Distributed Response Time Analysis of GSPN Models with MapReduce

Oliver J. Haggarty   William J. Knottenbelt   Jeremy T. Bradley
Department of Computing, Imperial College London,
180 Queen's Gate, London, SW7 2BZ, United Kingdom
{ojh06, wjk, jb}@doc.ic.ac.uk

Abstract

Generalised Stochastic Petri nets (GSPNs) are widely used in the performance analysis of computer and communications systems. Response time densities and quantiles are often key outputs of such analysis. These can be extracted from a GSPN's underlying semi-Markov process using a method based on numerical Laplace transform inversion. This method typically requires the solution of thousands of systems of complex linear equations, each of rank *n*, where *n* is the number of states in the model. For large models substantial processing power is needed and the computation must therefore be distributed.

This paper describes the implementation of a Response Time Analysis module for the Platform Independent Petri net Editor (PIPE2) which interfaces with Hadoop, an open source implementation of Google's MapReduce distributed programming environment, to provide distributed calculation of response time densities in GSPN models. The software is validated with analytically calculated results as well as simulated ones for larger models. Excellent scalability is shown.

Keywords: Generalised Stochastic Petri nets, MapReduce, Response Time Analysis

1 INTRODUCTION

The complexity of modern distributed systems continues to rise rapidly. It is therefore increasingly important to model these systems prior to their implementation to ensure they behave correctly. In this context, Generalised Stochastic Petri nets (GSPNs) are a popular graphical modelling formalism which are both intuitive and flexible. GSPNs have an underlying semi-Markov process which can be analysed for many qualitative and quantitative factors.

The focus of the present paper is on techniques for extracting response time densities and quantiles from GSPN models. Given their increasing use in Service Level Agreements, these are important performance measures for many computer and communication systems, such as web servers, communication networks and stock market trading systems. In particular, we describe the creation of a new Response Time Analysis module for the Platform Independent Petri net Editor (PIPE2) [3]. PIPE2[1] is an open source Petri net editor and analyser developed by several generations of students at Imperial College London as well as several external contributors. The module accepts a set of start and target markings (defined by logical expressions which describe the number of tokens that should be present on selected places) and outputs graphs of the corresponding response time density and (optionally) the cumulative distribution function of the time taken for the system to pass from the start markings into any of the target markings. The analysis makes use of a method based on numerical Laplace transform inversion, whereby we convolve the state sojourn times along all paths from the set of start markings to the target markings [8]. This involves the solution of many systems of complex linear equations, each of rank *n*, where *n* is the size of the GSPN's state space. For large *n* the calculations require a great deal of processing power. Consequently, we distribute the processing over a cluster of computers by interfacing PIPE2 with Hadoop, an open source implementation of Google's MapReduce distributed programming environment. This paradigm offers excellent scalability and robust fault tolerance.

The remainder of this paper is organised as follows. Section 2 presents relevant background material relating to Generalised Stochastic Petri nets and their response time analysis. Section 3 describes Hadoop, an open source implementation of the MapReduce distributed programming model. Section 4 describes the design and integration of an Hadoop-based Response Time Analysis module into the PIPE2 Petri net editor. Finally, Section 5 validates the module using small models with known analytical results, as well as larger models where results had been produced by simulation. The software is shown to work with model sizes with in excess of two million states, and to scale well with increasing analysis cluster size. Section 6 concludes.

2 BACKGROUND THEORY

Petri nets are a graphical formalism for describing concurrency and synchronisation in distributed systems. In their simplest form, they are also known as Place-Transition nets. These consist of a number of places, which may contain tokens, connected by transitions. A transition is *enabled* and can *fire* if the input places of the transition contain a certain number of tokens. These numbers are defined in a backwards incidence matrix whose rows correspond to places and columns to transitions. In so firing, the specified number of tokens are then removed from each place. A forward inci-

[1] Available from `http://pipe2.sourceforge.net`.

dence matrix similarly defines the number of tokens to add to each place following the transition.

A *marking* (or state) is a vector of integers representing the number of tokens on each place of the model. The *reachability set* or *state space* of a Place-Transition net is the set of all possible markings that can be reached from a given initial marking. The *reachability graph* shows the connections between these markings.

Generalised Stochastic Petri nets (see e.g. Figures 5 and 6) extend Place-Transition nets by incorporating timing information. A timed transition t_i has an exponentially distributed firing rate λ_i. Immediate transitions have priority over timed transitions and fire in zero time. Markings that enable timed transitions only are known as *tangible*, while markings that enable any immediate transition are called *vanishing*. The sojourn time in a tangible marking M_i is exponentially distributed with parameter $\mu_i = \sum_{k \in en(M_i)} \lambda_k$ where $en(M_i)$ is the set of transitions enabled by marking M_i. The sojourn time in vanishing markings is zero.

Formally, [2]:

Definition 2.1 *A Generalised Stochastic Petri net is an 8-tuple GSPN* $= (P, T, I^-, I^+, M_0, T_1, T_2, W)$. $P = \{p_1, ..., p_{|P|}\}$ *is a finite and non-empty set of places* $T = \{t_1, ..., t_{|T|}\}$ *is a finite and non-empty set of transitions.* $P \cap T = \emptyset$. $I^-, I^+ : P \times T \to \mathbb{N}_0$ *are the backward and forward incidence functions, respectively.* $M_0 : P \to \mathbb{N}_0$ *is the initial marking.* $T_1 \subseteq T$ *is the set of timed transitions.* $T_2 \subset T$ *is the set of immediate transitions;* $T_1 \cap T_2 = \emptyset$ *and* $T = T_1 \cup T_2$. $W = (w_1, ..., w_{|T|})$ *is an array whose entry* $w_i \in \mathbb{R}^+$ *is* either *a rate of a negative exponential distribution specifying the firing delay, when transition* t_i *is a timed transition,* or *a firing weight, when transition* t_i *is an immediate transition.*

We further define p_{ij} to be the probability that M_j is the next marking entered after marking M_i and, for tangible marking M_i, $q_{ij} = \mu_i p_{ij}$, i.e. q_{ij} is the instantaneous transition rate into marking M_j from marking M_i. These can be represented as a generator matrix $\mathbf{Q}$ whose rows correspond to M_i and columns to M_j. A GSPN is therefore isomorphic to a Semi Markov Process. As such, it has an embedded discrete-time Markov Chain (EMC) which can be described by a square matrix whose elements p_{ij} are given by $p_{ij} = \lim_{\tau \to \infty} H_{ij}(\tau)$ where $Hij(t)$ is the sojourn time distribution in state i when the next state is j and τ is the sojourn time.

2.1 Response Time Analysis using Numerical Laplace Transform Inversion

If we first consider a GSPN whose state space does not contain any vanishing states, the definition of the first passage time from a single source marking i to a non-empty set of target markings $\vec{j}$ is given by:

$$T_{i\vec{j}} = \inf\{u > 0 : M(u) \in \vec{j}, N(u) > 0, M(0) = i\}$$

where $M(u)$ is the marking of the GSPN at time u and $N(u)$ is the number of transitions which have fired by time u.

When studying GSPNs whose state spaces include vanishing states we define the passage time as:

$$T_{i\vec{j}} = \inf\{u > 0 : N(u) \geq M_{i\vec{j}}\}$$

where $M_{i\vec{j}} = \min\{m \in \mathbb{Z}^+ : X_m \in \vec{j} \mid X_0 = i\}$; here X_m is the state of the system after the mth transition firing [4].

There are two main methods for computing first passage time (and hence response time) densities in Markov models: those based on Laplace transforms and their inversion [1, 13] and those based on uniformisation [15, 14]. The latter, as implemented in the HYDRA [9, 5] tool, are more efficient but have difficulty in supporting vanishing states, especially when these are specified as the source or target states of a passage. In this paper we therefore chose the former approach, as implemented in the SMARTA tool [4, 7].

To find this passage time we must convolve the state sojourn time densities for all paths from i to $j \in \vec{j}$. In the Laplace domain as we can take advantage of the convolution property which states that the convolution of two functions is equal to the product of their Laplace transforms. We perform a first-step analysis to find the Laplace transform of the relevant density. This process can be thought of as first finding the probability density of moving from state i to its set of direct successor states $\vec{k}$ and then convolving it with the probability density of moving from $\vec{k}$ to the set of target states $\vec{j}$. Vanishing markings have a sojourn time density of 0, with probability 1, which results in their Laplace transform equalling 1 for all values of s. If $\mathcal{L}_{i\vec{j}}(s)$ is the Laplace transform of the density function $f_{i\vec{j}}(t)$ of the passage time variable $T_{i\vec{j}}$, then we can express $\mathcal{L}_{i\vec{j}}(s)$ as:

$$\mathcal{L}_{i\vec{j}}(s) = \begin{cases} \displaystyle\sum_{k \notin \vec{j}} \left(\frac{q_{ik}}{s + \mu_i}\right) \mathcal{L}_{k\vec{j}}(s) + \sum_{k \in \vec{j}} \left(\frac{q_{ik}}{s + \mu_i}\right) & \text{if}\, i \in \mathcal{T} \\ \sum_{k \notin \vec{j}} p_{ik} \mathcal{L}_{k\vec{j}}(s) + \sum_{k \in \vec{j}} p_{ik} & \text{if}\, i \in \mathcal{V} \end{cases}$$

where $\mathcal{T}$ is the set of tangible markings and $\mathcal{V}$ is the set of vanishing markings.

This system of linear equations can also be expressed in matrix–vector form. If, for example, we wish to find the passage time from state i to the set of states $\vec{j} = \{M_1, M_3\}$, where $\mathcal{T} = \{M_1, M_3, \ldots, M_n\}$ and $\mathcal{V} = \{M_2\}$, then:

$$\begin{pmatrix} s - q_{11} & -q_{12} & 0 & \cdots & -q_{1n} \\ 0 & 1 & 0 & \cdots & -p_{2n} \\ 0 & -q_{32} & s - q_{33} & \cdots & -q_{3n} \\ 0 & \vdots & 0 & \ddots & \vdots \\ 0 & -q_{n2} & 0 & \cdots & -q_{nn} \end{pmatrix} \mathbf{L} = \begin{pmatrix} q_{13} \\ p_{21} + p_{23} \\ q_{31} \\ \vdots \\ q_{n1} + q_{n3} \end{pmatrix} \qquad (1)$$

where $\mathbf{L} = (\mathcal{L}_{1\vec{j}}(s), \ldots, \mathcal{L}_{nj}(s))$. If we wish to calculate the passage time from multiple source states, denoted by the vector $\vec{i}$, the Laplace transform of the passage time density is given by:

$$\mathcal{L}_{\vec{i}\vec{j}}(s) = \sum_{k \in \vec{i}} \alpha_k \mathcal{L}_{k\vec{j}}(s)$$

where α_k is the steady-state probability that the GSPN is in state k at the starting instant of the passage. α_k is given by:

$$\alpha_k = \begin{cases} \pi_k / \sum_{n \in \vec{i}} \pi_n & \text{if} \, k \in \vec{i} \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

where π_k is the kth element of the steady-state probability vector π of the GSPN's underlying embedded Markov Chain.

Now that we have the Laplace transform of the passage time, we must invert it to get the density of interest in the real domain. To do this we can use Euler inversion [1] which allows us to perform the inversion numerically, without having to perform the integration of a complex number. It works by evaluating the Laplace transform $f^*(s)$ at various s-values determined by the value(s) of t at which we wish to evaluate $f(t)$. From these results it approximates the inverse Laplace transform of $f^*(s)$, i.e. $f(t)$. Formally:

$$f(t) \approx \frac{e^{A/2}}{2t} \mathrm{Re}\left(f^*\left(\frac{A}{2t}\right)\right) + \frac{e^{A/2}}{2t} \sum_{k=1}^{\infty} (-1)^k \mathrm{Re}\left(f^*\left(\frac{A + 2k\pi i}{2t}\right)\right) \tag{3}$$

where $A = 19.1$ is a constant that controls the discretisation error. This equation describes the summation of an alternating series, the convergence of which can be accelerated by employing Euler summation.

3 THE MAPREDUCE ENVIRONMENT

MapReduce was devised by Google researchers Dean and Ghemawat as a programming model, with an associated implementation, to facilitate the generation and processing of large data sets on clusters of commodity machines [6]. Whilst traditionally applied to text processing applications, it has become an increasingly popular tool for scientific data processing [10].

MapReduce was intended to allow reliable and efficient distributed programs to be written by developers with little prior experience of writing distributed applications. The framework presented to the developer is inspired by primitive functions of the Lisp programming language, whereby computations are split into a Map task and a Reduce task, both of which the developer is responsible for writing. We can summarise the paradigm as:

Map `(k1,v1)` → `list(k2,v2)`

Reduce `(k2,list(v2))` → `list(v2)`

The Map function is called multiple times, taking an input key/value pair of type `k1` and `v1` and performing some user defined processing to produce a list of intermediate key/value pairs of type `k2` and `v2`. The MapReduce framework then collects together all values associated with the same key to produce a number of key/list pairs - `k2,list(v2)`. Each of these are passed into a Reduce function and the values processed in some way such that a new list of values are produced. Typically this list contains zero or one elements though. Depending on the implementation this is output along with the intermediate key as a key/value pair.

It should be noted that the typing of the keys and values is important. The input keys and values can be from a different domain to the intermediate keys and values (i.e. `k1` and `k2` can be different types). However, the intermediate keys and values must be of the same type as the output keys and values.

3.1 Hadoop Implementation

There are a number of implementations of Google's MapReduce programming model, including Google's own, written in C++ and discussed in [6]. Different implementations can be tailored for the systems they are intended to run on, such as large networks of commodity PCs or powerful, multi-processor, shared-memory machines. In this section we will introduce Hadoop, an open-source Java implementation of the MapReduce model.

Hadoop consists of both the MapReduce framework and the Hadoop Distributed File System (HDFS), reminiscent of the Google File System (GFS). A distributed filesystem uses the local drives of networked computers to store data whilst making it available to all machines connected to the network. Hadoop is designed to be run on large, extensible clusters of commodity PCs and has been demonstrated to run on clusters of up to 2000 machines.

HDFS consists of three main processes: the Namenode, the Secondary Namenode and a number of Datanodes. The Namenode runs on a single master machine in the cluster and stores details of which machines make up the cluster and where each block is stored on which machines. It also handles replication. The Secondary Namenode is an optional back-up process for the Namenode. Datanode processes run on all other machines in the cluster (slaves). They communicate with the Namenode and handle requests to store blocks of data on the machine's local hard disk. They also update the Namenode as to the location of blocks and their current status.

The MapReduce framework is comprised of a single JobTracker and a number of TaskTrackers. The JobTracker process runs on a single, master machine (often the same as the Namenode) and can be thought of as the controller of the cluster. Users submit their MapReduce jobs to the JobTracker, which then splits the work between various machines in the cluster. A TaskTracker process runs on each machine in the cluster. It communicates with the JobTracker and is assigned

Map or Reduce tasks when it is available.

3.2 MapReduce Job Execution Overview

In order to give a clear picture of how Hadoop works we shall now describe the execution of a typical MapReduce job on the Hadoop platform. When the user submits their MapReduce program to the JobTracker the first step is to split the input data (often consisting of many files) into *M* splits of between 16 and 128 MB in size. There are *M* Map tasks and *R* Reduce tasks per job; both values can be specified by the user. When a TaskTracker receives an instruction to run a Map task from the JobTracker it spawns a TaskTrackerChild process to carry out the work. It then continues to listen for further instructions, thereby allowing multiple tasks to be run on multiprocessor or multicore machines. The TaskTrackerChild's first step is to read a copy of the task's associated input split from the HDFS. It parses this for key/value pairs before calling the Map function for each pair. After performing some user defined calculations, the Map function writes intermediate key/value pairs to the local disk. There are typically many of these per Map. These pairs are partitioned into *R* regions, each region containing key/value pairs for a subset of the keys. At the end of the Map task the TaskTracker informs the JobTracker it has completed its task and gives the location of the intermediate pairs it has created.

A TaskTracker that has been assigned a Reduce task will copy all the intermediate pairs from a single partition region to its local disk. These pairs will be distributed amongst the local disks of all workers that have run a Map task. Once copied, it sorts the pairs on their keys. A call to the Reduce function is made for each unique key and the list of associated values is passed in. The output of the reduce function is appended to an output file associated with the Reduce task. *R* output files will be produced per job.

It is often the case that a single Map task will produce many key/value pairs with the same key. Ordinarily, these will all need to be individually copied to the machine running the corresponding Reduce task. However, to reduce network bandwidth the MapReduce framework allows a Combiner function to be run on the same machine that ran the Map task, which partially merges intermediate data before it is transferred. Network bandwidth is further reduced by taking advantage of replication within the HDFS, whereby each block of data is stored on a number of local disks for fault tolerance reasons. When a machine requires some data the Namenode gives it the location on the machine storing the data which is closest on the network path. The MapReduce framework further takes advantage of this property by attempting to run Map tasks on machines that are already storing a copy of the corresponding file split on their local disk. This concept of "bringing the computation to the data" can have great performance benefits in a distributed environment.

The key mechanism for handling failure of nodes in the MapReduce cluster is re-execution. While the JobTracker is a very important part of the system and is a single point of failure, the chances of that one machine failing are low. Hadoop therefore currently does not have any fault tolerance procedures for it and the entire job must be re-executed. In a large cluster of slaves the chances of a node failing are much higher. To counter this, the JobTracker periodically pings each TaskTracker. If it does not receive a response within a certain time it marks the node as failed and re-schedules all Map tasks carried out by that node since the job started. This is necessary as the intermediate results for those tasks will be stored on that node's local hard-disk, which is now inaccessible. This allows a job to continue with minimal re-execution.

Hadoop offers a comprehensive HTML based monitoring console giving details of the health of nodes in the cluster and the progress of jobs which are running. Detailed timings of tasks and the nodes they have run on are reported allowing for early detection of problematic nodes.

4 PIPE2 RESPONSE TIME ANALYSIS

The Platform Independent Petri net Editor (PIPE) was created in 2002 at Imperial College London as a group project for MSc (Computing Science) students. The motivation was to produce an intuitive Petri net editor compliant with the latest XML Petri net standard, the Petri Net Mark-up Language (PNML). Subsequent projects and contributions from external developers have extended the program to version 2.5, adding support for GSPNs, further analysis features and improved GUI performance including an animation mode [3]. An important feature of PIPE2 is the facility for pluggable analysis modules. That is, an externally compiled analysis class can be dropped into a Module folder and the ModuleLoader class then uses Java reflection to integrate it into the application at run-time. All module classes must implement a predefined Module interface:

```
public void run(PNMLData petrinet) { ... }
public String getName() { ... }
```

Existing modules support tasks such as steady-state analysis, reachability graph visualisation and invariant analysis. A number of other modules are also currently being developed.

4.1 Overview of Module

Figure 1 shows the user-facing input window of the PIPE2 Response Time Analysis module. The upper panel allows the user to specify details of the analysis they wish to perform by entering logical expressions to identify sets of start and target markings and the range of *t* points to calculate over. There are also options to calculate the PDF and/or the CDF and whether the processing should be done locally or distributed using MapReduce. The bottom panel provides comprehensive

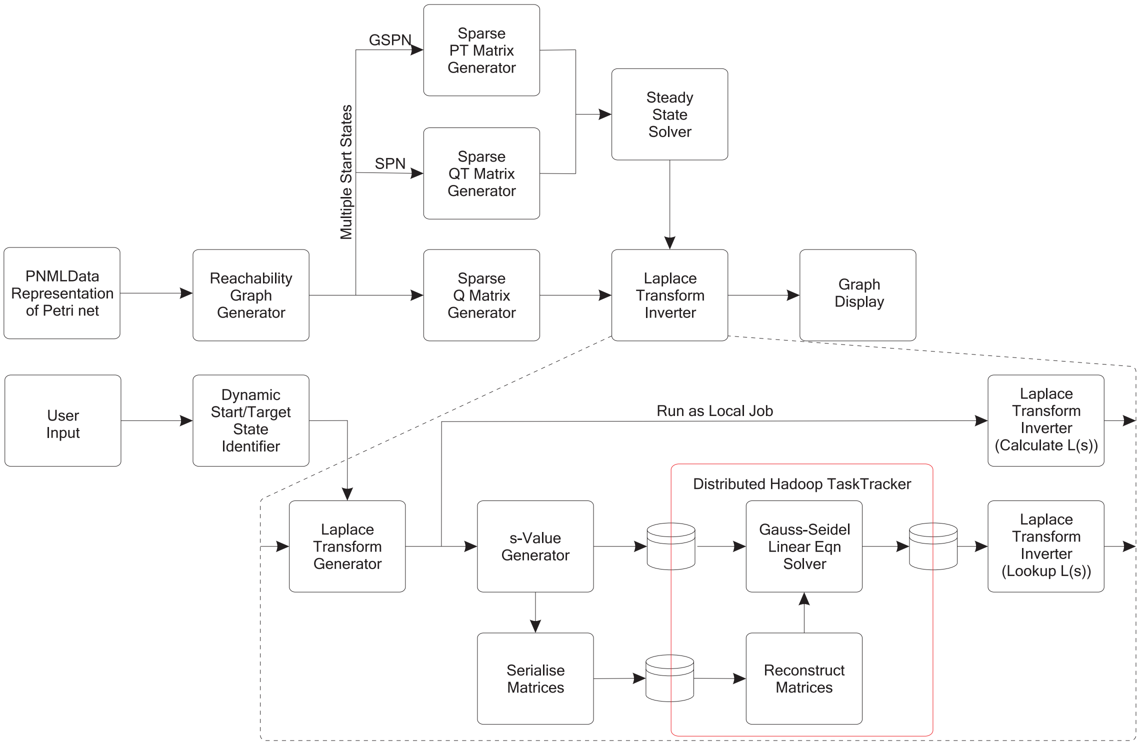
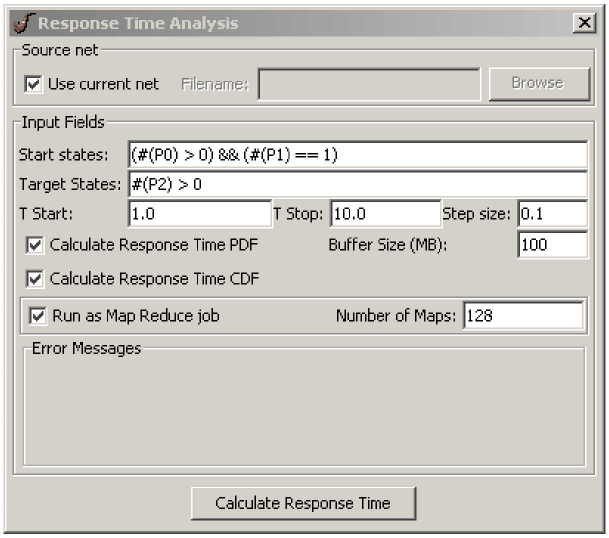

GSPN
SPN
Multiple Start States
Sparse PT Matrix Generator
Sparse QT Matrix Generator
Steady State Solver
PNMLData Representation of Petri net
Reachability Graph Generator
Sparse Q Matrix Generator
Laplace Transform Inverter
Graph Display
User Input
Dynamic Start/Target State Identifier
Run as Local Job
Laplace Transform Inverter (Calculate L(s))
Distributed Hadoop TaskTracker
Laplace Transform Generator
s-Value Generator
Gauss-Seidel Linear Eqn Solver
Laplace Transform Inverter (Lookup L(s))
Serialise Matrices
Reconstruct Matrices

Figure 2. Overview of Response Time Analysis module



Response Time Analysis

Source net

☑ Use current net Filename: Browse

Input Fields

Start states: (#(P0) > 0) && (#(P1) == 1)

Target States: #(P2) > 0

T Start: 1.0 T Stop: 10.0 Step size: 0.1

☑ Calculate Response Time PDF Buffer Size (MB): 100

☑ Calculate Response Time CDF

☑ Run as Map Reduce job Number of Maps: 128

Error Messages

Figure 1. User-facing input window of the PIPE2 Response Time Analysis module

error reporting. Further screens keep the user updated during the processing and graphically display the results. There is an option to cancel processing at any time.

Figure 2 shows a breakdown of the steps which the module takes in order to calculate response time densities for a GSPN model. The module can be seen to take the representation of the Petri net as a PIPE2 PNMLData object and use this to generate the various sparse matrices required for the calculation of the response time density. Next, the reachability graph (described as the generator matrix **Q** in the case of an SPN and as an EMC with probability transition matrix **P** in the case of a GSPN) is generated and the steady-state probability distribution vector is calculated (recall this is required to weight start states appropriately). The Laplace transform inverter can be run either locally or in distributed format using the Hadoop MapReduce platform. Distributing the LT inverter allows for large models to be analysed in a scalable manner in reasonable time.

The first step in the Laplace transform inverter is to generate the complex linear systems that must be solved to yield the Laplace transform of the convolution of all state sojourn times along all paths from the set of start markings to any of the set of target markings. These are calculated as described in Section 2.1 and are dependent on the target states recognised by the start/target state identifier. The number of linear systems to be solved depends on the number of time points specified by the user; these systems are then solved either locally or as a distributed MapReduce job on Hadoop. Finally, the results are displayed as a graph whose underlying data can be saved as a CSV file.

4.2 Reachability Graph Generator

The reachability graph genenerator used in the Response Time Analysis module is based on an existing one already implemented in PIPE2 by [11]. Its concept is to perform a breadth-first search of the states of the GSPN's underlying SMP. It starts with a single state and finds all the states that can be reached from it in a single transition. This process is then repeated for each of those successor states until all states have been explored. In order to detect cycles a record must be kept in memory of each state identified; this presents a significant problem when dealing with large state spaces. Storing an array representing the marking of each state's places would consume far too much memory. A better approach is to employ a probabilistic, dynamic hashing technique, as devised in [12]. Here, only a hash of the state's marking array is stored in one of many linked lists which are in turn stored in a hash table. By using a second hash function to determine which list to store each state in the risk of collisions is dramatically reduced. A full representation is also stored on disk as it is necessary when identifying start and target states. The MappedByteBuffer from the new I/O classes introduced in Java J2SE 5 were used to dramatically improve performance when writing to disk.

4.3 Dynamic Start/Target State Identifier

A passage time of interest can be specified by defining a set of start states and a set of target states. For example, a user might wish to calculate the passage time from any state where a buffer contains three items, to any state where it contains none. In Petri net modelling the buffer would correspond to a place while the items would be tokens. A convenient way for the user to be able to specify sets of start and target states is by giving conditions on the number of tokens on places. Finding the corresponding states is a non-trivial problem as the entire state space must be searched to identify such states. A very fast algorithm is required as state spaces can be huge. We accomplish this by allowing the user to enter a logical expression, whose terms compare the markings of places with constants or the markings of other places. This is then translated into a Java expression which is inserted into a template class file that is dynamically compiled and loaded at run-time to provide a method containing a simple logical expression which can check whether each state matches the user's conditions.

4.4 Matrix Generation

The sparse $\mathbf{Q}$ matrix generator takes the states and transitions stored in the output file of the reachability graph generator and constructs a square matrix describing the relationships between states. These matrices have few non-zero values and so a sparse matrix format was used to conserve memory usage based on that devised in [12] and shown in Figure 3. It can be seen that the two-dimensional array contains no actual values, rather column number and index values into another array where the actual values are stored. It is also necessary to record whether a state is tangible or vanishing, as this will influence how the $\mathbf{Q}$ matrix is transformed into the Laplace transform inversion of the passage time, as described in Equation 1. Storing the diagonal element at the end of each row helps in the efficiency of both generating the $\mathbf{Q}$ matrix and its conversion.

4.5 Steady-State Solver

The steady-state solver uses the Gauss-Seidel iterative method to find the steady-state distribution vector of a Markov chain represented by a $\mathbf{Q}$ (or $\mathbf{P}$) matrix by solving the equation $\pi\mathbf{Q} = 0$ (or $\pi\mathbf{P} = \pi$). To obtain standard linear system form $\mathbf{Ax} = \mathbf{b}$ requires the transpose of the $\mathbf{Q}$ or $\mathbf{P}$ matrix, which we generate with an appropriate transpose function. The sparse matrix format described previously allows for a very efficient Gauss-Seidel algorithm.

4.6 Linear Solution and Numerical Laplace Transform Inversion

The next step is to set up the linear system of Equation 1 of the form $\mathbf{AL} = \mathbf{b}$ with the aim of solving to find the response time vector, $\mathbf{L}$. The data necessary for this is extracted from the $\mathbf{Q}$ matrix and the set of target states. Recall that each element of the vector $\mathbf{L}_i = \mathcal{L}_{i\vec{j}}(s)$ represents the Laplace transform of the response time distribution between an initial state i and a set of target states $\vec{j}$ sampled at a point s for $1 \leq i \leq n$. If multiple start markings are identified, a vector α is calculated from the normalised steady-state probability vector and the quantity $\alpha \cdot \mathbf{L}$ found. This gives us the Laplace transform of the response time density from a set of initial states to a set of target states.

The solution process is driven by the time-range over which the user wishes to plot the probability density function of the response time. Each t-point of the final response time distribution requires 65 s-point function calls of the Laplace transform of the response time density[2]. Each s-point sample of the Laplace transform is given by a single solution of Equation 1. The precise set of s-values required are calculated from an Euler Laplace inversion algorithm derived from

[2] The number of s-points required is implementation dependent and varies according to the configuration of the Laplace Transform inversion algorithm employed

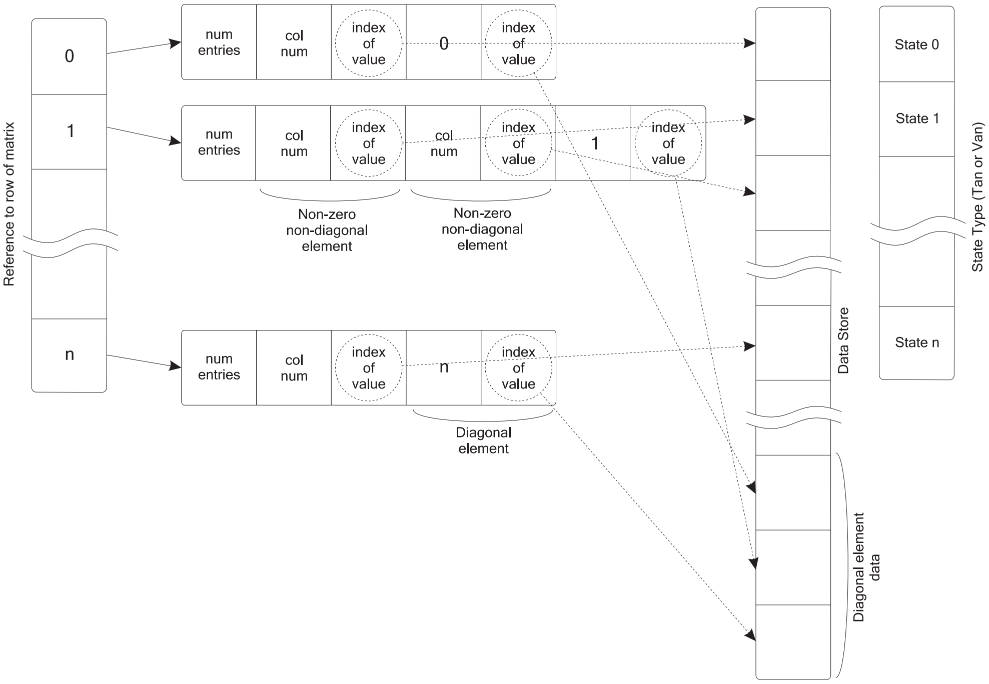


Figure 3. Sparse Matrix format module

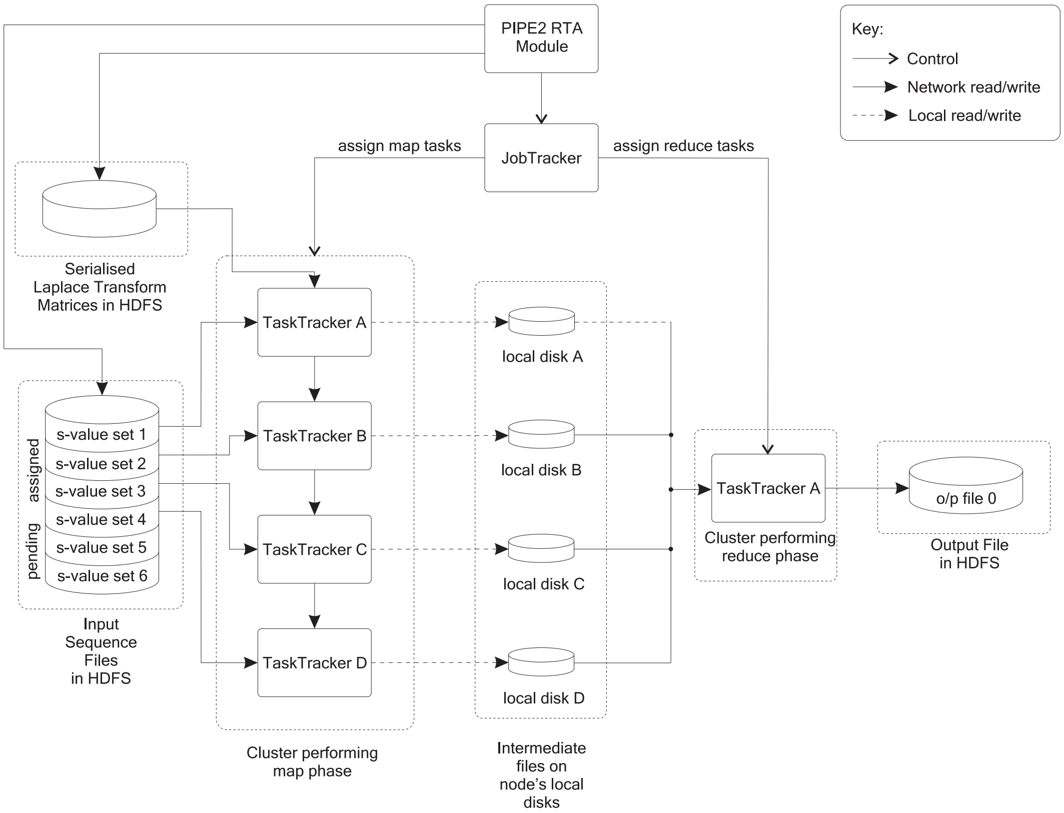


Figure 4. An overview of the MapReduce distributed linear equation solver used in the RTA module

Equation 3 as a function of the desired time range of the final plot. Thus a time range of 100 points may require as many as 6 500 distinct solutions of Equation 1, provided by a standard Gauss-Seidel iterative method.

For models with large state spaces solving the sets of linear equations is too processor intensive to do locally. We therefore integrate the module with the Hadoop MapReduce framework. An overview of this process is shown in Figure 4.

In order to store $\mathcal{L}(s)$, we set up a Hashmap indexed on the s-value of the Laplace transform. This has the advantage that any repeated s-values need only be calculated once.

The list of s-values is copied to a number of sequence files, a special file format containing key/value pairs which is used by Hadoop as an input to a MapReduce job. We set the s-values as the keys while at this stage the values are just placeholders for the results. Each sequence file corresponds to a Map task and the s-values are split evenly between them. It was necessary to do this explicitly as Hadoop's automatic file splitting functionality is aimed at much larger data files. Keys and values in a sequence file are required by Hadoop to be wrapped in a class which implements a custom comparable interface. While Hadoop has built-in support for certain Java primitives, it was necessary to create wrappers for Doubles and the open-source complex number library we used.

The **A**-matrix and **b** vector, as well as details of start states and their alpha weights are serialised and the resulting binary file is copied into the cluster's HDFS. Each TaskTracker must have access to these in order to solve the system of linear equations. At this point the MapReduce job can be started and the directory containing the sequence files is given to Hadoop as the input source for the job. Hadoop assigns each slave one or more Map tasks and sends it the associated sequence file. When a node receives a Map task it will run the Map function a number of times; once for each s-value in its associated sequence file. For the first Map function run on a node, the **A**-matrices are copied out of the HDFS to local storage and deserialised. Subsequent calls to the Map function (even as part of a different Map task) then use this local copy, thereby greatly reducing network traffic. Each Map function solves the set of complex linear equations for its s value using a complex version of Gauss-Siedel iterative algorithm similar to that used in the steady-state solver. It outputs a key/value pair whose key is s and value is an object which contains both the $\mathcal{L}(s)$ value and $\mathcal{L}(s)/s$. If multiple initial states have been specified the $\mathcal{L}(s)$ values are weighted appropriately. Calculating $\mathcal{L}(s)/s$ value now and later inverting means we can easily retrieve the CDF of the passage time, for very little extra computation.

Whilst the $\mathcal{L}(s)$ values are being calculated, a single Reduce task is started. We use the Reduce task simply to collect all the $\mathcal{L}(s)$ values from across the cluster where they have been stored locally by each Map function and copy them to a single output sequence file. There is no additional processing required during the Reduce phase. With the distributed job complete, the response time calculator copies the results into a HashMap indexed on s-values for fast access and runs the Euler algorithm to perform the Laplace Transorm inversion. This is run twice, once for each set of results to give the Response Time Distribution and the Cumulative Density Function.

5 NUMERICAL RESULTS

All results presented in this section were produced by PIPE2 running in conjunction with the latest development version of Hadoop (0.13.1) on a cluster of 15 Sun Fire x4100 machines, each with two dual-core, 64-bit Opteron 275 processors and 8GB of RAM. The operating system is a 64-bit version of Mandrake Linux and nodes are connected by gigabit ethernet and an Infiniband interface managed by a Silverstorm 9024 switch with a throughput of 2.5Gbit/s. One of the nodes was designated the master machine and ran the Hadoop Namenode and JobTracker processes, as well as PIPE2.

5.1 Validation

Our validation process began with the Branching Erlang model, taken from [13] and shown in Figure 5, which consists of two branches with known response times. In particular, the upper branch has an $Erlang(12,2)$ distribution, while the lower has an $Erlang(3,1)$ distribution. There is an equal probability of either branch being taken, as the weights of the immediate transitions are identical. As Erlang distributions are trivial to calculate analytically we can therefore compare the results form our numerical Laplace transform inversion method with their true values.

Figures 7 and 8 compare the results produced by PIPE2 and those calculated analytically for the cycle time density and its corresponding CDF function of the Branching Erlang model. Excellent agreement can be seen between the two. These results demonstrate the Response Time Analysis module's ability to handle cases where the set of source and target states overlap (i.e. to calculate cycle times), as well as bimodal density curves.

To validate the module for larger models with multiple start and target states we used the Courier Protocol model, first presented in [16] and shown in Figure 6. It models the ISO Application, Session and Transport layers of the Courier sliding-window communication protocol. $p1$ to $p26$ represent the sender while $p27$ to $p46$ represent the receiver. Data flows from sender to receiver over a network which is modelled by the two paths from $p13$ to $p35$. This split path models the sender's transport layer fragmenting outgoing data packets. All packets traverse the network via the path that begins with $t8$, except for the final packet which travels over the $t9$ path. When a packet is received, and acknowledgement is setn back to the sender which arrives on $p20$. No received data is sent to higher levels of the protocol until the final fragment is received. At this point a data token is passed up via $p27$. The ration of the weights of transitions $t8$ and $t9$ control the number of fragments produced per message. This ratio is known as the fragmentation ratio and for our model is set to 1. By increasing the number of tokens on $p14$, the sliding-window size, we can dramatically increase the state space of the model.

We begin our validation with the sliding-window size set to one, which results in a state space of 29 010. The module completed this exploration in less than 8 seconds on a single machine. Results for the passage time from the set of markings where $M(p11) > 0$ to those where $M(p20) > 0$ are shown in Figure 9, where 7 320 source markings and 1 860 target markings were identified. They closely match simulation results for this same model that were produced in [8] (see Figure 10. It should be noted that a direct and general comparison of the time complexity of the numerical and simulation approaches is difficult: in the former case the complexity depends on the rank and stiffness of the equations solved; in the latter it depends on the rate at which passages from source to target markings are observed while walking at random through the state space. It should also be noted that our model uses a scaled set of rates that are equal to the original benchmarked rates divided by 5 000. This is necessary as the range in magnitude of the original rates causes problems with the numerical methods used to invert the Laplace transform. The results presented here are the raw results from the PIPE2 module and so must be re-scaled to give the correct timings.

In order to ascertain how the Response Time Analysis module performs with models with larger state spaces we again used the Courier Protocol model, increasing its window size to 3. This results in a state space of 2 162 610 states (including vanishing states) with 5 469 150 transitions between them. Again, analysing from markings where $M(p11) > 0$ to markings where $M(p20) > 0$, we find there are 439 320 start markings and 273 260 target markings. Results were produced for 50 t-points ranging from 1 to 99 in increments of 2, resulting in a work queue of over 1 800 systems of linear equations, each of rank 2.2 million. State space exploration took 20 minutes, while the Laplace transform inversion took 8 hours 9 minutes when run on all 15 nodes (3 Map tasks per node). Generation times for the various other matrices totalled less than 20 seconds.

5.2 Processing Times

Table 1 shows the time taken to perform the Laplace transform inversion for the Courier Protocol model (window size 1) for 50 t-points on various cluster sizes. It should be noted that while timings shown are for single runs, when multiple runs were performed times were consistent. The cluster size column refers to the number of computer nodes assigned to the Hadoop cluster. The second column indicates the number of Map tasks assigned to each node. Hadoop allows multiple Map tasks to be run concurrently on a single node which is of particular benefit with multicore machines as it allows full use to be made of all cores. Where only one Map task was assigned to a node only one core was in use. This was scaled up to 8 and 15 machine clusters until all cores were in use at once. The third column shows the total number of cores being used simultaneously. The optimum map granularity for each cluster size was found through experimentation and is listed in the fourth column.

Cluster Size	No. Maps Per Node	Total Cores	Total Maps	Time (seconds)
1	1	1	10	3112.167
2	1	2	20	1596.322
4	1	4	40	809.653
8	1	8	80	433.173
8	2	16	80	256.694
8	4	32	80	192.982
15	1	15	80	252.515
15	2	30	80	165.561
15	4	60	100	131.754

Table 1. Laplace transform inversion times for the Courier Protocol (window size 1) on various cluster sizes

It is clear from Table 1 that the distributed response time calculator offers excellent scalability. With small clusters there is an approximate halving of calculation time as the cluster size is doubled. As the cluster sizes (and hence the number of Map tasks) grow this improvement drops slightly to a factor of approximately 1.8. This is to be expected as there is some overhead in setting up Map tasks.

When the number of cores used on each node is increased we again see a good reduction in processing times. However, we no longer see the calculation time halve as the available cores double. It is likely that this is due to contention for shared resources within each node, such as the system bus and memory. Further weight can be added to this argument by comparing the results for jobs run on 8 nodes with jobs run on 15 nodes. A job run on 32 cores spread over 8 nodes takes over 27 seconds longer than a job run on only 30 cores, but spread over 15 nodes.

The number of Map tasks for a particular Hadoop job can have a dramatic effect on the time taken to complete the job. While having one Map task per core in the cluster results in the least overhead it can actually result in poor performance. The main reason for this is that Map tasks take different amounts of time to complete, even when each one contains the same number of $\mathcal{L}(s)$ values to calculate. When running jobs it is not uncommon to see the slowest jobs take over three

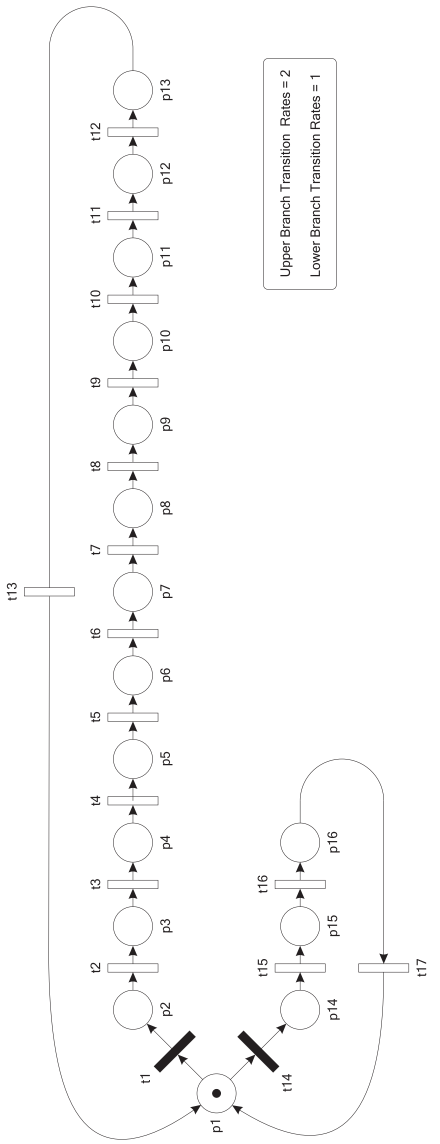


Figure 5. The Branching Erlang model

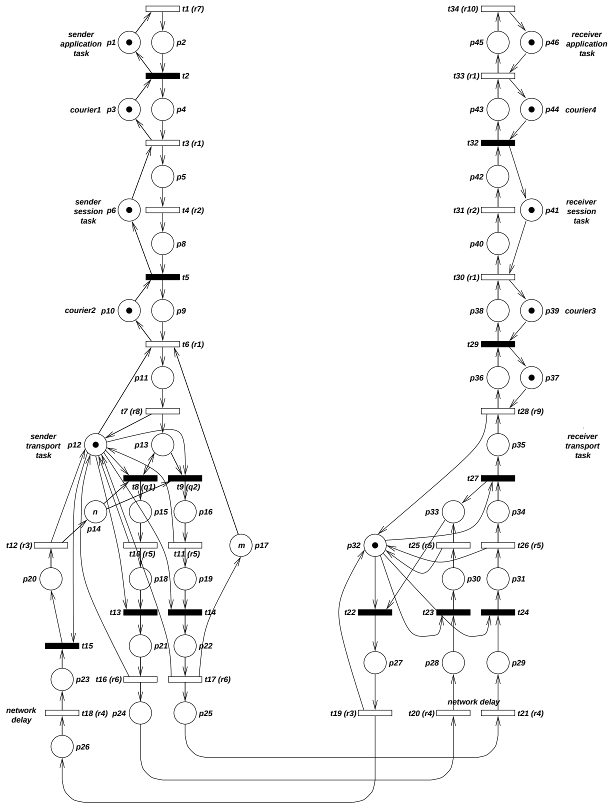


Figure 6. The Courier Protocol model

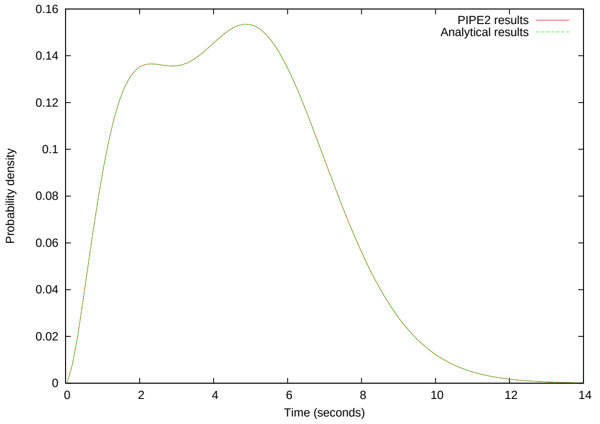


Figure 7. Cycle time distribution from markings where $M(p1) > 0$ to markings where $M(p1) > 0$ in the Branching Erlang model

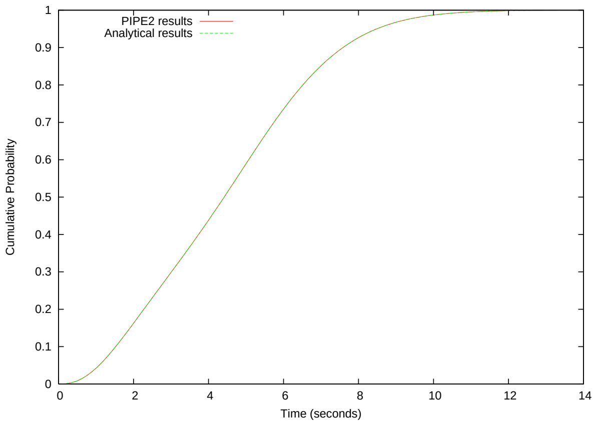


Figure 8. CDF of cycle time from markings where $M(p1) > 0$ to markings where $M(p1) > 0$ in the Branching Erlang model

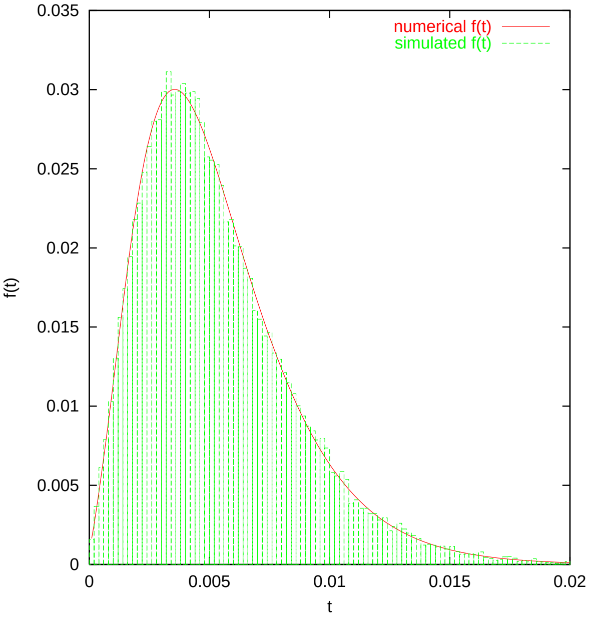


Figure 10. (Re-scaled) Numerical and simulated passage time density from markings where $M(p11) > 0$ to markings where $M(p20) > 0$ in the Courier Protocol model (window size 1)

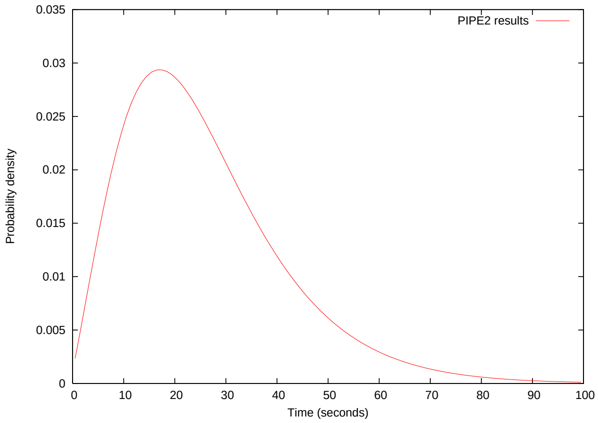


Figure 9. (Unscaled) Passage time density from markings where $M(p11) > 0$ to markings where $M(p20) > 0$ in the Courier Protocol model (window size 1)

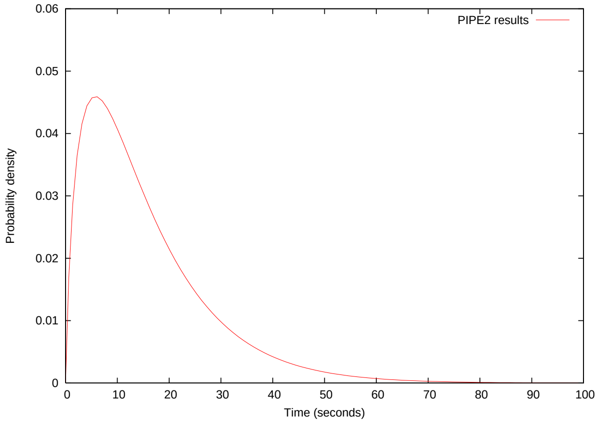


Figure 11. (Unscaled) Passage time density from markings where $M(p11) > 0$ to markings where $M(p20) > 0$ in the Courier Protocol model (window size 3)

times as long to complete as the faster ones. It is thought that this is largely due to certain $\mathcal{L}(s)$ values converging faster than others. Reducing the granularity, or increasing the number of Map tasks, reduces the length of time each Map task takes, and so reduces the time spent where most of the cluster is idle waiting for the last few Map tasks to complete.

Table 2 shows the time taken to perform the Laplace transform inversion for 200 t-points on the Courier Protocol model on a cluster of eight nodes, each running 4 Map Tasks with different numbers of Map Tasks specified. We can see that the optimum granularity for this job is for 256 Map tasks. At this granularity the maximum time to complete a Map task is approximately 150 seconds. This is the maximum time the job will spend waiting for a single Map task to complete when all others have finished. While this time is lower for the 384 Map task job, the benefit is outweighed by the additional overhead of scheduling and configuring an extra 128 Map tasks.

Granularity becomes even more important on heterogenous clusters. The undesirable situation where much of the cluster is idle while the last few Map tasks are executed can be exacerbated by the scheduler picking slower machines to run these tasks.

6 CONCLUSIONS

We have described the implementation of a Response Time Analysis module for an open-source Petri net editor and analyser, PIPE2. This module integrates with Hadoop, an open-source Java implementation of the MapReduce distributed programming environment to allow the response time analysis of large models using a cluster of commodity computers. Hadoop was originally developed for web indexing purposes, that is to perform relatively simple operations on huge data sets. We have shown that it can be successfully applied to a radically different type of problem, that of performing complex, computationally intensive calculations on much smaller data sets. We have seen that the MapReduce framework provided by Hadoop has been well suited to our problem of using the Euler algorithm for Laplace transform inversion. By identifying that the solution of many systems of complex linear equations, the computationally intensive part of the algorithm, are independent of one another we saw that this part of the algorithm could be distributed and would fit perfectly within the MapReduce paradigm. Using a popular open-source project to handle the distribution of processing allowed us to focus our development time on writing fast and efficient algorithms with the resulting product retaining excellent reliability with good fault tolerance for failing nodes and efficient scheduling of tasks among the cluster. There were some difficulties we had to overcome related to the architecture of Hadoop including the assumption that input files will contain large amounts of data and that there was no built-in support for high-precision floating point or complex data sets. There were also some unforeseen benefits, such as allowing us to take advantage of the automatic replication built into Hadoop's distributed file system to send the serialised matrices to each node in the cluster. Overall the framework provided excellent support for our solution and met most of our requirements.

We have also demonstrated techniques for conserving memory usage and improving performance which allow Java to become a viable language for this application, despite lacking explicit memory management facilities. Our solution for storing sparse matrices efficiently was key to this, by minimising memory required while simultaneously allowing for an optimised Gauss-Seidel algorithm. Utilising a dynamic, probabilistic hash based technique within our state space exploration algorithm was also essential. We also utilised some of the latest improvements in the Java language to increase performance. Models of up to at least 2.2 million states were shown to be easily accommodated using in-core processing. Re-implementing the linear equation solving algorithms as disk-based, rather than in-core, would allow for much larger model sizes.

Results produced by the Response Time Analysis module were validated for smaller models with analytically calculated results and for larger models with simulations. Excellent scalability was shown, with an almost linear improvement in calculation times with increased cluster sizes. Experimentation was performed to identify optimum granularity of Map tasks for certain model sizes.

REFERENCES

[1] J. Abate and W. Whitt. The Fourier-series method for inverting transforms of probability distributions. *Queueing Systems*, 10(1):5–88, 1992.

[2] F. Bause and P.S. Kritzinger. *Stochastic Petri Nets: An Introduction to the Theory*. Vieweg Verlag, Wiesbaden, Germany, 2nd edition, 2002.

[3] P. Bonet, C.M. Llado, R. Puijaner, and W.J. Knottenbelt. PIPE v2.5: A Petri net tool for performance modelling. In *Proc. 23rd Latin American Conference on Informatics (CLEI 2007)*, San Jose, Costa Rica, October 2007.

[4] J.T. Bradley, N.J. Dingle, W.J. Knottenbelt, and H.J. Wilson. Hypergraph-based parallel computation of passage time desnsities in large semi-Markov models. In *Linear Algebra and its Applications: Volume 386*, pages 311–334. Elsevier, July 2004.

[5] J.T. Bradley and W.J. Knottenbelt. The ipc/HYDRA tool chain for the analysis of PEPA models. In *Proc. 1st International Conference on the Quantitative Evaluation of Systems (QEST 2004)*, pages 334–335, September 2004.

No. Map Tasks	Calculation Time (s)	Fastest Map (s)	Slowest Map (s)
56	583.061	267	551
128	525.106	93	282
256	497.495	52	156
384	516.948	40	107

Table 2. Laplace transform inversion times for the Courier Protocol (window size 1) for various granularities

[6] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proceedings of the OSDI'04: Sixth Symposium on Operating System Design and Implementation*, San Francisco, California, U.S.A., December 2004.

[7] 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, October 2004.

[8] N.J. Dingle, P.G. Harrison, and W.J. Knottenbelt. Response time densities in Generalised Stochastic Petri net models. In *Proceedings of the 3rd ACM Workshop on Software and Performance (WOSP 2002)*, pages 46–54, Rome, Italy, 2002.

[9] N.J. Dingle, P.G. Harrison, and W.J. Knottenbelt. Uniformisation and hypergraph partitioning for the distributed computation of response time densities in very large Markov models. *Journal of Parallel and Distributed Computing*, 64(8):908–920, August 2004.

[10] J. Ekanayake, S. Pallickara, and G. Fox. Mapreduce for data intensive scientific analyses. In *Proc. 4th IEEE International Conference on eScience*, pages 277–284, 2008.

[11] T. Kimber, B. Kirby, T. Master, and M. Worthington. Petri nets group project final report. Technical report, Imperial College, London, United Kingdom, March 2007.

[12] W.J. Knottenbelt. Generalised Markovian analysis of timed transition systems. Master's thesis, University of Cape Town, Cape Town, South Africa, July 1996.

[13] W.J. Knottenbelt and P.G. Harrison. Passage time distributions in large Markov chains. In *Proceedings of ACM SIGMETRICS*, pages 77–85, Marina Del Rey, California, U.S.A., June 2002.

[14] B. Melamed and M. Yadin. Randomization procedures in the computation of cumulative-time distributions over discrete state Markov processes. *Operations Research*, 32(4):926–944, July–August 1984.

[15] J.K. Muppala and K.S. Trivedi. Numerical transient analysis of finite Markovian queueing systems. *Queueing and Related Models, Bhat, U.N.; Basawa, I.V. (eds.)*, pages 262–284, 1992.

[16] C.M. Woodside and Y. Li. Performance Petri net analysis of communication protocol software by delay-equivalent aggregation. In *Proceeding of the 4th International Workshop on Petri nets and Performance Models (PNPM'91)*, pages 64–73, Melbourne, Australia, December 1991. IEEE Computer Society Press.

7 AUTHOR BIOGRAPHIES

Oliver Haggarty obtained a BMus (Hons) from the University of Surrey in 2001. After working in the audio electronics industry he returned to university to study Computing Science at Imperial College London in 2007, completing his MSc with distinction and receiving the Trayport prize for academic excellence. Since then he has been working as a Software Developer at the Royal Bank of Scotland.

William Knottenbelt completed his BSc (Hons) and MSc degrees in Computer Science at the University of Cape Town in South Africa before moving to London in 1996. He obtained his PhD in Computing from Imperial College London in February 2000, and was subsequently appointed as a Lecturer in the Department of Computing in October 2000. Now a Senior Lecturer, his research interests include parallel computing and stochastic performance modelling.

Jeremy Bradley is a Senior Lecturer in the Department of Computing at Imperial College London. His research involves modelling systems with high-level formalisms such as stochastic Petri nets and stochastic process algebras, as well as novel algorithms for the performance analysis of semi-Markov processes. He has designed the only semi-Markov stochastic process algebra and has written and maintains the Imperial PEPA compiler.