	577.68	2032.5	13786	141383
	CGS its	60	91	104	132	146	173	179	213
	MB/node	20.0	20.2	21.0	24.1	33.4	58.5	79.8	161

Courier Steady-State Convergence Behaviour



Courier Steady-state Performance Measures

	$k=1$	$k=2$	$k=3$	$k=4$	$k=5$	$k=6$	$k=7$	$k=8$
λ	74.3467	120.372	150.794	172.011	187.413	198.919	207.690	214.477
P_{send}	0.01011	0.01637	0.02051	0.02334	0.02549	0.02705	0.02825	0.02917
P_{recv}	0.98141	0.96991	0.96230	0.95700	0.95315	0.95027	0.94808	0.94638
P_{sess1}	0.00848	0.01372	0.01719	0.01961	0.02137	0.02268	0.02368	0.02445
P_{sess2}	0.92610	0.88029	0.84998	0.82883	0.81345	0.80196	0.79320	0.78642
$P_{transp1}$	0.78558	0.65285	0.56511	0.50392	0.45950	0.42632	0.40102	0.38145
$P_{transp2}$	0.78871	0.65790	0.57138	0.51084	0.46673	0.43365	0.40835	0.38871

Table: Courier Protocol performance measures in terms of the transport window size k.

Parallel Sparse Matrix–Vector Products and Data Partitioning

- Parallel sparse matrix–vector product (or similar) operations form the kernel of many parallel numerical algorithms
- In our context this includes steady-state and passage time calculations that use iterative algorithms for solving very large sparse systems of linear equations.
- The data partitioning strategy adopted (i.e. the assignment of matrix and vector elements to processors) has a major impact on performance, especially in distributed memory environments.

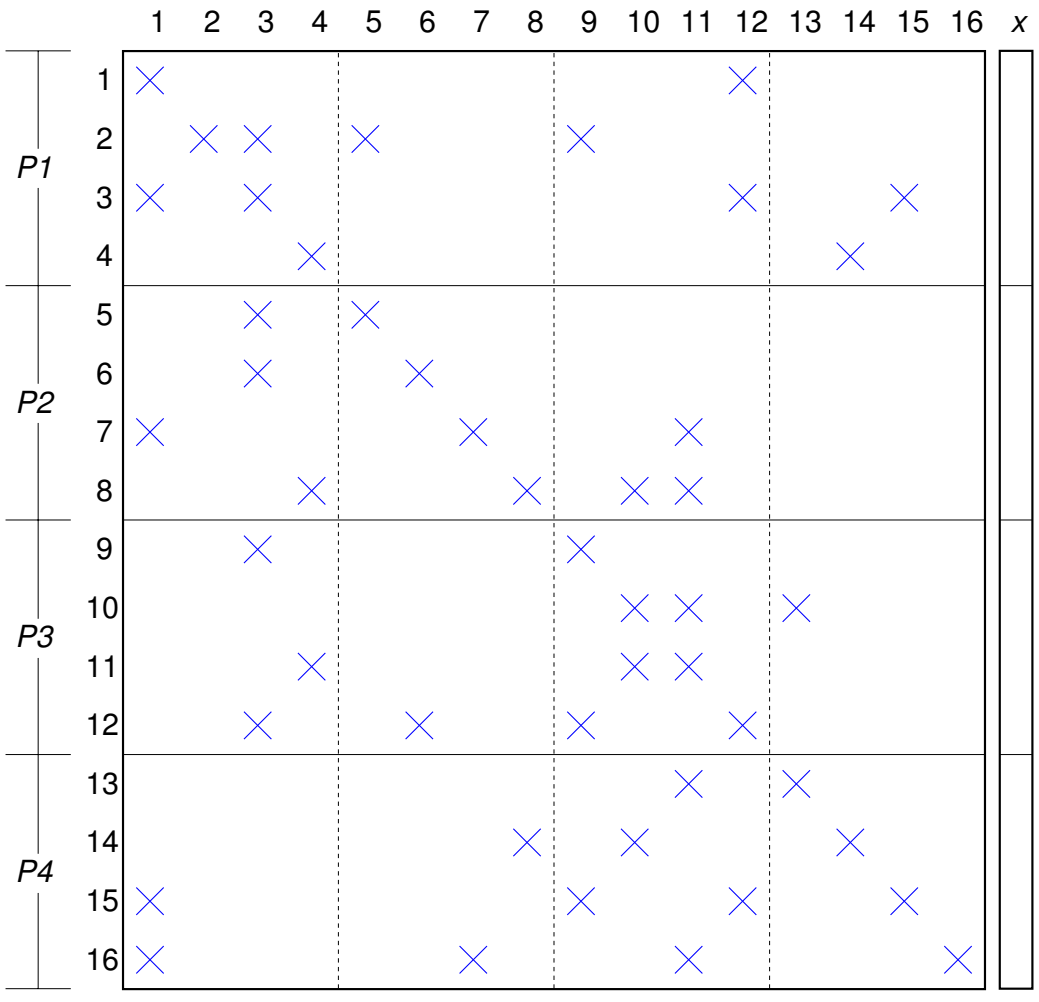
Partitioning Objectives and Strategies

- Aim is to allocate matrix and vector elements across processors such that:
  - computational load is balanced
  - communication is minimised
- Candidate partitioning strategies:
  - naive row (or column) striping
  - mapping of rows (or columns) and corresponding vector elements to processors using 1D graph or hypergraph-based data partitioning
  - mapping of individual non-zero matrix elements and vector elements to processors using 2D hypergraph-based partitioning

Naive Row-Striping

- Assume an $n \times n$ sparse matrix $\mathbf{A}$, an n-vector $\mathbf{x}$ and p processors.
- Simply allocate n/p matrix rows and n/p vector elements to each processor (assuming p divides n exactly).
- If p does not divide n exactly, allocate one extra row and one extra vector element to those processors with rank less than $n \bmod p$.
- What are the advantages and disadvantages of this scheme?

Naive Row-Striping: Example



1D Graph Partitioning

- An $n \times n$ sparse matrix **A** can be represented as an undirected graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$.
- Each row i ($1 \leq i \leq n$) in **A** corresponds to vertex $v_i \in \mathcal{V}$ in the graph.
- The (vertex) weight w_i of vertex v_i is the total number of non-zeros in row i.
- For the edge-set $\mathcal{E}$, edge e_{ij} connects vertices v_i and v_j with (edge) weight:
  - 1 if either one of $|a_{ij}| > 0$ or $|a_{ji}| > 0$,
  - 2 if both $|a_{ij}| > 0$ and $|a_{ji}| > 0$
- Aim to partition the vertices into p mutually exclusive subsets (parts) $\{P_1, P_2, \ldots, P_p\}$ such that *edge-cut* is minimised and load is *balanced*.

1D Graph Partitioning

- An edge e_{ij} is cut if the vertices which it contains are assigned to two different processors, i.e. if $v_i \in P_m$ and $v_j \in P_n$ where $m \neq n$.
- The edge-cut is the sum of the edge weights of cut edges and is an approximation for the amount of interprocessor communication
- Let

$$W_k = \sum_{i \in P_k} w_i \qquad \text{(for } 1 \leq k \leq p\text{)}$$

  denote the weight of part P_k, and $\overline{W}$ denote the average part weight.
- A partition is said to be balanced if:

$$W_k \leq (1 + \epsilon)\overline{W}$$

  for $k = 1, 2, \ldots p$.

1D Graph Partitioning

- Problem of finding a balanced p-way partition that minimizes edge cut is NP-complete.
- But heuristics can often be applied to obtain good sub-optimal solutions.
- Software tools:
  - CHACO
  - METIS
  - ParMETIS
- Once partition has been computed, assign matrix row i to processor k if $v_i \in P_k$.

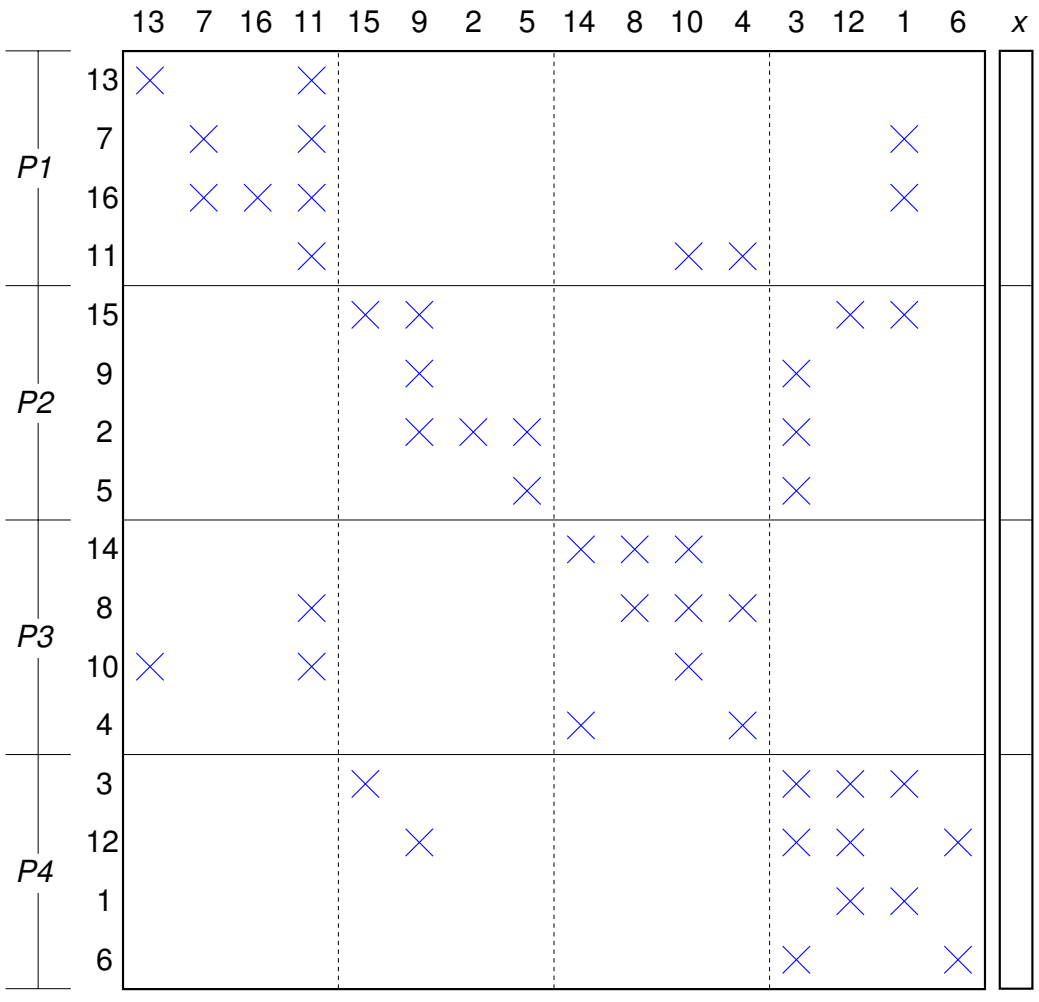
1D Graph Partitioning: Example

- Consider the graph corresponding to the sparse matrix **A** of the previous example.
- Graph partitioner recommends four parts as follows:

$$P_1 = \{v_{13}, v_7, v_{16}, v_{11}\} \quad P_2 = \{v_{15}, v_9, v_2, v_5\}$$

$$P_3 = \{v_{14}, v_8, v_{10}, v_4\} \quad P_4 = \{v_3, v_{12}, v_1, v_6\}$$

1D Graph Partitioning: Example



1D Hypergraph Partitioning

- An $n \times n$ sparse matrix **A** can be represented as a hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{N})$.
- $\mathcal{V}$ is a set of vertices and $\mathcal{N}$ is a set of nets or hyperedges. Each $n \in \mathcal{N}$ is a subset of the vertex set $\mathcal{V}$.
- Each row i ($1 \leq i \leq n$) in **A** corresponds to vertex $v_i \in \mathcal{V}$.
- Each column j ($1 \leq i \leq n$) in **A** corresponds to net $N_j \in \mathcal{N}$. In particular $v_i \in N_j$ iff $a_{ij} \neq 0$.
- The (vertex) weight w_i of vertex v_i is the total number of non-zeros in row i.
- Given a partition $\{P_1, P_2, \ldots, P_p\}$, the connectivity λ_j of net N_j denotes the number of different parts spanned by N_j. Net N_j is cut iff $\lambda_j > 1$.

1D Hypergraph Partitioning

- The cutsize or hyperedge cut of a partition is defined as:

$$\sum_{N_j \in \mathcal{N}} (\lambda_j - 1)$$

- Aim is to minimize the hyperedge cut while maintaining the balance criterion (which is same as for graphs).
- Again, problem of finding a balanced p-way partition that minimizes the hyper-edge cut is NP-complete, but heuristics can be used to find sub-optimal solutions.
- Software tools:
  - hMETIS
  - PaToH
  - Par*k*way

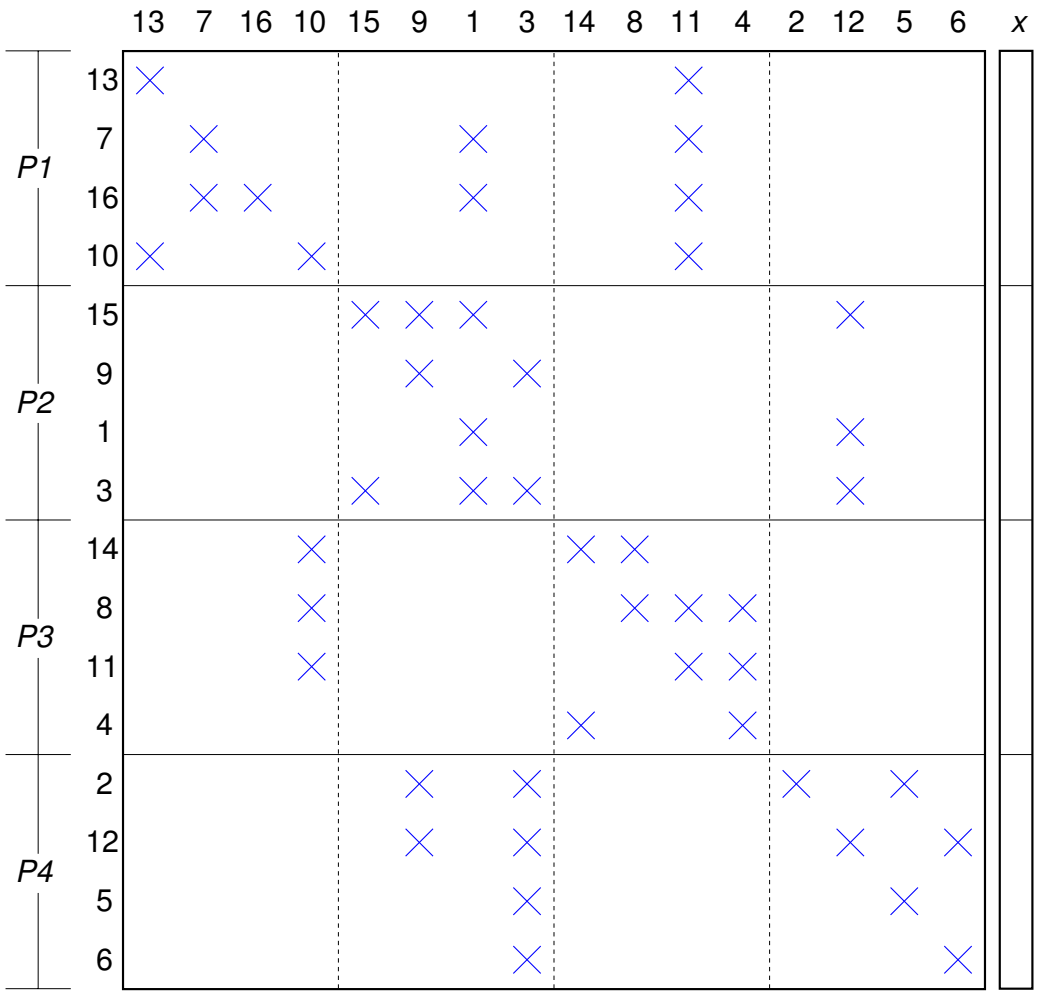
1D Hypergraph Partitioning: Example

- Consider the hypergraph corresponding to the sparse matrix **A** of the previous example.
- Hypergraph partitioner recommends four parts as follows:

$$P_1 = \{v_{13}, v_7, v_{16}, v_{10}\} \quad P_2 = \{v_{15}, v_9, v_1, v_3\}$$

$$P_3 = \{v_{14}, v_8, v_{11}, v_4\} \quad P_4 = \{v_2, v_{12}, v_5, v_6\}$$

1D Hypergraph Partitioning: Example



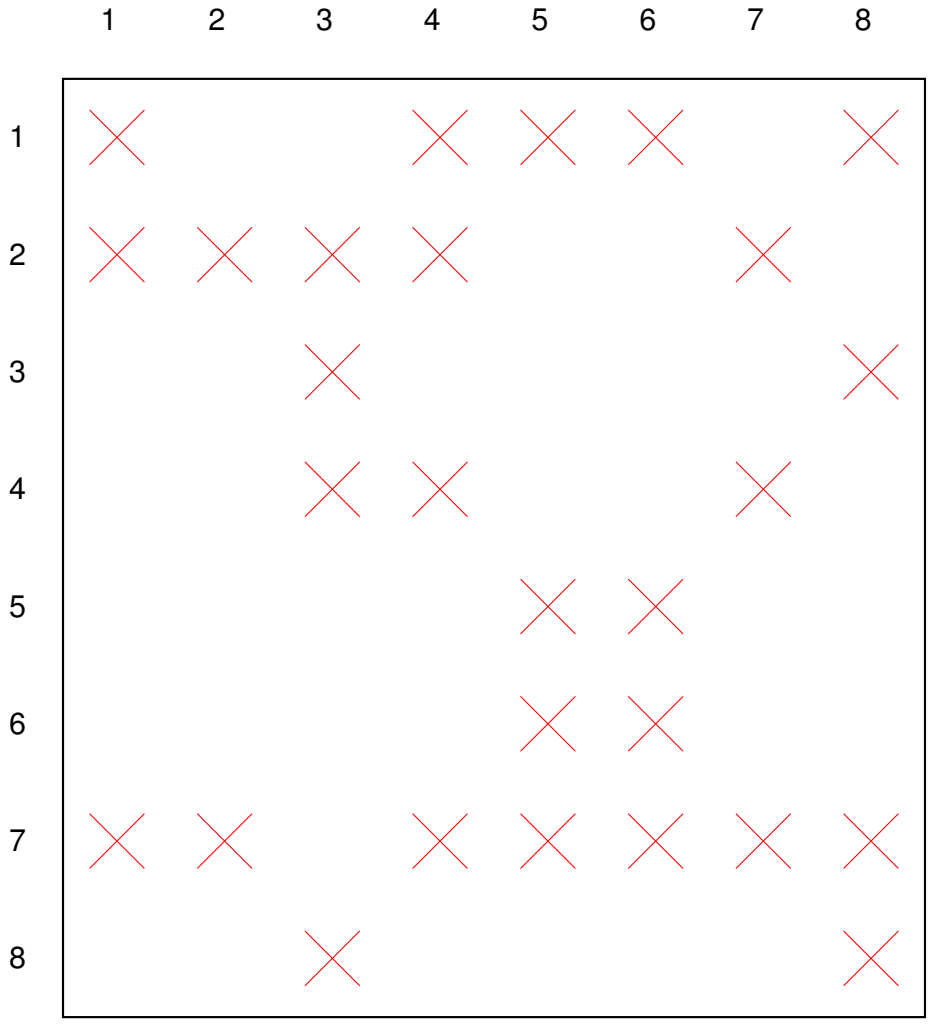
2D Hypergraph Partitioning

- The most general mapping possible is to allocate individual non-zero matrix elements and vector elements to processors.
- General form of parallel sparse matrix–vector multiplication follows four stages, where each processor:
  1. sends its x_j values to processors that possess a non-zero a_{ij} in column j,
  2. computes the products $a_{ij}x_j$ for its non-zeros a_{ij} yielding a set of contributions b_{is} where s is a processor identifier.
  3. sends b_{is} contributions to the processor that is assigned x_i.
  4. adds up received contributions for assigned vector elements, so $b_i = \sum_{s=0}^{p-1} b_{is}$

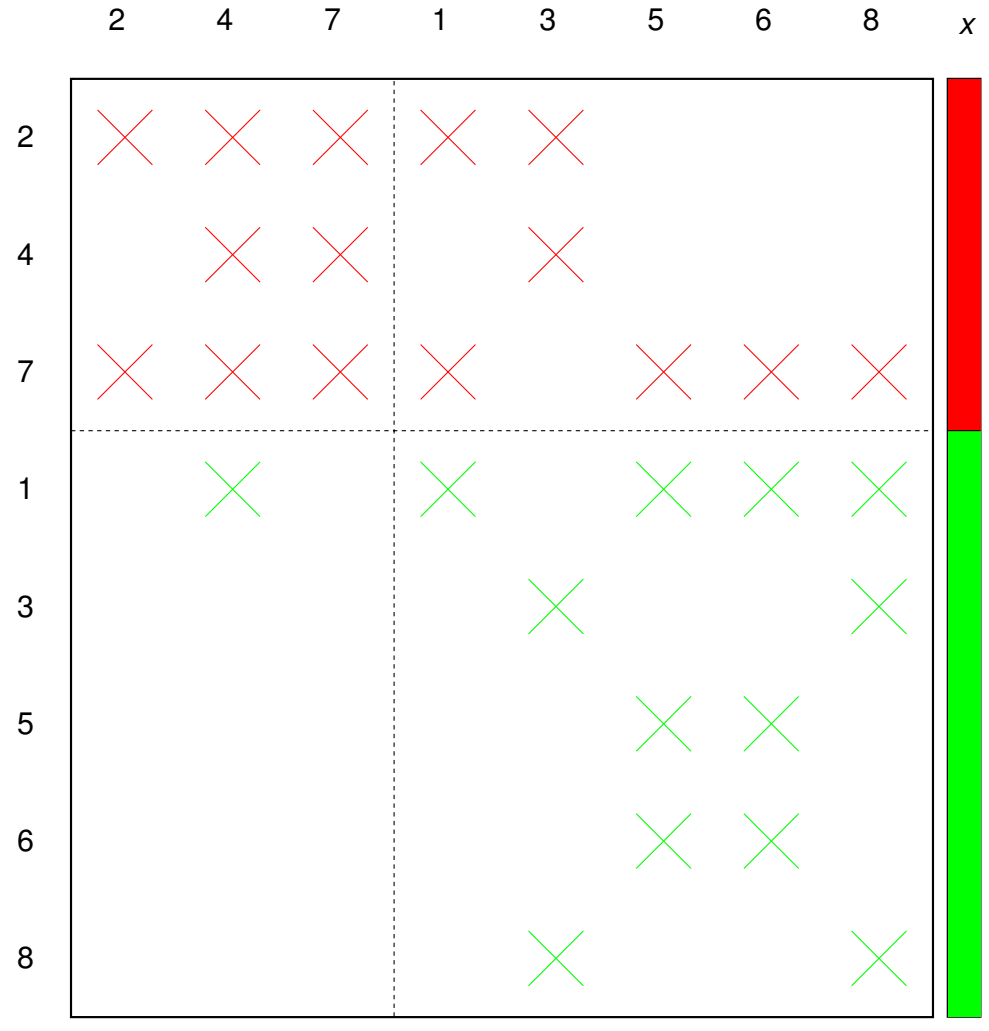
2D Hypergraph Partitioning

- Each non-zero is modelled by a vertex (weight 1) in the hypergraph; if a_{ji} is 0 then add "dummy" vertex (weight 0).
- Model Stage 1 comms cost by net whose constituent vertices are the non-zeros of column j. Model Stage 3 comms cost by net whose constituent vertices are the non-zeros of row i.
- Now partition hypergraph into p parts such that the $k-1$ metric is minimised, subject to balance constraint.
- Assign non-zero elements to processors according to partition.
- Assign b_i's to processors appropriately according to whether row i and/or column i hyperedge is cut (if any).

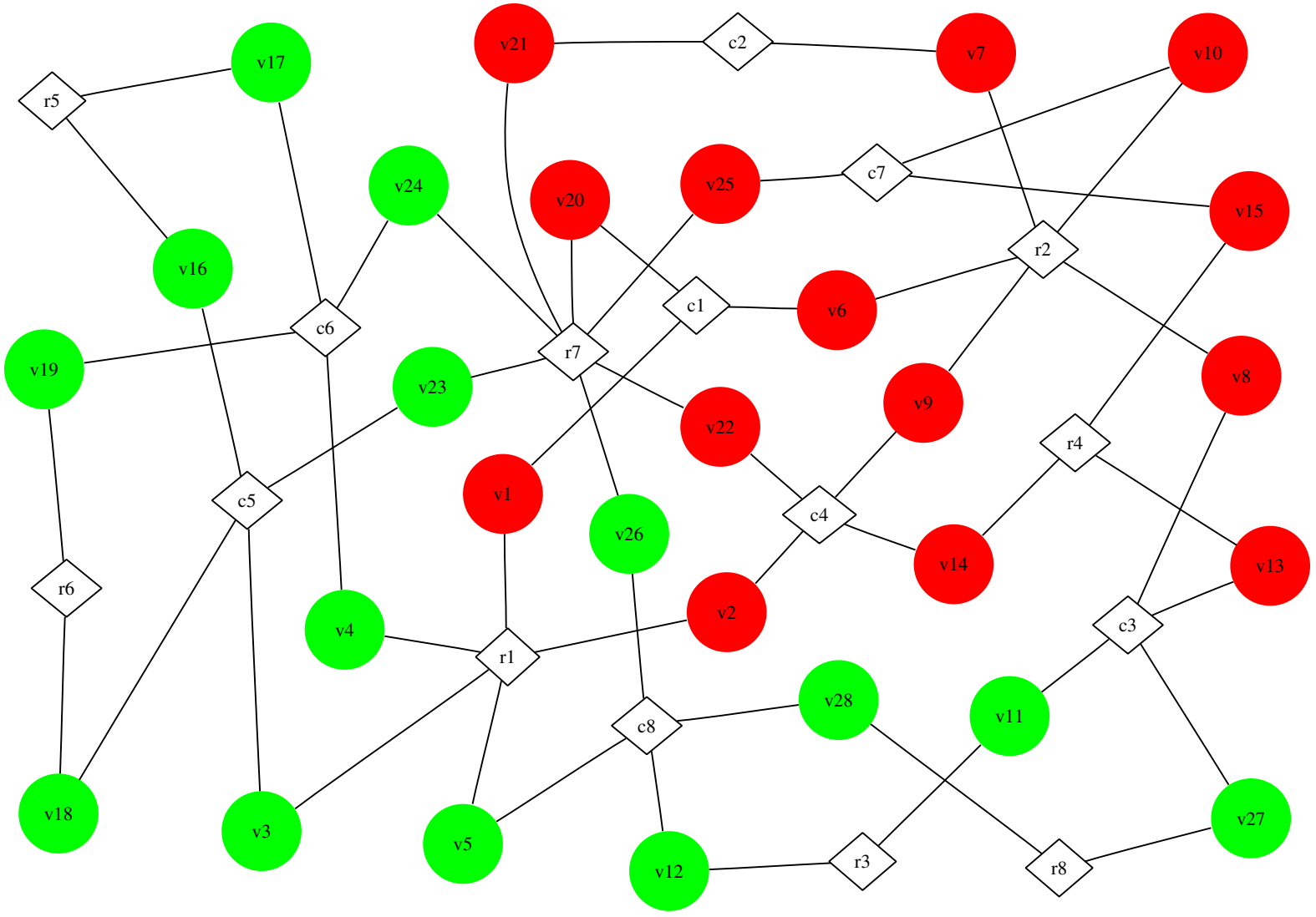
2D Hypergraph Partitioning: Example



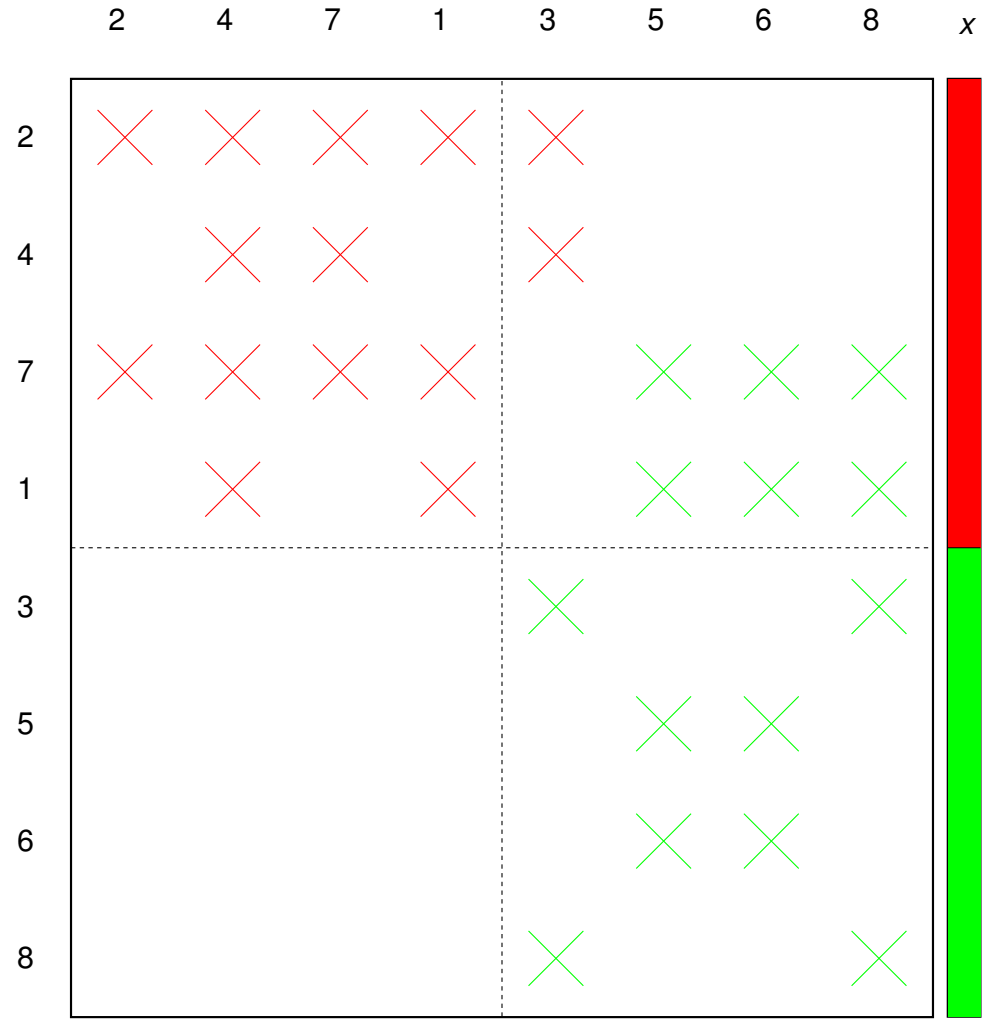
2D Hypergraph Partitioning: Example



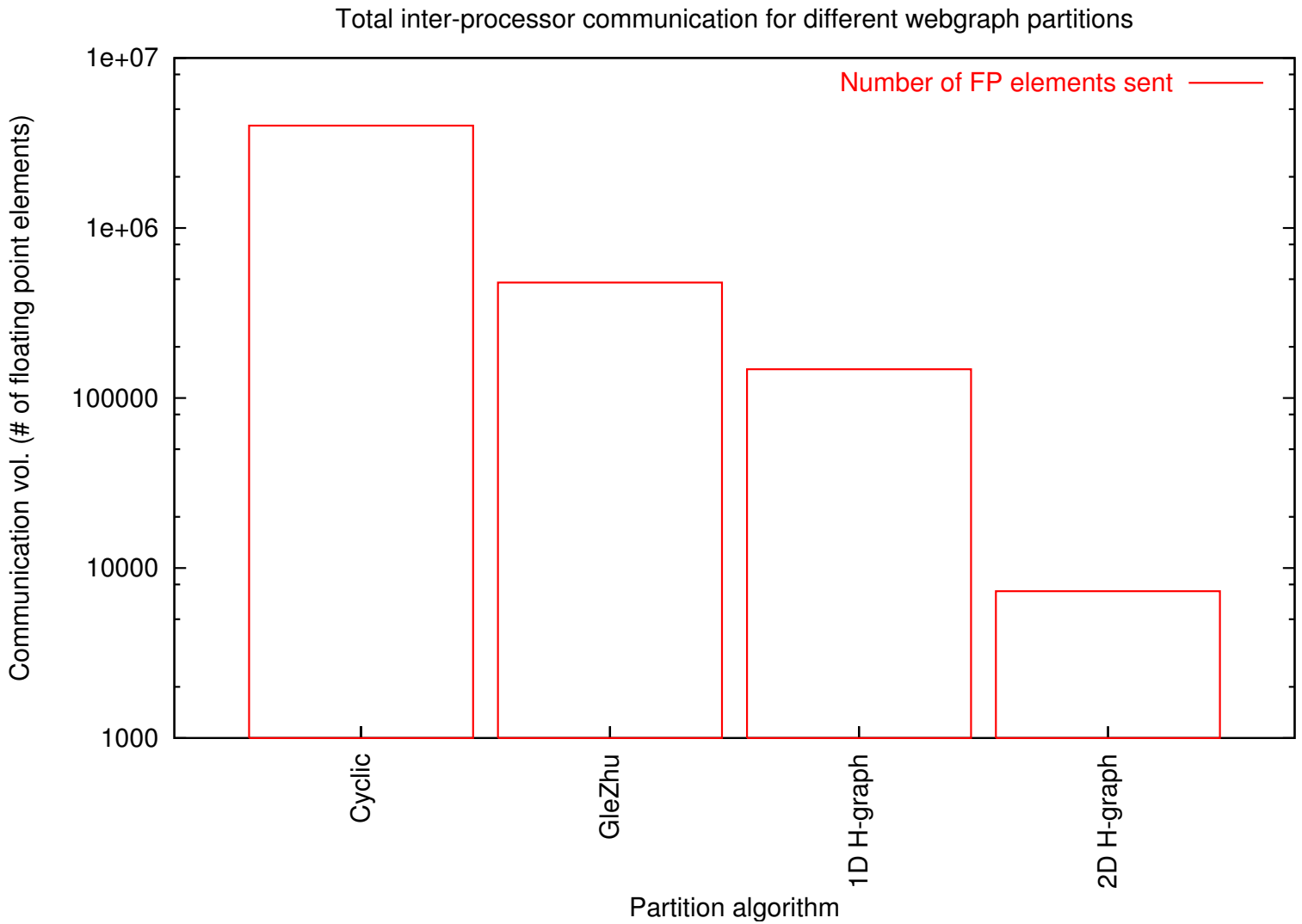
2D Hypergraph Partitioning: Example



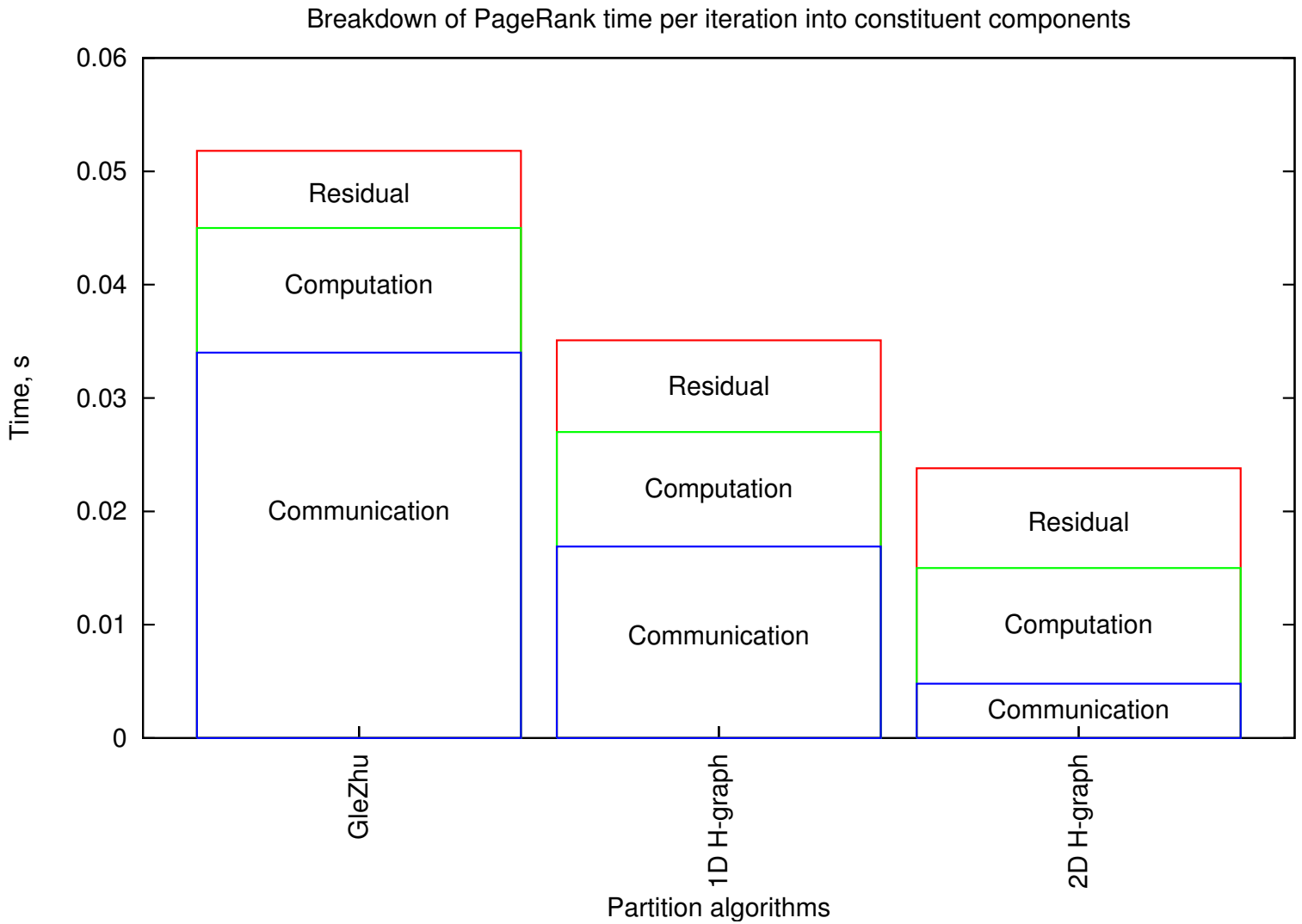
2D Hypergraph Partitioning: Example



Application to PageRank Computation



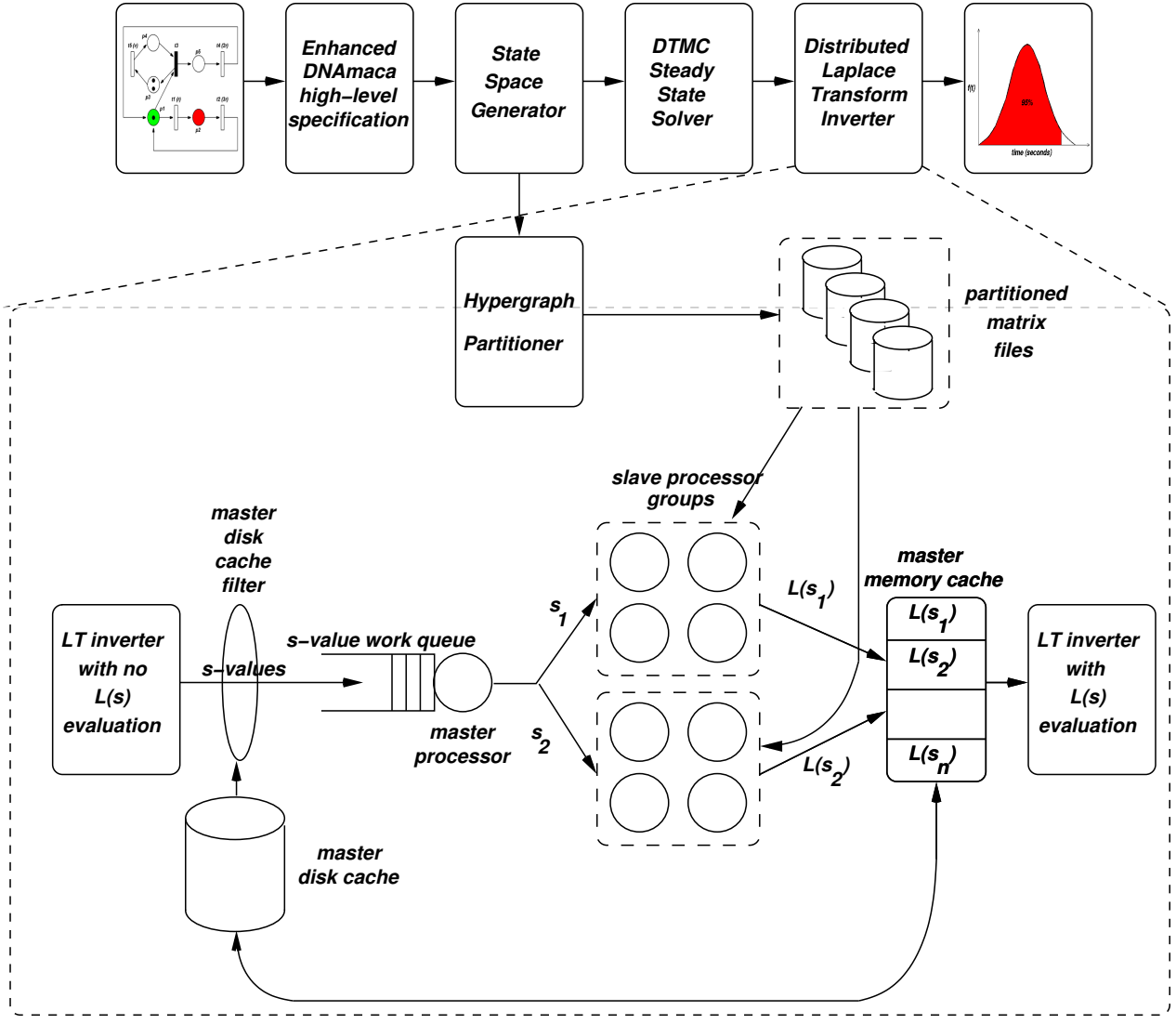
Application to PageRank Computation



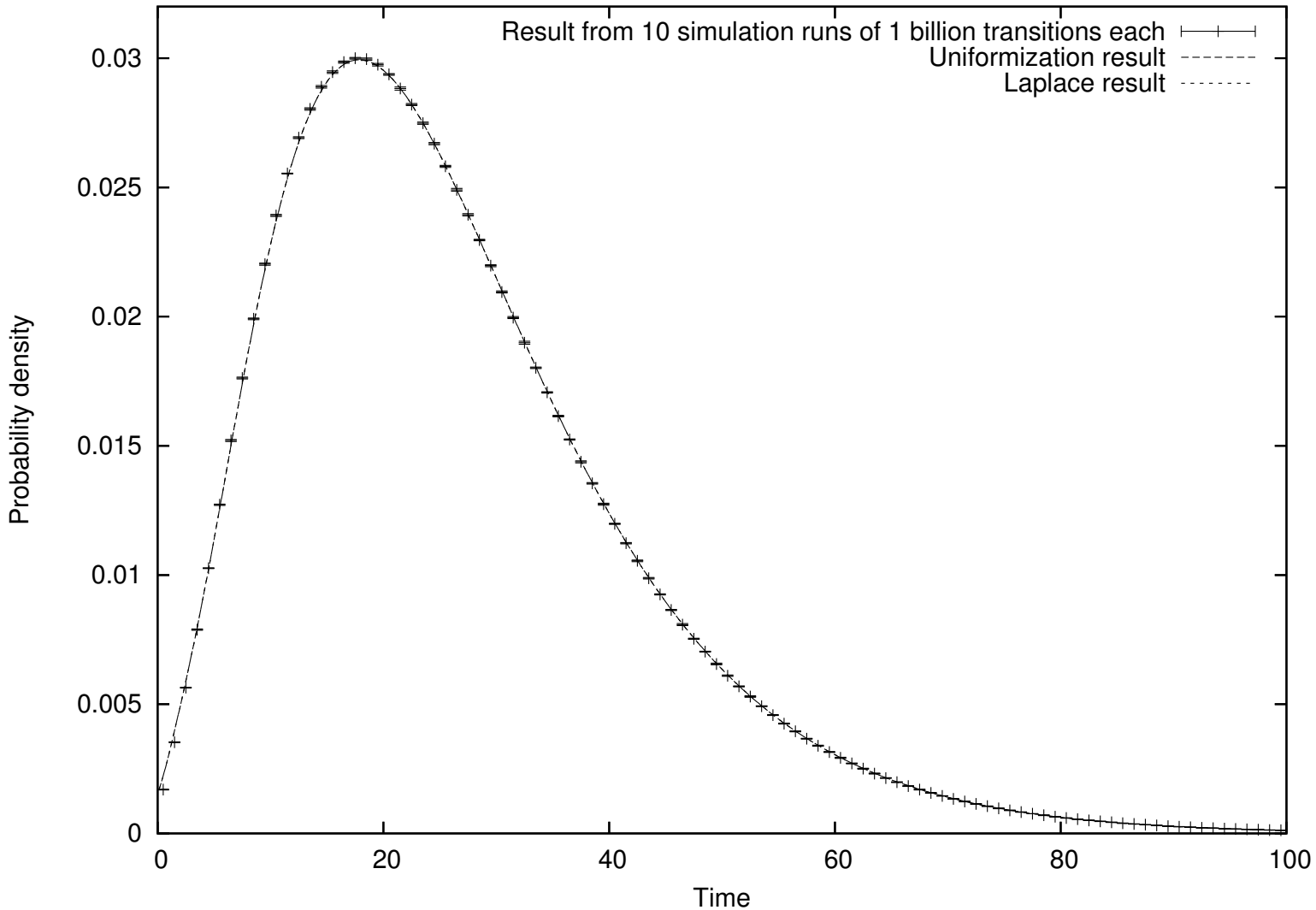
Comparison of Graph and Hypergraph Partitioning Techniques

- A graph partition aims to minimise the number of non-zero entries in off-diagonal matrix blocks.
- A hypergraph partition aims to minimise actual communication; the partition may have more off-diagonal non-zero entries than a graph partition but these will tend to be column aligned.
- Either sort of partitioning is preferable to a naive or random partition.
- For very large matrices, parallel partitioning tools are required (e.g. ParMeTiS for graphs, and Par*k*way or Zoltan for hypergraphs).

The SMARTA Tool



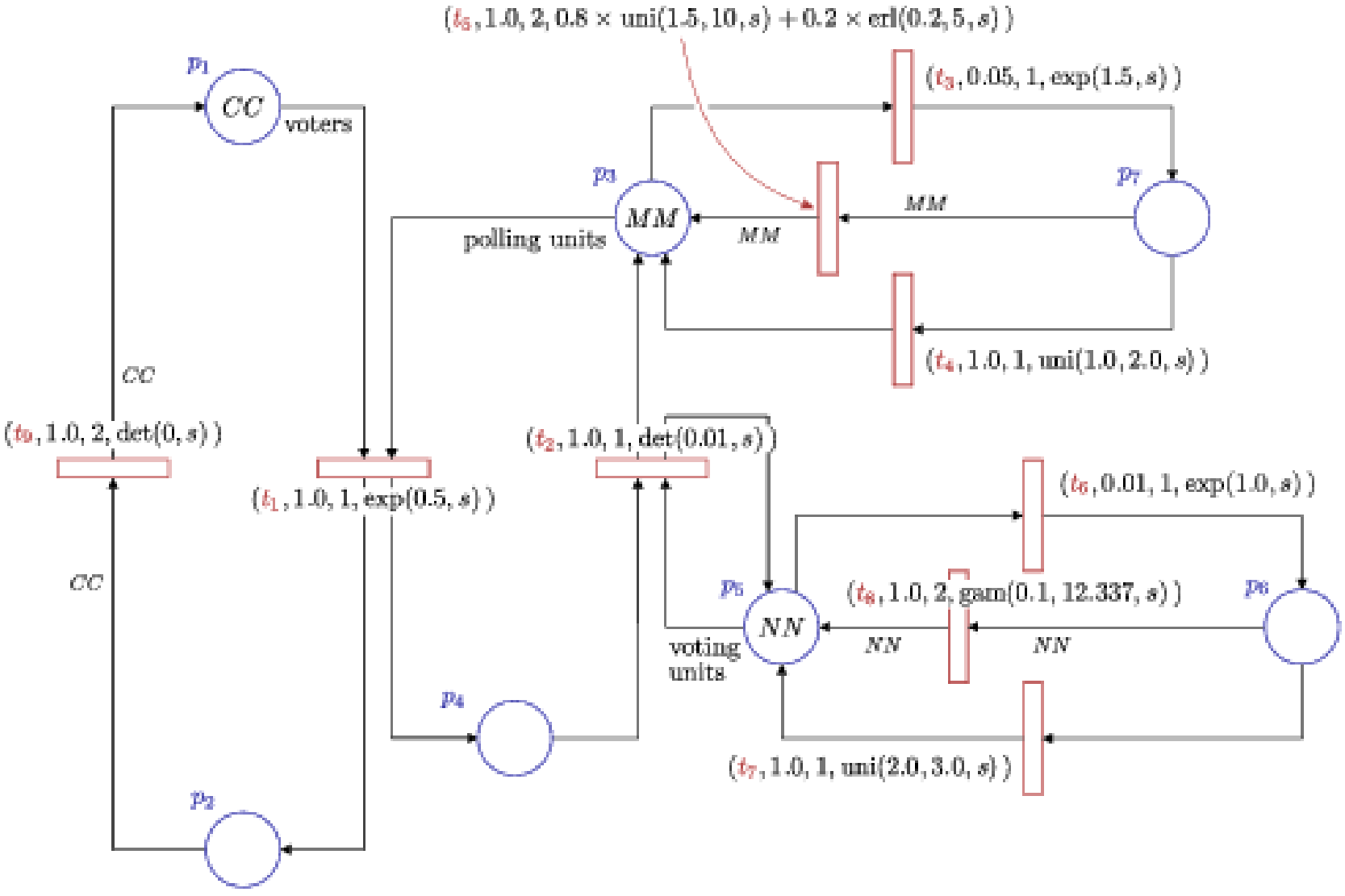
Courier Response Time Density Function



Some Tools Developed at Imperial

- **DNAmaca** (Data Network Architectures Markov Chain Analyser): Steady-state Analysis of Markov Chains (most suited to GSPNs and related formalisms)
- **smca** (Semi-Markov Chain Analyser): Steady-state Analysis of Semi-Markov Chains (SM-SPN models)
- **HYDRA** (Hypergraph-based Distributed Response-time Analyser): Uniformisation-based Passage-time Analysis of Markov Chains
- **SMARTA** (Semi-Markov Response-time Analyser): Numerical Laplace-transform Inversion-based Passage-time Analysis of Semi-Markov Chains
- **ipc** (Imperial Pepa Compiler): Specification and Analysis of PEPA models via DNAmaca and HYDRA
- **Par*k*way** (Parallel *k*-way Hypergraph Partitioner)

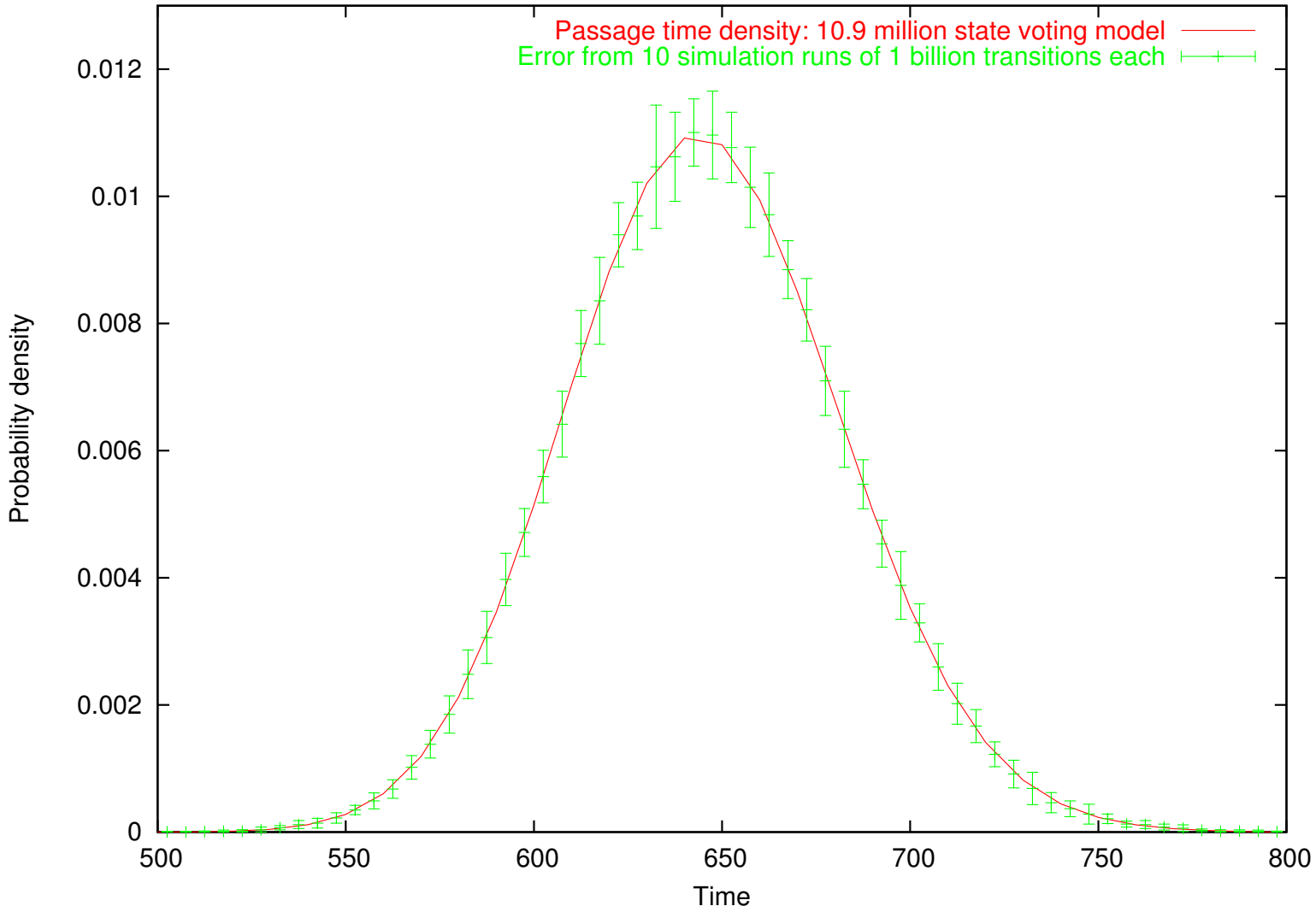
Description



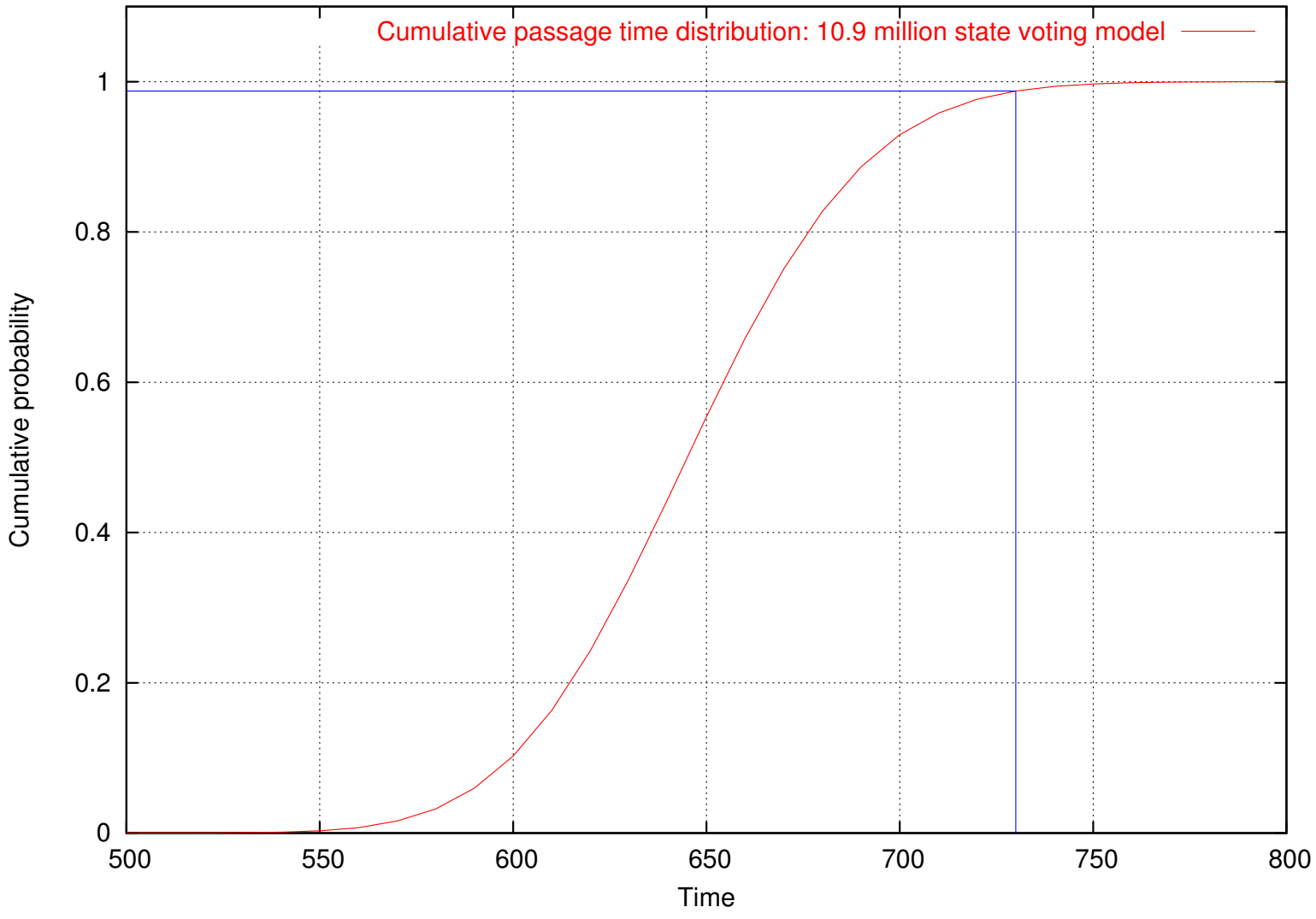
State Space

System	*CC*	*MM*	*NN*	States
1	60	25	4	106 540
2	100	30	4	249 760
3	125	40	4	541 280
4	150	40	5	778 850
5	175	45	5	1 140 050
6	300	80	10	10 999 140

Response Time Density Function



Response Time Cumulative Distribution Function

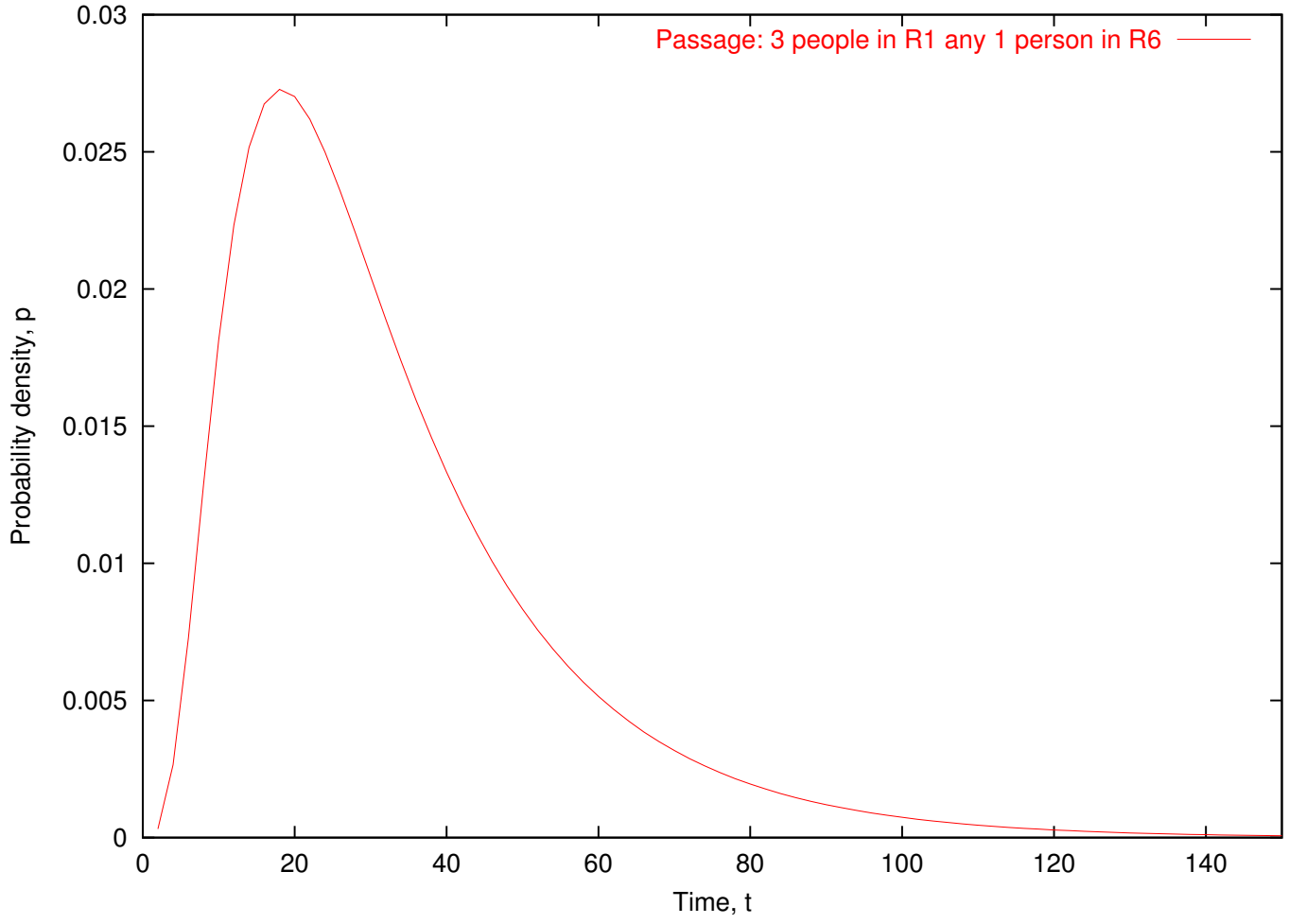


Description

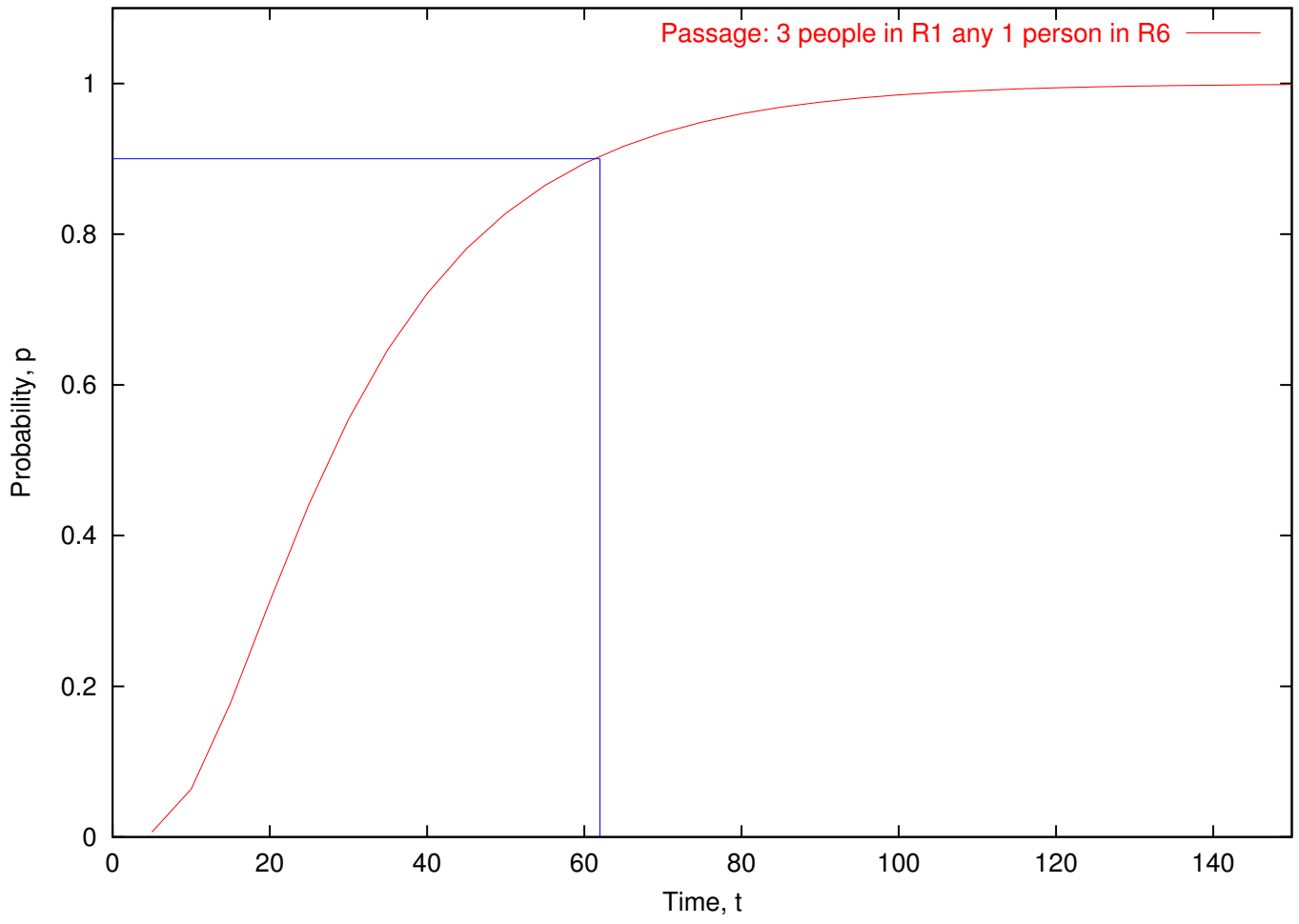
$$
\begin{aligned}
Person_1 &= (reg_1, r).Person_1 + (move_2, m).Person_2 \\
Person_i &= (move_{i-1}, m).Person_{i-1} + (reg_i, r).Person_i \\
&\quad + (move_{i+1}, m).Person_{i+1} \\
&\quad : 1 < i < N \\
Person_N &= (move_{N-1}, m).Person_{N-1} + (reg_N, r).Person_N \\
\\
Sensor_i &= (reg_i, \top).(rep_i, s).Sensor_i \quad : 1 \leq i \leq N \\
\\
Dbase_i &= \textstyle\sum_{j=1}^{N} (rep_j, \top).Dbase_j \quad : 1 \leq i \leq N \\
\\
Sys &= \textstyle\prod_{j=1}^{M} Person_1 \underset{Reg}{\bowtie} \prod_{j=1}^{N} Sensor_j \underset{Rep}{\bowtie} Dbase_1
\end{aligned}
$$

where $Reg = \{reg_i \mid 1 \leq i \leq N\}$ and $Rep = \{rep_i \mid 1 \leq i \leq N\}$

Response Time Density Function



Response Time Cumulative Distribution Function



Thank you! Any (more) questions?