An Empirical Study of the Scalability of Performance Analysis Tools in the Cloud

Nicholas J. Dingle
Department of Computing, Imperial College London,
180 Queens Gate, London SW7 2AZ.
Email: njd200@doc.ic.ac.uk

***Abstract*—Calculation of performance metrics such as steady-state probabilities and response time distributions in large Markov and semi-Markov models can be accomplished using parallel implementations of well-known numerical techniques. In the past these implementations have usually been run on dedicated computational clusters and networks of workstations, but the recent rise of cloud computing offers an alternative environment for executing such applications. It is important, however, to understand what effect moving to a cloud-based infrastructure will have on the performance of the analysis tools themselves. In this paper we investigate the scalability of two existing parallel performance analysis tools (one based on Laplace transform inversion and the other on uniformisation) on Amazon's Elastic Compute Cloud, and compare this with their performance on traditional dedicated hardware. This provides insight into whether such tools can be used effectively in a cloud environment, and suggests factors which must be borne in mind when designing next-generation performance tools specifically for the cloud.**

***Keywords*-Cloud computing; Performance analysis; Parallel computing; Empirical study;**

I. Introduction

One common approach for the analysis of Markov and semi-Markov chains with many millions of states is to use the combined compute power and memory capacity of a number of computers in parallel; for example, see [1], [2], [3], [4]. Our own prior work has similarly exploited parallel techniques to compute response time densities and distributions in very large Markov and Semi-Markov chains [5], [6], [7], [8], [9].

To exploit the power of these implementations, the user is typically required to possess a dedicated computational cluster or network of workstations. Such hardware is, however, expensive to buy and to run, requires sufficient space with associated power and cooling to house it, and staff to maintain it. With the coming pressure on academic research budgets in the UK, it is conceivable that individual research groups will struggle to continue to acquire such resources for themselves. Cloud computing holds the promise of dramatically reducing these overheads. Instead of purchasing and maintain one's own machines, the user instead pays for time on machines owned by a third party.

The idea of centralised computing services is not new, and indeed UK academic users have access to large-scale parallel computing resources either within their own institutions (e.g. Imperial College's High Performance Computing Service [10]) or nationally via the EPSRC-managed HECToR machines [11]. Such resources tend to be geared towards high-end scientific computing users requiring hundreds of processors for many hours at a time, however, while performance analysts usually require much more modest (although still expensive) resources for far less time. Something like Amazon's Elastic Compute Cloud (EC2) service is therefore far better suited to our requirements: computing power is available to all users, regardless of how many or how few processors are required, for as long as they need them at a cost which is far cheaper than that of equivalent in-house resources.

A key concern in using existing performance analysis tools in the cloud is how well those tools themselves perform in this environment. The performance of parallel programs is often expressed in terms of their scalability; that is, to what extent using extra processors reduces the overall runtime. When producing scalability results for publication, it is usual to run the tools on idle machines with little load on the network to minimise the effect of external factors on their performance.

Such an ideal situation cannot be expected in a cloud computing environment, however. By its very nature, our tools will have to share physical hardware (both processors and network connections) with many other executing applications. How our tools perform in this environment will therefore differ greatly from their performance in ideal conditions. It is important to understand how their performance scales, however, as cloud computing offers the ability to make use of large numbers of processors far more cheaply than we could ourselves own. If our tools cannot efficiently use these extra resources then they will either need to be modified, or replaced with tools that can.

In this paper we study the scalability of two of our previously-presented performance analysis tools: a Laplace transform-based response time analyser [6] and the HYpergraph-based Distributed Response-time Analyser (HYDRA) [7], [12]. We compare their scalability in a cloud computing environment (Amazon EC2) with that observed on a variety of parallel environments (including parallel computers, dedicated clusters and networks of workstations) in the context of a case study analysis of the Flexible Manufacturing System (FMS) Generalised Stochastic Petri Net (GSPN) model.

The remainder of this paper is organised as follows. Section II briefly describes the theoretical underpinnings and parallel implementations of the performance analysis tools chosen for this study, before Section III describes

the Amazon EC2 environment and the Amazon Machine Image with which this study will be conducted. Section IV then presents the observed scalability results for the chosen tools across a range of parallel environments and discusses the implications. Finally, Section V concludes and considers future work.

II. Performance Analysis Tools

We will study the scalability of two previously-presented performance analysis tools: a Laplace transform inverter [6] and the HYpergraph-based Distributed Response-time Analyser (HYDRA) [7], [12]. Although the core computation carried out by both tools is repeated sparse matrix-vector multiplication, the way in which they parallelise the problem is different and consequently they place very different communication loads on the network. For each tool, we briefly sketch the underlying theory and then describe its parallel operation.

A. Laplace Transform Inverter

In a GSPN, the first passage time from a single source marking i into a non-empty set of target markings $\vec{j}$ is:

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

where $M(t)$ is the marking of the GSPN at time t and $N(t)$ denotes the number of state transitions which have occurred by time t. $P_{i\vec{j}}$ corresponds to the first time that the system enters a state in the set of target states $\vec{j}$, given that the system began in the source state i and at least one state transition has occurred. $P_{i\vec{j}}$ is a random variable with probability density function $f_{i\vec{j}}(t)$ such that:

$$\mathbb{P}(a < P_{i\vec{j}} < b) = \int_a^b f_{i\vec{j}}(t)\ dt \quad (0 \leq a < b)$$

In order to determine $f_{i\vec{j}}(t)$ it is necessary to convolve the state holding-time density functions over all possible paths from state i to all of the states in $\vec{j}$.

The calculation of the convolution of two functions in t-space can be more easily accomplished by multiplying their Laplace transforms together in s-space and inverting the result. The calculation of $f_{i\vec{j}}(t)$ is therefore achieved by calculating the Laplace transform of the convolution of the state holding times over all paths between i and $\vec{j}$ and then numerically inverting this Laplace transform.

We proceed by means of a first-step analysis. That is, to calculate the first passage time from state i into the set of target states $\vec{j}$, we consider moving from state i to its set of direct successor states $\vec{k}$ and thence from states in $\vec{k}$ to states in $\vec{j}$. The Laplace transform of the (exponential) sojourn time density function of tangible marking i is $\mu_i/(s+\mu_i)$, but for a vanishing marking the sojourn time is 0 with probability 1, giving a corresponding Laplace transform of 1 for all values of s. We therefore distinguish between passage times which start in a tangible state and those which begin in a vanishing state:

$$L_{i\vec{j}}(s) = \begin{cases} \sum_{k \notin \vec{j}} \left(\frac{q_{ik}}{s-q_{ii}}\right) L_{k\vec{j}}(s) + \sum_{k \in \vec{j}} \left(\frac{q_{ik}}{s-q_{ii}}\right) & \text{if } i \in \mathcal{T} \\ \sum_{k \notin \vec{j}} p_{ik} L_{k\vec{j}}(s) + \sum_{k \in \vec{j}} p_{ik} & \text{if } i \in \mathcal{V} \end{cases}$$

This system of linear equations can be expressed in matrix–vector form. For example, when $\vec{j} = \{1\}$, $\mathcal{V} = \{2\}$ and $\mathcal{T} = \{1, 3, \ldots, n\}$ we have:

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

Given a particular (complex-valued) s, these equations can be solved for $L_{i\vec{j}}(s)$ by standard iterative numerical techniques for the solution of systems of linear equations in $\mathbf{Ax} = \mathbf{b}$ form. $L_{i\vec{j}}(s)$ can then be inverted to yield $f_{i\vec{j}}(t)$ using numerical techniques such as the Euler [13] or Laguerre [14] methods.

We have implemented a response time analysis pipeline for GSPN models, as shown in Fig. 1. Models are specified in an enhanced form of the DNAmaca Markov Chain Analyser interface language [15]. From the high-level model, a state space generator produces the reachability graph, along with a list of the source and destination markings that match their respective high-level descriptions. A steady-state solver then computes appropriate weights for the source markings.

Control is then passed to the distributed Laplace transform inverter, which employs the master-slave architecture shown in Fig. 1. It is the scalability of this portion of the tool that we will investigate later in this paper. The inverter is written in C++ and uses the Message Passing Interface (MPI) [16] standard, so it is portable to a wide variety of parallel computers and workstation clusters.

Initially, the master node determines the distinct values of the complex variable s at which the Laplace transform of the response time distribution, $L(s)$, will need to be evaluated. Those values of s for which there is no value of $L(s)$ already stored in the disk cache are added to a global work queue.

At start-up, slave processors read into memory the reachability graph as well as a list of the source and destination markings and the weights to apply to the distribution for each source marking. Each slave processor then applies for an s-value from the global work queue. The slave calculates the corresponding value of $L(s)$ by solving the set of sparse linear equations using an appropriate iterative numerical method such as Successive Over-Relaxation (SOR) or Conjugate Gradient Squared (CGS).

Slave processors return computed values of $L(s)$ to the master. The master stores the returned value in memory and disk caches and immediately issues more work to the slaves if any is available. When all values of $L(s)$ have been computed, the master runs through the Laplace

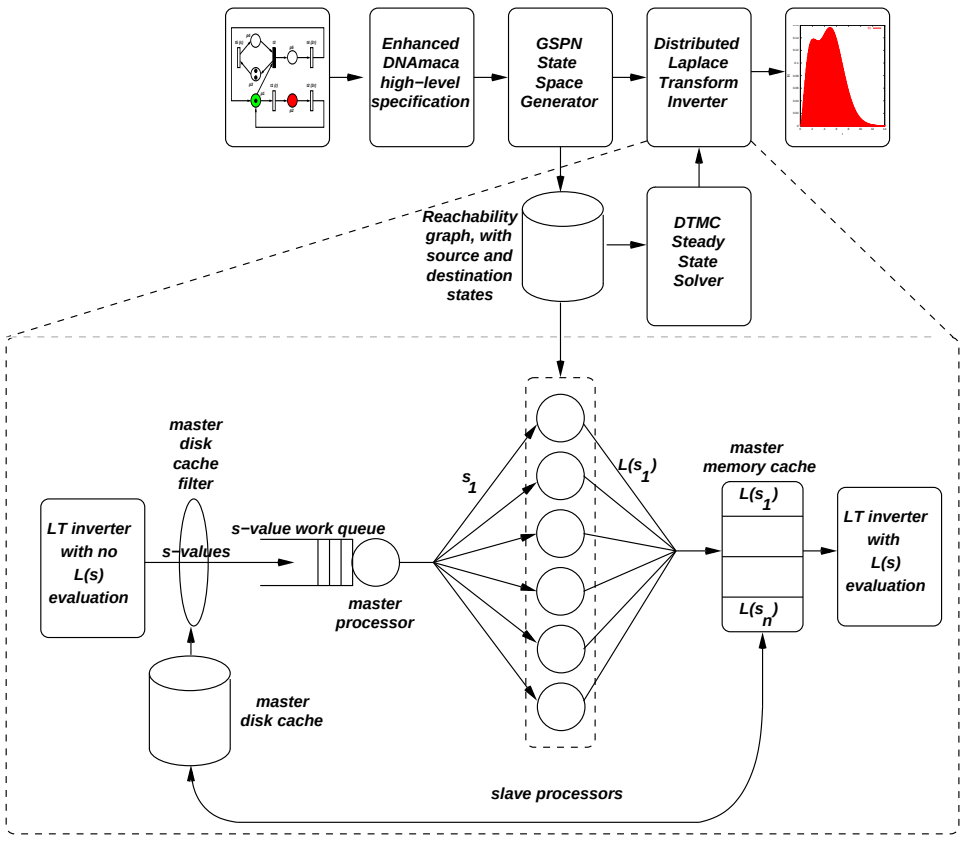


Figure 1. Response time analysis tool showing operation of the distributed Laplace transform inverter in detail [6].

transform inversion algorithm again, this time performing all calculations and obtaining any values of $L(s)$ needed from the memory cache. The resulting points on the response time density curve are written to a disk file, and displayed using the GNUplot graph plotting utility.

The single global work queue with multiple servers ensures a good load balance and very high utilisation of slave processors. In addition, there is no inter-slave communication and the amount of master-slave communication is low. We therefore expect this tool to exhibit good scalability.

B. HYDRA

Response time densities and quantiles in Continuous Time Markov Chains (CTMCs) can also be computed through the use of uniformisation (also known as randomization) [17], [18], [19], [20]. This transforms a CTMC into one in which all states have the same mean holding time, $1/q$, by allowing "invisible" transitions from a state to itself. This is equivalent to a discrete-time Markov chain, after normalisation of the rows, together with an associated Poisson process of rate q. The one-step transition probability matrix $\mathbf{P}$ which characterises the one-step behaviour of the uniformised DTMC is derived from the generator matrix $\mathbf{Q}$ of the CTMC as:

$$\mathbf{P} = \mathbf{Q}/q + \mathbf{I} \tag{1}$$

where the rate $q > \max_i |q_{ii}|$ ensures that the DTMC is aperiodic.

The calculation of the first passage time density between two states has two main components. The first considers the time to complete n hops ($n = 1, 2, 3, \ldots$). Recall that in the uniformised chain all transitions occur with rate q. This means that the convolution of n of these holding-time densities is the convolution of n exponentials all with rate q, which is an n-stage Erlang density with rate q.

Secondly, it is necessary to calculate the probability that the transition between a source and target state occurs in exactly n hops of the uniformised chain, for every value of n between 1 and a maximum value m. This is calculated by repeated sparse matrix-vector multiplications. The value of m is determined when the value of the nth Erlang density function drops below a threshold value. After this point, further terms are deemed to add nothing significant to the passage time density and are disregarded.

The density of the time to pass between a source state i and a target state j in a uniformised Markov chain can therefore be expressed as the sum of m n-stage Erlang densities, weighted with the probability that the chain moves from state i to state j in exactly n hops ($1 \leq n \leq m$). The response time between the non-empty set of source states $\vec{i}$ and the non-empty set of target states $\vec{j}$ therefore has probability density function:

$$f_{\vec{ij}}(t) = \sum_{n=1}^{\infty} \left(\frac{q^n t^{n-1} e^{-qt}}{(n-1)!} \sum_{k \in \vec{j}} \pi_k^{(n)} \right)$$
$$\simeq \sum_{n=1}^{m} \left(\frac{q^n t^{n-1} e^{-qt}}{(n-1)!} \sum_{k \in \vec{j}} \pi_k^{(n)} \right) \quad (2)$$

where

$$\boldsymbol{\pi}^{(n+1)} = \boldsymbol{\pi}^{(n)} \mathbf{P} \qquad \text{for } n \geq 0 \quad (3)$$

The key opportunity for parallelism in HYDRA is in the repeated sparse matrix-vector multiplications of Eq. 3. To perform these operations efficiently in parallel it is necessary to map the non-zero elements of $\mathbf{P}$ onto processors such that the computational load is balanced and communication between processors is minimised. To achieve this, we use hypergraph partitioning to assign matrix rows and corresponding vector elements to processors [21].

Fig. 2 shows the architecture of the HYDRA tool. As with the Laplace transform-based tool, the process of calculating a response time density begins with a high-level model specified in an enhanced form of the DNAmaca interface language. Next, a probabilistic, hash-based state generator uses the high-level model description to produce the generator matrix $\mathbf{Q}$ of the model's underlying Markov chain as well as a list of the initial and target states. $\mathbf{P}$ is constructed from $\mathbf{Q}$ according to Eq. 1 and partitioned using a hypergraph partitioning tool.

The pipeline is completed by our distributed response time density calculator, which is again implemented in C++ and uses MPI. Initially each processor tabulates the Erlang terms for each t-point required (cf. Eq. 2). Computation of these terms ends when they fall below a specified threshold value. The terminating condition also determines the maximum number of hops m used to calculate the right-hand factor, a sum which is independent of t.

Each processor reads in the rows of the matrix and the corresponding elements of the vector $\boldsymbol{\pi}^{(0)}$ that correspond to its allocated partition. Each processor then determines which vector elements need to be received from and sent to every other processor on each iteration,

The vector $\boldsymbol{\pi}^{(n)}$ is then calculated for $n = 1, 2, 3, \ldots, m$ by repeated sparse matrix-vector multiplications of the form of Eq. 3. For each matrix-vector multiplication, each processor begins by using non-blocking communication primitives to send and receive remote vector elements, while calculating the product of local matrix elements with locally stored vector elements. The use of non-blocking operations allows computation and communication to proceed concurrently on parallel machines where dedicated network hardware supports this effectively. The processor then waits for the completion of non-blocking operations (if they have not already completed) before multiplying received remote vector elements with the relevant matrix elements and adding their contributions to the local matrix-vector product cumulatively.

From the resulting local matrix-vector products each processor calculates and stores its contribution to the sum $\sum_{k \in \vec{j}} \pi_k^{(n)}$. After m iterations have completed, these sums are accumulated onto an arbitrary master processor where they are multiplied with the tabulated Erlang terms for each t-point required for the passage time density. The resulting points are written to a disk file and are displayed using the GNUplot graph plotting utility.

Our previous work has observed that the use of hypergraph partitioning to minimise communication during sparse matrix-vector multiplication gives HYDRA good scalability on both parallel computers with fast interconnection networks and also on networks of workstations connected via switched Ethernet [7], [12].

III. Amazon Elastic Compute Cloud

The Amazon Elastic Compute Cloud (Amazon EC2) is a service that allows users both to purchase computing resources on-demand and also to reserve them to guarantee availability in the future. Central to EC2 are Amazon Machine Images (AMIs), which are instantiations of the Linux or Windows operating system that are brought into being by the user and run as virtual machines. Usage is charged per instance per hour, the current rate in the US-East region for the on-demand Linux instances used in this study being \$0.085 per instance per hour[1]. Users manage the creation and termination of instances via a web-page or the Linux command line, and once instances are up and running it is possible to `ssh` into them, as with a physical machine, and execute programs.

Amazon provide a range of standard AMIs, based on Windows and various flavours of Linux, that come pre-installed with commonly-used packages such as MySQL, Apache and Condor. They also provide tools to enable users to build their own AMIs containing exactly the applications and packages that they require, which can then be shared with the wider EC2 community. Both of our tools described in the previous section require MPI and, although none of the standard Amazon AMIs include this, there is a user-produced AMI that does [22], [23]. We have therefore used this AMI as the execution environment for our empirical study. Note that this AMI is only available in the US-East region of EC2, and not in the European region.

IV. Results

Fig. 3 shows a 22-place GSPN model of a flexible manufacturing system [24], which forms the case study for this paper. The model describes an assembly line with three types of machines ($M1$, $M2$ and $M3$) which assemble four types of parts ($P1$, $P2$, $P3$ and $P12$). Initially, there are k unprocessed parts of each type $P1$, $P2$ and $P3$ in the system. There are no parts of type $P12$ at start-up since these are assembled from processed parts of type $P1$ and

[1] See http://aws.amazon.com/ec2/pricing/ for a full list of rates

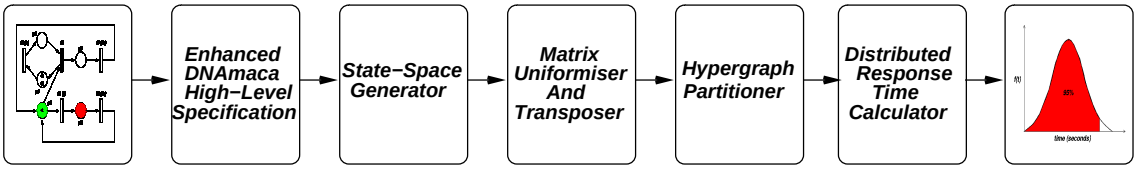


Figure 2. HYDRA tool architecture [12].

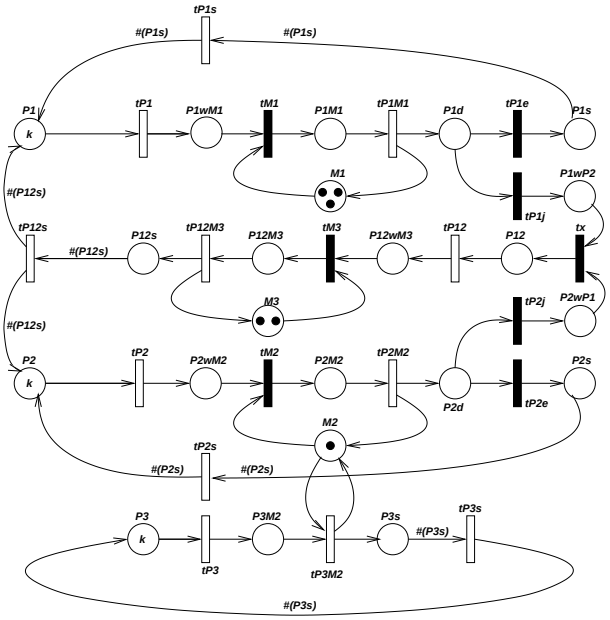


Figure 3. The GSPN model of a Flexible Manufacturing System [24].

$P2$ by the machines of type $M3$. When parts of any type are finished, they are stored for shipping on places $P1s$, $P2s$, $P3s$ and $P12s$.

For $k = 6$, the GSPN's underlying Markov chain has $537\,768$ tangible states, while for $k = 7$ it has $1\,639\,440$ tangible states. In both cases, we compute the response time density from markings where there are k unprocessed parts of types $P1$ and $P2$ to the first marking encountered where a finished part of type $P12$ has been produced.

The two scalability metrics in which we are interested for our tools are speed-up (S_p) and efficiency (E_p). Speed-up is the improvement gained from solving the problem on multiple processors compared with on a single processor, and is calculated as:

$$S_p = \frac{T_1}{T_p}$$

where T_1 is the run-time of the tool on one processor and T_p is the run-time on p processors.

Efficiency represents the proportion of time that a participating processor is busy, and is defined as:

$$E_p = \frac{S_p}{p}$$

where p is the number of processors used. Ideally $S_p = p$ and $E_p = 1$, but in practice the observed values will be lower due to the overheads imposed by communication.

EC2 currently restricts a user to a maximum of 20 running AMI images at any one time, and so this limits the maximum number of parallel processors for which results could be produced. We have previously demonstrated both tools running on more than 16 processors, however, and do not envisage any technical reasons why they could not do so on EC2 were this limit to be raised.

A. Laplace Transform Inverter

Tab. I shows the run-times, speed-ups and efficiencies for the calculation of response time densities on p processors for the FMS model with $k = 6$ using the Laplace transform inversion tool. Corresponding graphs of speed-up and efficiency are shown in Fig. 4. Note that the run-times are averaged over 5 runs.

These results are presented for four architectures. The "PC (2004)" results are reproduced from [8] and were produced on a network of PC workstations linked together by 100Mbps switched Ethernet, each PC having an Intel Pentium 4 2.0GHz processor and 512MB RAM.

We also present new results from a network of modern PCs, a dedicated cluster and Amazon EC2. The "PC (2010)" results were produced on a network of Intel Core2 Duo 3.0GHz processor workstations with 4GB RAM, which were linked together by 1Gbps switched Ethernet.

The "Camelot" cluster consists of 16 dual-processor dual-core nodes, each of which is a Sun Fire x4100 with two 64-bit Opteron 2.2GHz processors and 8GB of RAM. Nodes are connected with both Gigabit Ethernet and Infiniband interfaces; the Infiniband fabric runs at 2.5Gbps and is managed by a Silverstorm 9024 switch.

Each processor in the "Amazon" results is an Amazon EC2 Small Instance running the AMI described in the previous section. This is equivalent to a 1.0-1.2GHz Opteron or Xeon processor with 1.7GB of RAM[2].

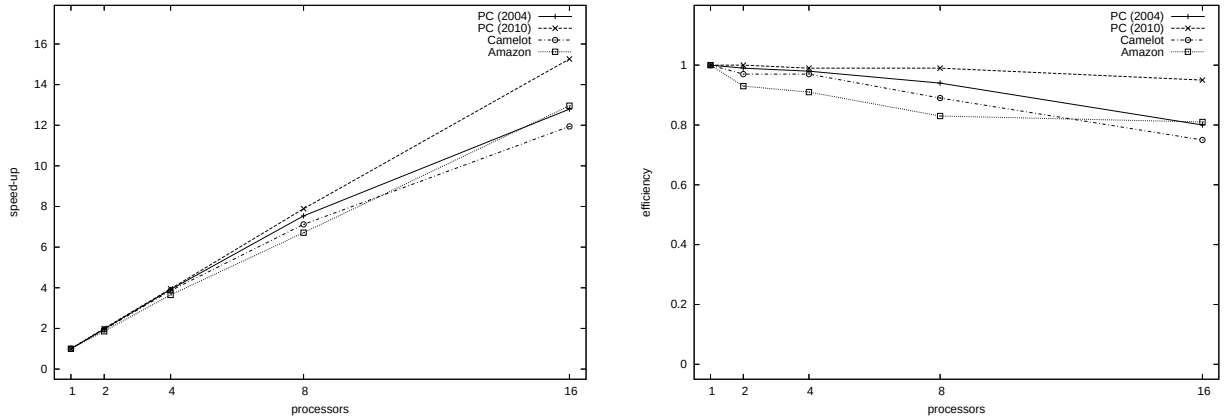
As discussed previously, we would expect the Laplace transform tool to exhibit good scalability as there is very little inter-processor communication, and this is shown to be the case in these results. Indeed, it is noticeable that on EC2 the speed-up trend is almost linear, while on the network of PCs from 2004 and (surprisingly) the Camelot cluster the speed-up trend may be beginning to level-off after 16 processors. These results suggests that the master-slave architecture with minimal intercommunication is an appropriate design for cloud-based parallel tools.

[2]See http://aws.amazon.com/ec2/. Retrieved April 2010.

p	PC (2004)			PC (2010)			Camelot			Amazon		
	T	S_p	E_p	T	S_p	E_p	T	S_p	E_p	T	S_p	E_p
1	5 096.0	1.0	1.0	1 190.5	1.00	1.00	4 181.3	1.00	1.00	2 835.9	1.00	1.00
2	2 582.6	1.97	0.99	592.4	2.00	1.00	2 149.1	1.95	0.97	1 522.4	1.86	0.93
4	1 298.4	3.92	0.98	301.4	3.95	0.99	1 083.1	3.86	0.97	776.2	3.65	0.91
8	675.8	7.54	0.94	150.9	7.89	0.99	587.6	7.12	0.89	422.7	6.71	0.83
16	398.4	12.79	0.80	78.0	15.26	0.95	350.3	11.94	0.75	218.8	12.96	0.81

Table I

AVERAGE RUN-TIMES IN SECONDS (T), SPEED-UPS (S_p) AND EFFICIENCIES (E_p) FOR p-PROCESSOR RESPONSE TIME DENSITY CALCULATIONS IN THE FMS MODEL WITH $k = 6$ USING THE LAPLACE TRANSFORM INVERSION TOOL.



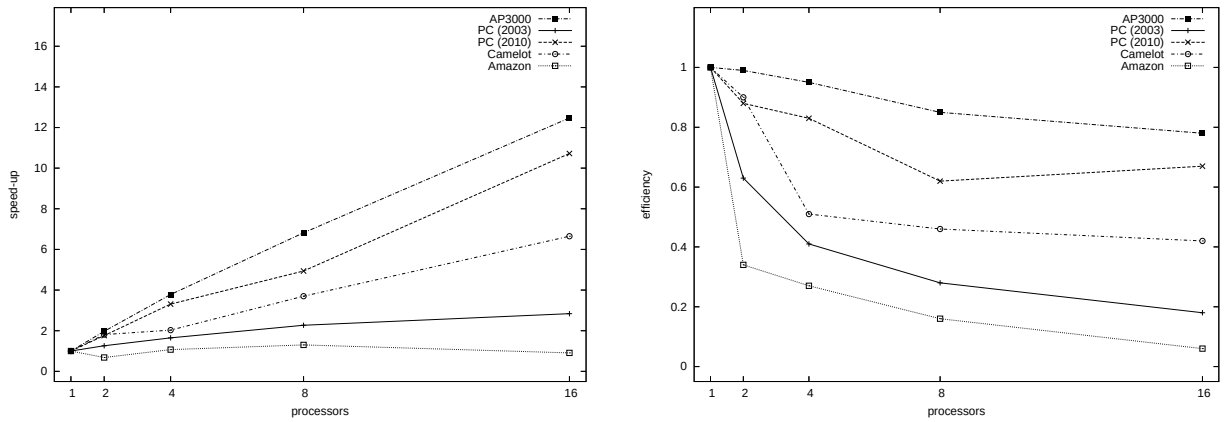
(a) Speed-up (b) Efficiency

Figure 4. Speed-up and efficiency graphs for p-processor response time density calculations in the FMS model with $k = 6$ using the Laplace transform inversion tool.

p	AP3000			PC (2003)			PC (2010)			Camelot			Amazon		
	T	S_p	E_p	T	S_p	E_p	T	S_p	E_p	T	S_p	E_p	T	S_p	E_p
1	1 243.3	1.00	1.00	325.0	1.00	1.00	76.8	1.00	1.00	178.1	1.00	1.00	112.5	1.00	1.00
2	630.5	1.97	0.99	258.7	1.26	0.63	43.5	1.76	0.88	98.7	1.81	0.90	166.2	0.68	0.34
4	328.2	3.78	0.95	197.1	1.65	0.41	23.2	3.31	0.83	87.9	2.03	0.51	104.8	1.07	0.27
8	182.3	6.82	0.85	143.0	2.27	0.28	15.5	4.94	0.62	48.2	3.70	0.46	86.3	1.30	0.16
16	99.7	12.47	0.78	114.6	2.84	0.18	7.2	10.72	0.67	26.8	6.65	0.42	123.4	0.91	0.06

Table II

AVERAGE RUN-TIMES IN SECONDS (T), SPEED-UPS (S_p) AND EFFICIENCIES (E_p) FOR p-PROCESSOR RESPONSE TIME DENSITY CALCULATIONS IN THE FMS MODEL WITH $k = 7$ USING HYDRA.



(a) Speed-up (b) Efficiency

Figure 5. Speed-up and efficiency graphs for p-processor response time density calculations in the FMS model with $k = 7$ using HYDRA.

B. HYDRA

Tab. II shows the run-times, speed-ups and efficiencies for the calculation of response time densities on p processors for the FMS model with $k = 7$ using HYDRA. Corresponding graphs of speed-up and efficiency are shown in Fig. 5. Once again, these run-times were averaged over 5 runs.

These results are presented for five architectures. The "PC (2010)", "Camelot" and "Amazon" environments are as described in the previous section, and we also reproduce two sets of historical results ("AP3000" and "PC (2003)") from [8] to provide a further basis for comparison. The Fujitsu AP3000 was a distributed-memory parallel computer running Solaris. It was based on a grid of 60 processing nodes, each of which had an UltraSPARC 300MHz processor and 256MB RAM. These nodes were interconnected by a 2D wraparound mesh network that used wormhole routing and that had a peak throughput of 520Mbps. The "PC (2003)" results were produced on a network of Athlon 1.4GHz workstations, each with 512MB RAM, that were linked together by a 100Mbps switched Ethernet network.

HYDRA requires much more communication than the Laplace transform inverter, as vector elements must be exchanged between processors after every iteration. Although the use of hypergraph partitioning minimises the amount that must be sent, we observe that the speed-ups achieved are accordingly lower than for the Laplace transformer inverter – although it must be acknowledged that the raw run-times are much faster for HYDRA. It is somewhat galling that the most consistent speed-ups are achieved on a decade-old machine that has now been retired, closely followed by a network of modern workstations! Indeed, the relatively poor performance of the Camelot cluster in these results has led us to investigate whether there might be a hardware or configuration problem, and this is still on-going at the time of writing.

We also observe that the scalability of HYDRA on Amazon EC2 is the worst of all five architectures. Although we expected the speed-up and efficiency to be lower than on the dedicated hardware platforms, it is still very surprising to see just how badly HYDRA fares in the cloud. Hypergraph partitioning does reduce the amount of data sent dramatically (see [8] for further details), but clearly not by enough. No information is available on how the physical machines in the Amazon data-centres are interconnected, but there must be a great deal of contention (both for virtual machines accessing the network cards on the same physical machine, and for access to the network fabric itself). Furthermore we are at the mercy of relatively high network latencies; on dedicated machines the physical nodes are located close together and so latency is low, but when running in the cloud our virtual machines may not be located in the same physical machine, server rack or even data centre.

These results demonstrate that the close coupling of processors in HYDRA's architecture is probably unsuitable for tools designed to run in the cloud. Run-times are still relatively low, so it is not infeasible to use HYDRA in this environment, but to get the best out of large numbers of processors (and remember we are charged per instance that we use) we should look to a more loosely-coupled tool design. These results should not be taken to suggest that running HYDRA on a cloud service like EC2 is entirely without merit, however. Although there is little improvement in run-time, adding extra processors does increase the size of model that can be analysed as each processor only stores a portion of the global state-space.

V. Conclusion

This paper has investigated the scalability of two performance analysis tools on Amazon's Elastic Compute Cloud, and has compared the parallel speed-ups and efficiencies they achieve in that environment with those observed on traditional dedicated computational clusters and networks of workstations. We observed that the Laplace transform tool, with its loosely-coupled master-slave architecture with work delegated on request from a global queue, scaled much better in the cloud than HYDRA, where participating processors had to exchange large amounts of data at every step.

For the future we intend to incorporate these lessons into our next generation of performance analysis tools so that they are able to be used efficiently in cloud computing environments, as well as on clusters and networks of workstations. The master-slave architecture with minimal processor intercommunication has demonstrated scalability on EC2 comparable with that achieved on dedicated machines, and therefore appears a good blueprint for the future. The biggest drawback is the limitation this imposes of having to hold the entire state-space of the model in the memory of one machine, whereas with HYDRA it is distributed across multiple machines. Our recent work on aggregation suggests ways in which the state-spaces of models could be reduced in size, however [25]. We will also investigate the benefits of exploiting Amazon's dedicated Elastic Block Store (EBS) to produce a disk-based tool [26], [27], [28].

In the interim we could also ease the deployment of our existing tools on EC2 by creating our own custom AMIs that package them alongside MPI and other required libraries. HYDRA does work on EC2, albeit without scaling very well, and so such a move would still be useful to performance analysts. Purpose-built tools could be made to work far more efficiently, however.

References

[1] M. Benzi and M. Tuma, "A parallel solver for large-scale Markov chains," *Applied Numerical Mathematics*, vol. 41, pp. 135–153, 2002.

[2] P. Buchholz, M. Fischer, and P. Kemper, "Distributed steady state analysis using Kronecker algebra," in *Proceedings of the 3rd International Conference on the Numerical Solution of Markov Chains (NSMC'99)*, Zaragoza, Spain, September 1999, pp. 76–95.

[3] W. Knottenbelt, "Parallel performance analysis of large Markov models," Ph.D. dissertation, Imperial College, London, United Kingdom, February 2000.

[4] A. Ogielski and W. Aiello, "Sparse matrix computations on parallel processor arrays," vol. 14, no. 3, pp. 519–530, May 1993.

[5] J. Bradley, N. Dingle, W. Knottenbelt, and H. Wilson, "Hypergraph-based parallel computation of passage time densities in large semi-Markov models," *Linear Algebra and its Applications*, vol. 386, pp. 311–334, 2004.

[6] N. Dingle, P. Harrison, and W. Knottenbelt, "Response time densities in Generalised Stochastic Petri Net models," in *Proceedings of the 3rd International Workshop on Software and Performance (WOSP'02)*, Rome, July 24th–26th 2002, pp. 46–54.

[7] ——, "Uniformization and hypergraph partitioning for the distributed computation of response time densities in very large Markov models," *Journal of Parallel and Distributed Computing*, vol. 64, no. 8, pp. 908–920, August 2004.

[8] N. Dingle, "Parallel computation of response time densities and quantiles in large markov and semi-markov models," Ph.D. dissertation, Imperial College London, United Kingdom, 2004.

[9] P. Harrison and W. Knottenbelt, "Passage time distributions in large Markov chains," in *Proceedings of ACM SIGMETRICS 2002*, Marina Del Rey CA, June 2002, pp. 77–85.

[10] Imperial College High Performance Computing Service, http://www3.imperial.ac.uk/ict/services/teachingandresearchservices/highperformancecomputing.

[11] HECToR: UK National Supercomputing Service, www.hector.ac.uk.

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

[13] J. Abate, G. Choudhury, and W. Whitt, "An introduction to numerical transform inversion and its application to probability models," in *Computational Probability*, W. Grassman, Ed., Kluwer, Boston MA, 2000, pp. 257–323.

[14] ——, "On the Laguerre method for numerically inverting Laplace transforms," *INFORMS Journal on Computing*, vol. 8, no. 4, pp. 413–427, 1996.

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

[16] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: Portable Parallel Programming with the Message Passing Interface*. Cambridge MA: MIT Press, 1994.

[17] G. Bolch, S. Greiner, H. Meer, and K. Trivedi, *Queueing Networks and Markov Chains*. Wiley, August 1998.

[18] B. Melamed and M. Yadin, "Randomization procedures in the computation of cumulative-time distributions over discrete state Markov processes," *Operations Research*, vol. 32, no. 4, pp. 926–944, July–August 1984.

[19] A. Miner, "Computing response time distributions using stochastic Petri nets and matrix diagrams," in *Proceedings of the 10th International Workshop on Petri Nets and Performance Models (PNPM'03)*, Urbana-Champaign, IL, September 2nd–5th 2003, pp. 10–19.

[20] J. Muppala and K. Trivedi, "Numerical transient analysis of finite Markovian queueing systems," in *Queueing and Related Models*, U. Bhat and I. Basawa, Eds. Oxford University Press, 1992, pp. 262–284.

[21] U. Catalyürek and C. Aykanat, "Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication," vol. 10, no. 7, pp. 673–693, July 1999.

[22] P. Skomoroch, "Data Wrangling Image: Fedora Core 6 MPI Compute Node with Python Libraries," AMI ID: ami-3e836657, http://developer.amazonwebservices.com/connect/entry.jspa?categoryID=101&externalID=705. Retrieved April 2010.

[23] ——, "MPI cluster programming with Python and Amazon EC2," in *Proceedings of the 6th Annual Python Community Conference (PyCon 2008)*, Chicago, March 2008, http://www.datawrangling.com/mpi-cluster-with-python-and-amazon-ec2-part-2-of-3. Retrieved April 2010.

[24] G. Ciardo and K. Trivedi, "A decomposition approach for stochastic reward net models," *Performance Evaluation*, vol. 18, no. 1, pp. 37–59, 1993.

[25] M. Günther, N. Dingle, J. Bradley, and W. Knottenbelt, "Passage-time computation and aggregation strategies for large semi-markov processes," *Performance Evaluation*, under revision.

[26] D. Deavours and W. Sanders, "An efficient disk-based tool for solving large Markov models," *Performance Evaluation*, vol. 33, no. 1, pp. 67–84, June 1998.

[27] M. Kwiatkowska and R. Mehmood, "Out-of-core solutions of large linear systems of equations arising from stochastic modelling," in *Proceedings of Process Algebra and Performance Modelling (PAPM'02)*, Copenhagen, Denmark, July 25th–26th 2002, pp. 135–151.

[28] W. Knottenbelt and P. Harrison, "Distributed disk-based solution techniques for large Markov models," in *Proceedings of the 3rd International Conference on the Numerical Solution of Markov Chains (NSMC'99)*, Zaragoza, Spain, September 1999, pp. 58–75.