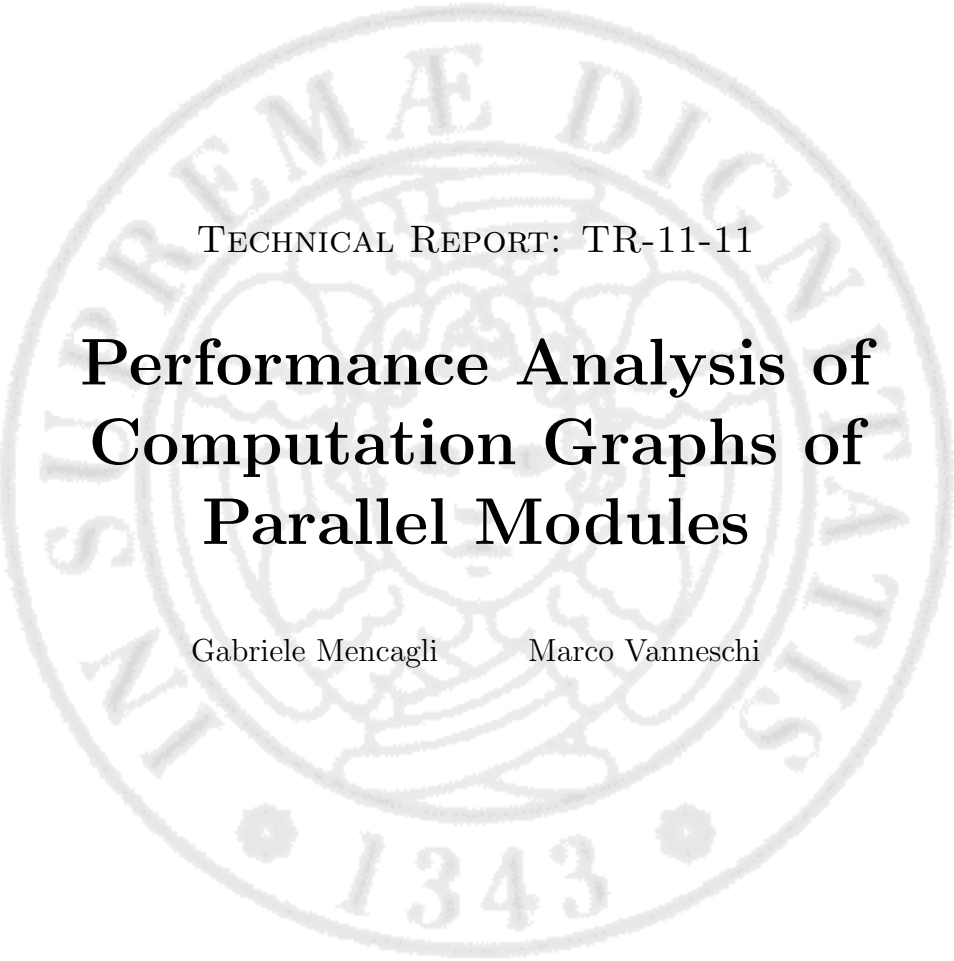
Università di Pisa
Dipartimento di Informatica



Technical Report: TR-11-11

Performance Analysis of Computation Graphs of Parallel Modules

Gabriele Mencagli Marco Vanneschi

August 2, 2011

Address: Largo B. Pontecorvo 3, 56127 Pisa, Italy. Tel: +39 050 2212700 Fax: +39 050 2212726

Performance Analysis of Computation Graphs of Parallel Modules

Gabriele Mencagli Marco Vanneschi

August 2, 2011

Abstract

In this report we provide the fundamental results for applying a formal performance modeling of distributed parallel computations described as computation graphs of parallel modules. In our approach parallelism is expressed inside each module (based on structured parallelism schemes) and between different modules that can compose general graph structures. Our methodological approach, based on Queueing Theory and Queueing Network Theory foundations, provides the necessary tools for predicting the steady-state behavior of a parallel computation.

1 Introduction

From a general point of view, a parallel application can be represented as a directed application graph (*work-flow*) of independent modules cooperating by exchanging typed messages[1]. Modules can exchange single values or, as in stream-based computations, a sequence of messages, by means of communication channels. In this context we can distinguish between two different levels of parallelism:

- ***intra-module parallelism***: the module computation is activated by receiving messages (i.e. tasks) from a set of source modules, according to a non-deterministic or data-flow semantics. For each activation the module starts either a sequential or a parallel computation. In the latter case the internal parallelization follows a specific structured paradigm: i.e. intra-module parallelism is expressed by instantiating well-known structured parallelization schemes for which the parallel organization and the properties of the parallelization approach are known and clearly identified. This aspect represents a fundamental characteristic of our methodology;
- ***inter-module parallelism***: modules can be composed in computation graphs of general structure. Modules can represent different subjects taking part of the whole application (e.g. as in a client-server system), or can

[1] In the rest of this report we will assume a classical local environment model (also called message-passing) for parallel computations.

correspond to different application phases involving complex and time-consuming processing.

The methodology that we are introducing is aimed to completely model the performance at any level, formalizing the internal behavior of a single module and the performance of the entire computation graph.

In[1] a performance modeling for steam-based parallel computations is expressed in terms of fundamental results in the area of Queueing Theory and Queueing Networks. These theories are a starting point that allow us to formalize important issues related to:

- how to evaluate the performance of a graph computation starting from the knowledge of the theoretical performance of each module;
- how to evaluate the effective performance of a module based on its theoretical performance and the performance behavior of the other modules of the computation graph;
- how to detect bottlenecks in a parallel computation, that is modules that seriously limit the performance of the entire application.

A basic point consists in modeling the performance of a module M (e.g either sequential or internally parallel) by abstracting its behavior as a *queueing system*, as shown in Figure 1. This scheme is a logical one, not necessarily

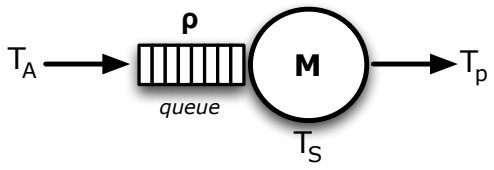


Figure 1: A computation module modeled as a queueing system.

corresponding to the real implementation. However, it is aimed at capturing the essential elements of the problem at hand. For example, in same real cases there are distinct communication channels between modules in both directions: a single queue in front of M could not exist physically, however it is emulated by a set of channel buffers in the source and destination nodes. The behavior of a queueing system is characterized by expressing five different parameters:

1. The *service discipline*: if not explicitly defined the FIFO policy is assumed;
2. The *queue size*, that is the number of buffer positions available for storing the in-coming requests to the module;
3. The probability distribution of a random variable *inter-arrival time* t_a (i.e. time interval between two consecutive arrivals of requests), with average value T_A and variance σ_A;

4. The probability distribution of a random variable *service time* t_s, which represents the theoretical time passing between the beginning of the executions on two consecutive stream elements. This parameter depends only on the internal features of the module *in isolation*, not considering the interactions with other modules of the computation graph. We denote with T_S and σ_S the average value and the variance of this random variable;

5. The probability distribution of a random variable *inter-departure time* t_p (with average value T_p and variance σ_P), which indicates the time between two successive result departures from the module.

A central parameter for our performance evaluation is the ***utilization factor*** ρ of the queueing system, defined by the following ratio:

$$\rho = \frac{T_S}{T_A} \tag{1}$$

It expresses a global, average measure of the congestion degree, or traffic intensity, of requests to the queueing system. Large values represent high congestion degrees whereas small utilization factors consist in a more limited workload condition to the node.

With the modeling introduced above, each application module can be abstracted as a queueing system and the computation graph can be described as a network of queues[2], where the departures of some nodes form the arrivals of others. From the network topology viewpoint, queueing networks can be categorized into two broad classes namely **open queueing networks** and **closed queueing networks**. In an open queueing network a possibly infinite number of requests are generated by source nodes, go through several nodes or even revisit a particular node more than once and finally leave the system. On the other hand, in a closed queueing network requests neither arrive at nor depart from the system, but a fixed number of requests circulate through the nodes of the network. Open and closed networks are powerful modeling tools that have been applied for realistically formalizing the performance behavior of different classes of systems. Open networks have been used for modeling flows as in traffic models and notably in data networks. Closed networks are considered a valuable tools for modeling systems where there exists a finite input population. CPU scheduling[3], supply-chain manufacturing systems[4] and window-type network flow control[5] are typical examples in this sense.

Several classes of stream-based parallel computations can be modeled using queueing network models. In this section we will use different modeling techniques for two general computation graph structures:

Acyclic computation graphs describe complex distributed applications involving several computing phases. A large set of tasks is generated by source modules in the computation graph. Each task passes through the module following a certain routing strategy: each module performs a specific elaboration for each received task. For modeling the performance of such applications we will use *acyclic open queueing networks*: i.e. a request in the network can pass through any particular node at most once.

Cyclic computation graphs describe parallel computations exhibiting a request-reply behavior. Notable examples are *client-server* applications in which client modules transit requests and then wait for the corresponding results from a server module. In this case the performance behavior will be studied by using *cyclic closed queueing network* models.

Based on the previous distinction in the rest of this section we will describe the performance modeling of these two classes of computation graphs. The following considerations will be completely independent w.r.t the internal behavior of each computation module, i.e. if it is sequential or internally parallel. In fact, the only thing that we need to know is the theoretical service time of each module that composes a graph. The behavior of intra-module parallelism will be based on the very same results provided in the following sections for general application graphs.

2 Acyclic Computation Graphs: analytical treatment

For acyclic graphs, Queueing Networks theory is a sufficiently powerful methodology for our modeling purposes. It does not utilize an explicit analytical treatment in terms of probability distributions, instead the performance modeling is expressed in terms of some basic results about the information flow in the network, the presence of bottlenecks and the average values of inter-arrival and service time variables [2]. In the rest of the discussion *we will assume that each computation module produces exactly one output stream value for each received task*. This assumption simplifies the model construction without limiting its scope, since many stream-based computations can be described as a graph following this behavior (or semantically reducible to it).

In order to evaluate acyclic graphs of computation modules we consider two interrelated phases:

- ***Transient analysis*** consists in a formal studying of the network behavior in the initial *transient phase* of the execution. For transient phase we intend the initial situation in which the mean inter-departure time from each node can significantly change at relatively short time periods due to the starting conditions of the network (e.g. the theoretical service time of the nodes and their maximum queue size). This analysis is aimed to evaluate for each node its utilization factor and to discover the presence of *bottlenecks*: when $\rho > 1$ a node represents a bottleneck, since it is not able to process the in-coming requests at their arrival rate but the information flow is delayed by the node presence;

- ***Steady-State analysis*** provides results for evaluating the effective performance (i.e. the mean inter-departure times) of each node in the network

[2] For the acyclic graph modeling if not specifically expressed we intend service time and inter-arrival time values as average measurements.

during the *steady-state phase*. For steady-state we intend the situation in which the mean inter-departure times are completely stabilized and the network behavior is no longer influenced by the initial conditions.

Both during the transient and the steady-state phase, the mean inter-departure time from each node may be higher than its theoretical mean service time. In particular $T_p = T_S + \Delta$ where $\Delta \geq 0$ is a delay induced by two possible situations that may happen:

- a node M may receive service requests with an average inter-arrival time higher than its service time. This means that after the completion of the service on a stream element, the node must wait (it is blocked) for the reception of the next one before starting the successive service;
- in real systems queues have a fixed maximum size in terms of in-coming tasks received by other nodes. If a task attempts to enter a full capacity destination queue upon completion of a service at node M, it is forced to wait in this node until the destination node has a free position in its queue. During this phase the source module M stops processing tasks (it is blocked) until destination node completes a task service. This behavior is known as *blocking-after-service*[6] and is an effective modeling for computing modules that asynchronously exchange messages onto channels with a limited buffer size.

Let us suppose that during our graph analysis we discover a node M:

- if the mean inter-arrival time to M is greater than its service time (i.e. during the transient phase its utilization factor is less than 1), the inter-departure time from M equals its inter-arrival time and the node is not a bottleneck (at steady-state its utilization factor ρ keeps to be lower than 1);
- if the mean inter-arrival time is lower than the mean service time, M is a bottleneck and during the transient phase its utilization factor is greater than 1. When its input queue becomes full, upstream nodes start to be blocked and the effect is that at steady-state the effective inter-arrival time to M will be increased in such a way as to coincide with its average service time. *This means that the condition $\rho > 1$ is only a transient one.*

Therefore the following proposition is verified by flow conservation at each node of a queueing network:

Proposition 2.1 (Steady-State behavior of a node)**.** *At steady-state for each node its effective mean inter-arrival time is equal to its mean inter-departure time. If the inter-arrival time also coincides with the average service time of the node, the node is a bottleneck and its utilization factor stabilizes to* 1*. Otherwise, if a node is not a bottleneck, its utilization factor stabilizes to a value less than* 1*.*

In the rest of this section we will provide the basic results for studying acyclic computation graphs of any form. In particular we are interested in some analytical results that allow us to study the graph behavior during the transient phase, identify the bottleneck nodes and their blocking effects on the other nodes in such a way as to determine the long-term, steady-state behavior of the graph.

In the following discussion we will start by considering deterministic arrivals and service times: i.e. initially we will suppose constant service times (with zero variance) for each node of the network. Later in this report we will describe the impact of randomness on the provided results.

2.1 Analysis of Tandem Queueing Systems

We start from a first situation in which two nodes are joined in series as depicted in Figure 2. Requests are generated by the first node S_1 and will join the next one S_2. In other words the departing requests from the first node form the arrivals to the second one. Let us suppose that these two nodes are

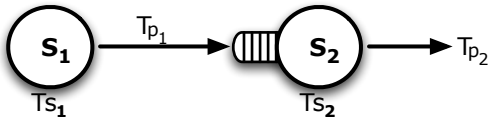


Figure 2: Two-queue tandem system.

characterized by theoretical service times T_{S1} and T_{S2} respectively. During the transient phase the utilization factor of the second node is determined by the following ratio:

$$\rho_{S_2} = \frac{T_{S2}}{T_A} = \frac{T_{S2}}{T_{S1}}$$

At the beginning of the system execution, for the second node its inter-arrival time T_A corresponds to the service time of the first node of the network. At this point we need to evaluate what nodes are bottlenecks in this simple graph and the effective behavior of each node at steady-state, that is the inter-departure times T_{p1} and T_{p2} from the two nodes.

According to the utilization factor definition, the second node is a bottleneck iff its utilization factor is greater than 1 (i.e. if $T_{S2} > T_{S1}$). Let us consider the case in which this is not true, so the service time of the first node is not smaller than the second one: $T_{S1} \geq T_{S2}$. This scenario is simulated in Figure 3. In this case the bottleneck node of the graph is the first one. As we have seen if a node is a bottleneck, at steady-state its service time coincides with its inter-departure time (and also to its fictitious inter-arrival time) because the node is never blocked due to communications with other nodes in the network. Therefore we have that $T_{p1} = T_{S1}$. For the second node its utilization factor is less than 1 (the node is under-utilized) and it is periodically blocked for receiving a new request from S_1. From Figure 3 we can see that S_2 is delayed by a waiting

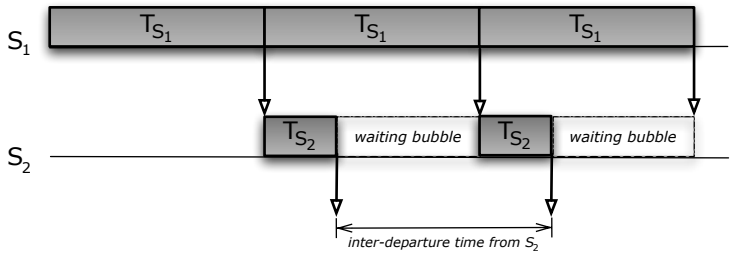


Figure 3: Two-queue tandem analysis: first node is the bottleneck.

bubble Δ equals to $T_{S1} - T_{S2}$. Thus the inter-departure time from S_2 is given by:

$$T_{p2} = T_{S1} + \Delta = T_{S2} + (T_{S1} - T_{S2}) = T_{S1}$$

This means that during the transient phase (and also during the steady-state phase too) the inter-departure time from the second node equals the service time of the first one, if S_1 is the bottleneck node of the tandem network.

The opposite case considers the situation in which the utilization factor of the second node is greater than 1, thus $T_{S2} > T_{S1}$. After an initial transient phase the queue of the second node becomes full and the steady-state behavior is depicted in Figure 4. Upon the completion of the current service, S_1 is not

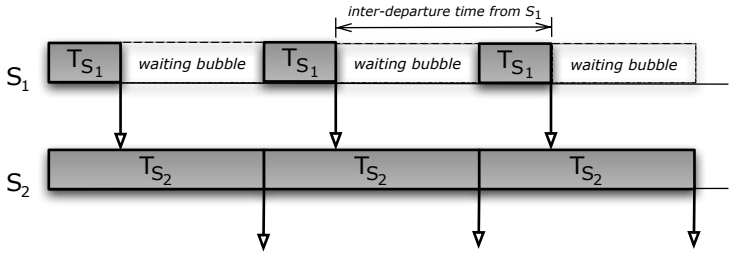


Figure 4: Two-queue tandem analysis: second node is the bottleneck (steady-state behavior).

able to transmit the next request to the second node S_2 until this node frees a position in its queue. This means that the first node is delayed by the remaining service time in the second node, which is equal to $\Delta = (T_{S2} - T_{S1})$. Hence at steady-state the inter-departure time from the first node becomes:

$$T_{p1} = T_{S1} + \Delta = T_{S1} + (T_{S2} - T_{S1}) = T_{S2}$$

Therefore the inter-departure time from the first node equals the theoretical service time of the second node. Moreover, since the second node is the bottleneck of the graph, also its inter-departure time equals its service time: i.e. $T_{p2} = T_{S2}$.

We can summarize the previous results concerning a two-queue tandem system:

$$T_{p1} = T_{p2} = \max\{T_{S1}, T_{S2}\} \tag{2}$$

At steady-state the effective behavior of the two nodes, that is their inter-departure times, are equal to the maximum theoretical service time of the two nodes of the network.

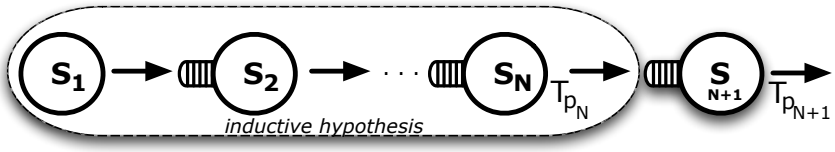


Figure 5: Performance analysis of a pipeline graph.

The previous results can be generalized to a tandem system of an arbitrary number of nodes (see Figure 5), also known as the *pipeline* graph. In this case the following proposition holds:

Proposition 2.2 (Pipeline)**.** *In a pipeline graph of any length the bottleneck is the node with the largest theoretical service time. Moreover, the inter-departure time from each node of the graph is equal to that theoretical service time.*

Proof. The proposition can be proved by induction on the pipeline length. The two-queue tandem system is the base case which has been already demonstrated earlier. Hence we can directly consider the inductive case: we have a pipeline of N nodes and we know from the inductive hypothesis that the inter-departure time T_{p_i} from each node is equal to the maximum theoretical service time of the N nodes in the graph: i.e. $\forall i = 1, 2, \dots N \; T_{p_i} = T_{S_Z} = \max_{j=1}^{N}\{T_{S_j}\}$, where S_Z, with $1 \leq z \leq N$, is the bottleneck node. Now, in order to complete the inductive reasoning, we consider the presence of a further node S_{N+1} with theoretical service time $T_{S_{N+1}}$, which is added at the end of the pipeline graph (see Figure 5). We have two possible situations:

- If $T_{S_{N+1}} \leq T_{S_Z}$ the inter-arrival time to S_{N+1} (which is equal to the inter-departure time from S_N i.e. $T_{p_N} = T_{S_Z}$) is greater than the service time of the new node. We are in the very same scenario as the one depicted in Figure 3. The new node is periodically blocked to wait for the reception of the next request from node S_N. This means that the inter-departure time from S_{N+1} equals its inter-arrival time (both during the transient and the steady-state phase), that is $T_{p_{N+1}} = T_{p_N} = T_{S_Z}$. Therefore the proposition is verified;

- Let us consider the case $T_{S_{N+1}} > T_{S_Z}$. During the transient phase the inter-arrival time to the new node is equal to the inter-departure time from the last node of the pipeline (i.e. $T_{p_N} = T_{S_Z}$) and it is lower than

the theoretical service time $T_{S_{N+1}}$ of the new node. This means that we are in the case depicted in Figure 4. Since $T_{S_{N+1}} > T_{S_Z} = \max_{j=1}^{N}\{T_{S_j}\}$, S_{N+1} becomes the new bottleneck of the graph and its inter-departure time equals its service time, i.e. $T_{p_{N+1}} = T_{S_{N+1}}$. At steady-state the node S_N is periodically blocked for transmitting a new request to S_{N+1}, and for Proposition 2.1 its inter-departure time adapts to the service time of the new bottleneck node $T_{p_N} = T_{S_{N+1}}$. The reasoning can be repeated for node S_{N-1} up to node S_1, proving that the inter-departure time from each node of the pipeline equals the service time of S_{N+1} (i.e. the new bottleneck of the graph).

□

2.2 Analysis of a Queueing System with Multiple Destinations

Let us consider the case of a node with multiple out-going communications with other nodes (Figure 6). Let us suppose to have a graph composed of a source node S and a set of destination nodes $D_1, D_2, \ldots, D_N$. In general destination nodes are able to provide distinct services: each request from S is routed to a specific destination node according to a certain probability distribution. Let p_i the probability that a request from S is directed to the destination node D_i, where:

$$\sum_{i=1}^{N} p_i = 1$$

If T_{D_i} is the service time of any D_i, a crucial point is to determine the inter-arrival time T_{A_i} to each destination node and thus if they are bottlenecks or not. If we denote with T_{p_S} the inter-departure time from the source node S,

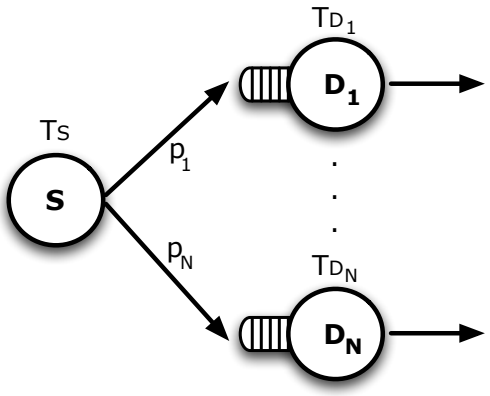


Figure 6: Example of a queueing system with multiple destinations.

we can determine the inter-arrival time to each destination during the initial transient phase:

Proposition 2.3 (Inter-arrival time during the transient phase)**.** *During the initial transient phase, the inter-arrival time T_{A_i} to each destination D_i is given by:*

$$T_{A_i} = \frac{T_{p_S}}{p_i} \tag{3}$$

Proof. Node S transmits requests to a particular destination node D_i with probability p_i and to the set of the other destination nodes with probability $1 - p_i$. For this reason the inter-arrival time t_{A_i} is a random discrete variable with the following distribution:

t_{A_i}	Probability
T_{p_S}	p_i
$2 \cdot T_{p_S}$	$p_i \cdot (1 - p_i)$
$\dots$	$\dots$
$N \cdot T_{p_S}$	$p_i \cdot (1 - p_i)^{N-1}$

Thus the average inter-arrival time T_{A_i} is given by[3]:

$$T_{A_i} = \sum_{n=0}^{\infty} n\, T_{p_S}\, p_i\, (1-p_i)^{n-1} = \frac{p_i\, T_{p_S}}{(1-p_i)} \sum_{n=0}^{\infty} n\, (1-p_i)^n = \frac{p_i\, T_{p_S}}{(1-p_i)} \cdot \frac{(1-p_i)}{p_i^2} = \frac{T_{p_S}}{p_i}$$

□

The previous result allows us to determine the utilization factor for each destination node, and thus to establish if some of them are bottlenecks or not. If no destination node is a bottleneck, its inter-arrival time is greater than its service time, i.e. $T_{A_i} \geq T_{D_i}$. Therefore in this case the inter-departure time T_{p_i} from each destination node (both during the transient and the steady-state phase) equals its inter-arrival time: i.e. $T_{p_i} = T_{A_i}$.

On the other hand, if at least one destination node is a bottleneck, all the inter-arrival times can no more be derived independently each other: i.e. the bottlenecks influence the inter-arrival times to the other destination nodes of the network. Fortunately we can prove that only the worst bottleneck node, that is that with the highest utilization factor influences the inter-arrival times. In fact the following proposition holds:

Proposition 2.4 (Steady-State analysis of a multiple-destination queue)**.** *If at least one destination node is a bottleneck ($\exists i : \rho_i > 1$), let us denote D_Z the node with the highest utilization factor: i.e. $\rho_z = \max_{i=1}^{N} \rho_i$. The effective*

[3]We use the general property: $\sum_{n=0}^{\infty} n\, x^n = \frac{x}{(1-x)^2}$ for $x < 1$.

(steady-state) inter-arrival time to any destination node D_i with $i = 1, \ldots, N$ is obtained by correcting the result of Proposition 2.3 due to the bottleneck presence. The effective inter-arrival time is determined by the following expression:

$$T'_{A_i} = T_{D_Z} \cdot \frac{p_z}{p_i} \tag{4}$$

Proof. The bottleneck existence introduces a delay in the S activity with probability p_z, i.e. the steady-state inter-departure time from S is increased w.r.t the transient one T_{p_S}. This influences the steady-state inter-arrival times to the other destination nodes. From Proposition 2.1 the new inter-departure time from S is a value such that:

$$\frac{T'_{p_S}}{p_z} = T_{D_z} \Longrightarrow T'_{p_S} = p_z \cdot T_{D_z} \geq p_z \cdot T_{A_z} = p_z \cdot \frac{T_{p_S}}{p_z} = T_{p_S}$$

The inequality holds since D_Z is a bottleneck (i.e. $\rho_z > 1$).

At this point we can show that, after the correction of the inter-departure time from S, no destination node can have an utilization factor higher than 1. This can be proved by absurd. Suppose that exists a destination node D_j different from D_Z which remains a bottleneck after correcting the inter-departure time from S. This means that the following inequality is verified:

$$\frac{T_{D_j}}{T'_{A_j}} > 1$$

where T'_{A_j} is the corrected inter-arrival time to D_j. By expanding the expression we have:

$$\frac{T_{D_j}}{T'_{p_S}} \cdot p_j = \frac{T_{D_j}}{T_{D_Z} \cdot p_z} \cdot p_j > 1$$

That can be transformed into:

$$T_{D_j} \cdot p_j > T_{D_Z} \cdot p_z \Rightarrow \frac{T_{D_j} \cdot p_j}{T_{p_S}} > \frac{T_{D_Z} \cdot p_z}{T_{p_S}} \Rightarrow \rho_j > \rho_z$$

which is absurd since by initial hypothesis we have assumed that D_Z was the destination node with the highest utilization factor. The consequence of this fact is that, as steady-state, no further destination node is a bottleneck anymore. Therefore at this point we can apply Proposition 2.3 in order to find the effective inter-arrival time to the destination nodes, which is given by:

$$T'_{A_i} = \frac{T'_{p_S}}{p_i} = T_{D_z} \cdot \frac{p_z}{p_i}$$

□

With the previous results, only D_z is the real bottleneck of the graph that influences the steady-state behavior of all the other nodes. This means that its theoretical service time coincides with its inter-departure time (and also with its corrected inter-arrival time). For all the other destinations instead, their steady-state inter-arrival times are higher than their service times and they coincide with their inter-departure times: i.e. $T_{p_i} = T'_{A_i}$.

2.3 Analysis of a Queueing System with Multiple Sources

Let us consider a node S that accepts service requests from a set of sources (clients) $C_1, C_2, \ldots, C_N$ (see Figure 7). Let us denote with T_S the service time of S and with T_{p_i} the inter-departure time from each client C_i. We can note

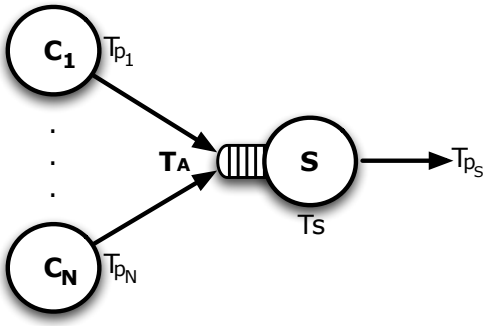


Figure 7: Example of a queueing system with multiple sources.

that this graph does not contain cycles: the node S starts a service whenever a request is present in its queue and the results are transmitted outside the depicted network, e.g. to further destination nodes. The total inter-arrival time T_A to S can be determined by applying the proposition:

Proposition 2.5 (Aggregate inter-arrival time)**.** *If a node S has multiple sources each one with an inter-departure time T_{p_i} to S, the total inter-arrival time to S is given by:*

$$T_A = \frac{1}{\sum_{i=1}^{N} \frac{1}{T_{p_i}}} \tag{5}$$

Proof. As it is known the inverse of the inter-arrival time is the arrival rate (or frequency), that is the number of requests received in a time unit. In the previous graph structure the arrival rate from each client is equal to:

$$\lambda_i = \frac{1}{T_{p_i}}$$

The total number of requests received by S in a time unit can be simply determined by summing the individual arrival rates from each clients: i.e. $\lambda_{tot} = \sum_{i=1}^{N} \lambda_i$. Thus we have:

$$T_A = \frac{1}{\lambda_{tot}} = \frac{1}{\sum_{i=1}^{N} \lambda_i} = \frac{1}{\sum_{i=1}^{N} \frac{1}{T_{p_i}}}$$

□

Once the aggregate inter-arrival time to S has been determined, we can calculate its utilization factor. if $\rho_S \leq 1$ the inter-departure time from S (both during the transient and the steady-state phase) equals its inter-arrival time (the node is periodically blocked for receiving the tasks from clients), i.e. $T_{p_S} = T_A$ and client inter-departure times continue to be equal to T_{p_i}.

On the other hand if $\rho_S > 1$, the node is a bottleneck and its inter-departure time equals its theoretical service time: i.e. $T_{p_S} = T_S$. In this case we need to determine the steady-state inter-departure times from clients that will certainly be greater than the original ones. In function of the specific probability distributions of service times (e.g. deterministic and exponential distributions are notable cases), this problem can be studied in a "stand-alone" fashion, proving approximated results based on queueing theory. Instead in the next section we will provide an elegant and effective approach for determining the performance analysis of acyclic graphs which is valid for a broad range of graph structures.

2.4 An algorithm for the Performance Analysis of Single-Source graphs

With the previous results we have the basic tools for evaluating the performance of acyclic graphs modeling complex application work-flows. An important problem that we need to study is how we can apply the previous results in order to define an automatic procedure for exploiting the performance analysis of acyclic graphs. In particular we need an algorithm designed for solving the following problem:

Problem 2.6 (Steady-State analysis of acyclic computation graphs)**.** *Given an acyclic computation graph $G = (V, E)$ in which each node represents a computation module, we need a procedure that determines the inter-departure times, that is the effective performance behavior at steady-state, of each node in the graph.*

In Figure 8 is depicted an example of an acyclic computation graph of seven modules working asynchronously and cooperating by exchanging messages. In the graph each node is labeled with its theoretical service time (let us consider t as a standardized time unit) and when a node has multiple in-coming edges, they are labeled with the corresponding probability of reception. We need an algorithm that has the following features:

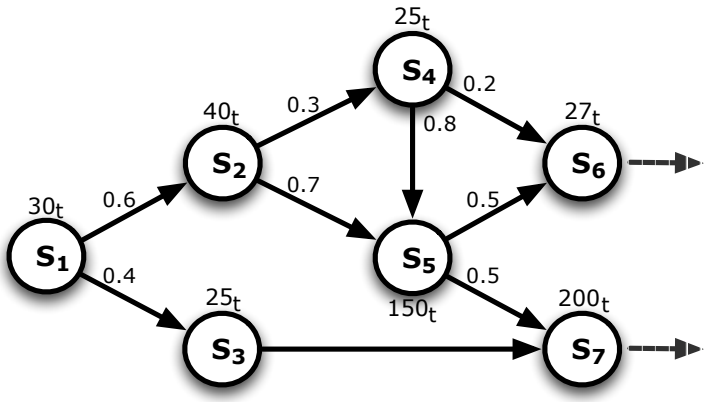


Figure 8: An acyclic computation graph labeled with the service times of each node.

- it needs to perform an ordered graph traversal: to correctly establish for each node its inter-arrival time and thus its utilization factor, each node should be visited only when all its in-coming neighbors have been visited and their inter-departure times correctly determined;
- for each node of the graph it is necessary to calculate its inter-arrival time and its utilization factor in order to discover if it is a bottleneck or not. We have two possible situations. (1) The currently visited node is not a bottleneck: its service time is equal or less than its inter-arrival time and substantially the node does not influence the inter-departure times neither of the already visited nodes nor of the nodes that are still to be explored. (2) The current node is a bottleneck: as we have seen the condition $\rho > 1$ is only a transient one because the bottleneck presence influences the inter-departure times of the previously explored nodes (they will be properly incremented).

The first requirement implies that the nodes of the graph should be visited according to a specific ordering. As it is known from basic notions in Graph Theory, every directed acyclic graph has at least one *topological ordering*, i.e. an ordering of its nodes such that the starting vertex of every edge occurs earlier in the ordering than the ending vertex. It can also be shown that an acyclic graph can have multiple admissible topological orderings whereas, as a very special case, pipeline graphs admit only a unique ordering (i.e. they can be mapped onto a total ordering relation). Therefore let us suppose to have one of these topological orderings as input of the algorithm. We can observe that if the algorithm visits the nodes following this ordering, the first requirement will be achieved: each node will be visited iff all its in-coming neighbors have already been explored. An example of a topological ordering of the graph in Figure 8 is

depicted in Figure 9.

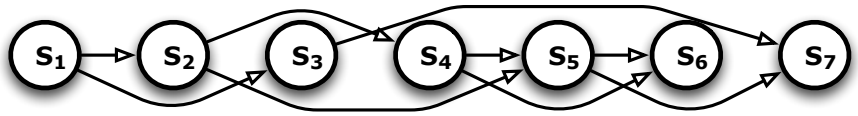


Figure 9: A topological ordering of the graph depicted in Figure 8.

We need a data-structure that represents each node of the graph. Each node n has four numerical attributes that describe: (1) its inter-arrival time; (2) its service time; (3) its inter-departure time; (4) its utilization factor. Moreover the node maintains a list **OUT** of references to out-going neighbors and a list **IN** of pairs $(n^{'}, p)$, where $n^{'}$ is a reference to one of its in-coming neighbors which transmits to n with probability p.

Function `of a node data-structure`

```
1 Class Node {
2     double T_A;
3     double T_S;
4     double T_p;
5     double ρ;
6     ListNode OUT;
7     ListPair IN;
8 }
```

This data-structure is properly initialized when the algorithm starts. For each node the service time variable and the IN and OUT lists are properly initialized according to the structure of the input graph. The inter-arrival and the inter-departure times will be calculated by the algorithm. For each sink node we assume the presence of a fictitious out-going edge such that the inter-departure time can always be defined. For each source node (although in-coming edges do not exist), the inter-arrival time is kept to be equal to the inter-departure time from that node (and furthermore at the beginning of the execution it coincides with the service time of the node too).

The algorithm evolves as follows:

1. The inputs are a directed graph $G = (V, E)$ and one of its topological ordering S (represented as an array of $|V|$ nodes);

2. The algorithm performs the graph traversal by visiting each node following the ordering S. For each explored node its inter-arrival time and its actual utilization factor are determined. Therefore we are able to identify if the node is a bottleneck or not;

3. If the currently explored node is not a bottleneck ($\rho \leq 1$), its inter-departure time equals its inter-arrival time and the graph traversal continues with the next node in the topological ordering S;

4. If the explored node is a bottleneck ($\rho > 1$), its inter-departure time coincides with its service time. At this point the algorithm needs to update the inter-departure times of the nodes already visited in the graph ordering;

5. The algorithm ends when all the nodes have been explored and bottlenecks are not discovered anymore (i.e. all nodes have an utilization factor less or equals to 1).

The fourth point is the most critical one. An elegant approach can be formulated for input acyclic graphs in which ***there is exactly one source node***. In this case we are able to define an efficient algorithm whose correctness can be proved by introducing the following invariant property:

Invariant 2.7. *When the i-th node in the input topological ordering S is visited, all the previously explored nodes (from the first one to the $(i-1)$-th of the ordering) will have an utilization factor less or at most equals to* 1.

The invariant is satisfied at the beginning of the execution. Every topological ordering of a single source graph starts with the source node which is the first vertex. As stated before, for the source node its inter-arrival time is initialized to its service time, thus its utilization factor is initially equal to 1. If, during the graph traversal, no bottleneck node is discovered, the algorithm will end when the last node is visited. In this case for each node its inter-departure time equals its inter-arrival time and the graph analysis is trivially completed.

On the other hand let us suppose that when the i-th node of ordering is visited, its utilization factor is greater than 1. This situation is depicted in Figure 10. Let us consider the currently discovered bottleneck the node S_b at

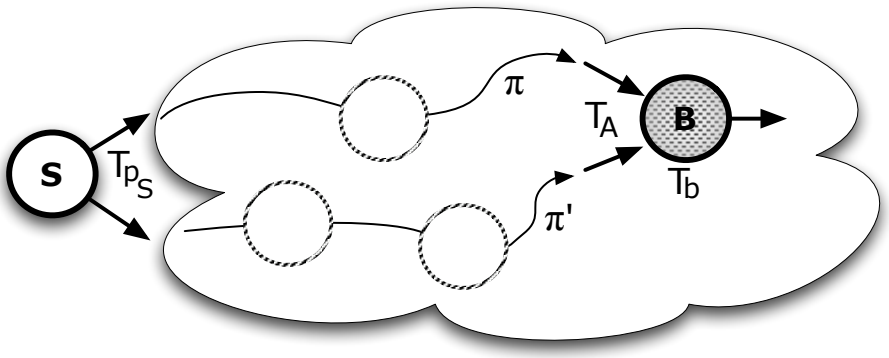


Figure 10: Bottleneck discovery.

position i of the topological ordering. Its service time is T_b which is greater than its actual calculated inter-arrival time T_A (i.e. $\rho_b > 1$). Since, by the invariant, every previous node in the ordering has already been visited and its utilization factor is less (or equal) to 1, the inter-arrival time to S_b can be expressed in function of the actual inter-departure time T_{p_S} from the unique source node of the graph. To this end we take the set $\mathscr{P}(S_b)$ of all the paths in the graph starting from the source node and ending to the current bottleneck node S_b. A path π is an ordered sequence of edges such that the origin of each is equal to the destination of its predecessor edge. E.g:

$$\pi = \Big\langle (N_1, p_1, N_2), (N_2, p_2, N_3), \ldots, (N_{k-1}, p_{k-1}, N_k) \Big\rangle$$

Where a directed edge is represented as a triple $e = (N, p, N^{'})$ where the first and the third element are the two end-point vertices of the edge and the second element is the probability that the first node transmits to the second one. For brevity we indicate with $e.p$ the probability corresponding to the edge e. By invariant all the nodes preceding S_b in the ordering have $\rho \leq 1$, thus we can determine the inter-arrival time to S_b by taking all the paths starting from the source node and ending to S_b, and iteratively applying Proposition 2.3 to calculate the inter-arrival time.

$$T_A = \left(\sum_{\forall \pi \in \mathscr{P}(S_b)} \left(\frac{\prod\limits_{\forall e \in \pi} e.p}{T_{p_S}} \right) \right)^{-1} \tag{6}$$

As we have seen the presence of the new discovered bottleneck node S_b influences the inter-departure times of all the previously explored nodes: i.e. they must be properly corrected. In particular after these corrections, the new inter-arrival time $T_A^{'}$ to S_b must be equal to its service time T_b (see Proposition 2.1). Moreover after the corrections the utilization factors of all the previously explored nodes will decrease, since their inter-arrival times are higher than the ones before the discovery of the bottleneck S_b. Thus, similarly to the previous case, we can express the new inter-arrival time to S_b in function of the corrected inter-departure time from the source node $T_{p_s}^{'}$:

$$T_A^{'} = \left(\sum_{\forall \pi \in \mathscr{P}(S_b)} \left(\frac{\prod\limits_{\forall e \in \pi} e.p}{T_{p_S}^{'}} \right) \right)^{-1} = T_b \tag{7}$$

In order to understand how we can correct the inter-departure time from the source, we can express the following relation: $T_{p_s}^{'} = T_{p_s} \cdot \alpha$ where α is a multiplicative factor greater than 1 (since the corrected inter-departure time needs to be greater than the previous one). At this point we can study how α can be determined:

$$\left(\sum_{\forall \pi \in \mathscr{P}(S_b)} \left(\frac{\prod\limits_{\forall e \in \pi} e.p}{\alpha\, T_{p_S}} \right) \right)^{-1} = \frac{\alpha\, T_{p_S}}{\sum\limits_{\forall \pi \in \mathscr{P}(S_b)} \left(\prod\limits_{\forall e \in \pi} e.p \right)} = T_b$$

From which we obtain the right expression for the coefficient α:

$$\frac{T_{pS}}{\sum\limits_{\forall \pi \in \mathscr{P}(S_b)} \left(\prod\limits_{\forall e \in \pi} e.p \right)} = \frac{T_b}{\alpha}$$

We can observe that the first element of the equation is the original inter-arrival time T_A, thus we can write:

$$T_A = \frac{T_b}{\alpha} \Longrightarrow \alpha = \frac{T_b}{T_A} = \rho_b$$

Therefore, when a new bottleneck node is discovered, we can correct the inter-departure time from the source node by multiplying the old inter-departure time by the utilization factor of the bottleneck node.

Proposition 2.8 (Invariant preservation)**.** *During the algorithm execution, if the i-th node of the topological ordering is a bottleneck ($\rho_i > 1$), we need to correct the inter-departure time from the source by multiplying this value by the utilization factor ρ_i. Then the algorithm is re-started from the beginning and, this time, when the i-th node is reached its utilization factor will be equal to* 1 *and all the previous nodes will continue to have an utilization factor less than* 1*.*

Proof. This proposition proves the correctness of the algorithm. Multiplying the inter-departure time of the source by the utilization factor of the discovered bottleneck is the only way to achieve a new corrected inter-arrival time $T_A^{'}$ to S_b equals to its service time T_b. Since $\rho_b > 1$ this means that the corrected inter-departure time $T_{p_s}^{'}$ is greater than the original one T_{p_s}, and thus the nodes preceding S_b in the ordering will continue to have an utilization factor less than 1. □

The algorithm 2 presents an automatic procedure for single source acyclic graph analysis. The algorithm proceeds in the following fashion. All the nodes are visited according to an input topological ordering. For each node is calculated its inter-arrival time by accessing its IN neighbor list (row 4). After that the utilization factor of the node is determined (row 5) and the bottleneck and non-bottleneck cases are examined. The most simply situation is when no bottleneck is discovered: in this case (from row 9 to 11) the inter-departure time of the current node is equal to the calculated inter-arrival time. Otherwise, if a bottleneck is discovered (from row 6 to 8), the inter-departure time of the source (first node in the ordering) is corrected as we have said and the visit re-starts from the first node.

Proposition 2.9 (Time complexity of Steady-State analysis)**.** *At the worst case the time complexity of steady-state analysis is $O(|V|^2)$ for sparse graphs and $O(|V|^3)$ for dense graphs.*

Algorithm 2: Steady-State Analysis(G, S)

Data: a single-source acyclic graph $G = (V, E)$ and one of its topological ordering S.

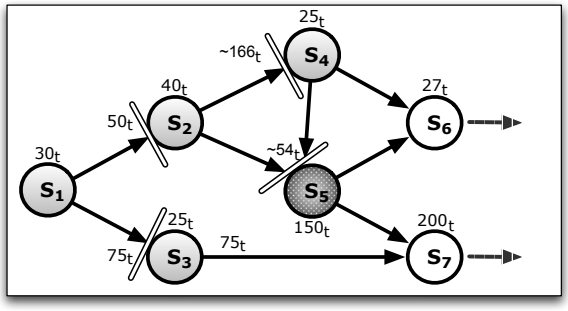
Result: at the end of the execution the attribute T_p of each node corresponds to its inter-departure time at steady-state.

```
1  begin
2      i ← 1;
3      while i ≤ |V| do
4          S[i].T_A = ( Σ_{(u,p)∈S[i].IN} p / u.T_p )^{-1};
5          S[i].ρ = S[i].T_S / S[i].T_A;
6          if S[i].ρ > 1 then bottleneck case
7              S[1].T_p = S[1].T_p × S[i].ρ;
8              i ← 1;
9          else not bottleneck case
10             S[i].T_p = S[i].T_A;
11             i ← i + 1;
12
13 end
```

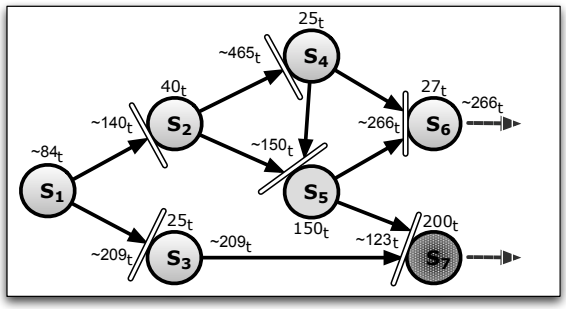
Proof. The cost in terms of time complexity of a graph traversal (without any restart) is $O(|V| + |E|)$, since for each node its list IN is visited once (see row 4). The traversal of the graph needs to be re-started whenever a bottleneck node is discovered. Let us consider B the number of bottleneck nodes that are discovered during the algorithm execution, where $0 \leq B \leq |V|$. For each bottleneck node the source inter-departure time is corrected and the traversal re-starts from the beginning. Thus the complexity of Steady-State analysis is $O\left(B \cdot (|V| + |E|)\right)$ where at the worst case $B = |V|$ (i.e. whenever a node is explored for the first time it results a bottleneck). Therefore for sparse graphs (where $|E| = O(|V|)$) the time complexity is $O(|V|^2)$ whereas for dense graphs (where $|E| = O(|V|^2)$) is $O(|V|^3)$. We can also notice that if no bottleneck node is never discovered (i.e. $B = 0$), the time complexity of the algorithm is the same of a simple graph traversal. □

For completeness we can observe that the algorithm needs as input the acyclic graph G but also one of its topological ordering S. As it is known the time complexity for finding a topological ordering is the same of a DFS (*depth-first search*) traversal[7] of the graph, i.e. $O(|V| + |E|)$. Thus the cost of Algorithm 2 certainly dominates the overall time complexity for the steady-state analysis.

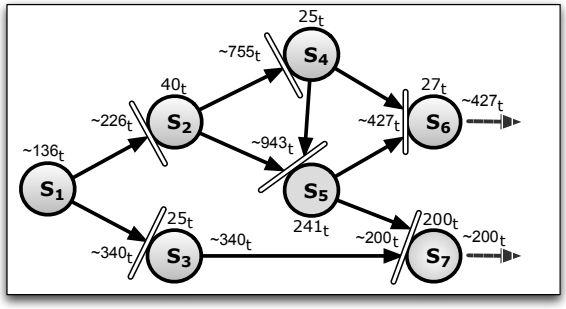
Example. In Figure 11 it is provided an example of steady-state analysis of



(a) The graph traversal starts from the node S_1. It is the unique source so its inter-arrival time is initially equal to its service time. Next, node S_2 is explored: the node is not a bottleneck since its inter-arrival time ($50t$) is greater than its service time. The same thing happens for nodes S_3 (with inter-arrival time $75t$) and for S_4 with inter-arrival time $\sim 166t$. When S_5 is discovered, its is a **bottleneck**: its inter-arrival time $\sim 54t$ is less than its service time $150t$ and its utilization factor is $\rho_5 =\sim 2.79$. Therefore we update the inter-departure time of the source node that passes from $30t$ to $30t \cdot \rho_5 =\sim 84t$.



(b) At this point the graph traversal re-starts from node 1. When S_5 is reached, it is not a bottleneck anymore (i.e. $\rho_5 = 1$). Now the node S_6 is explored and it is not a bottleneck (its inter-arrival time is $\sim 266t$). Then the last node S_7 is visited and its inter-arrival time $\sim 123t$ is less than its service time $200t$. So this node is a **bottleneck** and its utilization factor is $\rho_7 =\sim 1.62$. Hence we update the inter-departure time of the source node that passes from $\sim 84t$ to $84t \cdot \rho_7 =\sim 136t$.



(c) The graph traversal re-starts from node 1. At this point no bottleneck node is identified: i.e. for every node in the graph its utilization factor is now lower or equal to 1. The algorithm terminates correctly providing the steady-state behavior of the acyclic graph.

Figure 11: An example of steady-state analysis of an acyclic graph.

Node	Service time	Inter-departure time	Utilization factor
S_1	$30t$	$\sim 136t$	~ 0.22
S_2	$40t$	$\sim 226t$	~ 0.174
S_3	$25t$	$\sim 340t$	~ 0.0736
S_4	$25t$	$\sim 755t$	~ 0.0326
S_5	$150t$	$\sim 241t$	~ 0.62
S_6	$27t$	$\sim 427t$	~ 0.063
S_7	$200t$	$200t$	1

Table 1: Results of steady-state analysis.

an acyclic graph. Let us consider the input graph depicted in Figure 8 labeled with the service times for each of its nodes and the routing probabilities for each edge. Figure 11 depicts the different phases of the execution following the topological ordering shown in Figure 9. Gray nodes represent explored vertices, white nodes correspond to vertices that are still to be explored whereas a point-based black node is the currently discovered bottleneck. The final results are also shown in Table 1.

2.4.1 Impact of the randomness on the Performance Analysis of acyclic computation graphs

We conclude the analysis of acyclic computation graphs by providing a brief discussion about the impact of the randomness on the accuracy of the results achieved with the steady-state analysis algorithm. In the previous sections we have assumed deterministic service times for each node of the graph: i.e. for each node its theoretical service time is a fixed constant value. In this case the accuracy of the algorithm has been evaluated on several example graphs through a Queueing Network simulator (*Java Modelling Tool*[8]). The simulation results demonstrate an absolute precision of the algorithm which is able to quantify the steady-state behavior for each node.

Things can become different if we introduce randomness, i.e. if we suppose stochastic random variables that model the service time of each node. Here, the service time assumes stochastic values following a probability density function with a known average value. A valuable modeling for our purposes is assuming that the service processes of each node follow an *exponential distribution*. With this assumption each node in the network is modeled as a M/M/1 queue (instead of D/D/1 queues as in the previous discussion). As it is known the main property of this distribution is *memoryless*: if we assume that each service request (task) is independent from the others, the service time for completing a task does not depend on the service times spent for the previous tasks calculated by the node. This property is simple enough for solving the steady-state analysis analytically in closed form. Moreover, for stream-based parallel computations, the independence among tasks is a reasonable approximation of many real cases.

All the previously introduced propositions for pipeline graphs and for multiple-

destination and multiple-source queues still remain valid assuming that, instead of having service times that assume constant values, we have a mean value for each service time of a node in the graph. W.r.t the deterministic case, in the exponential case the size of each queue plays an important role for attenuating the impact of the randomness. In fact we expect that the results of the steady-state analysis approximates well the behavior of a M/M/1 network if, for each node, the queue size is large enough (but still bounded). For this reason we have simulated the behavior of the network shown in Figure 8, in which the service time values are assumed to be the average values of corresponding exponential random variables. The simulations have been performed by using Java Modelling Tools and by varying the size of each queue. Tests for 50, 25, 12, 6, 3, 2 and 1 buffer positions are depicted in Table 2.

Node	Er %.(25)	Er %.(12)	Er %.(6)	Er %.(3)	Er %.(2)	Er %.(1)
S_1	.5014	.2314	2.4086	7.1857	10.2389	18.0247
S_2	.7897	.7922	2.4254	7.1375	10.3436	18.3095
S_3	.6359	.3814	2.8383	6.9518	10.1564	17.6470
S_4	.3411	.1071	1.8848	6.8147	9.46308	18.2592
S_5	.7325	.2265	2.2014	6.6678	9.77189	17.8800
S_6	.1996	.3984	2.6694	6.8376	9.64912	17.3708
S_7	.0820	.0820	.9448	6.4509	9.43553	17.6844

Table 2: Accuracy of the steady-state analysis in function of the queue size of each node.

In the table are reported for each queue node, the percentage errors of the simulation inter-departure times w.r.t the ones obtained by the algorithm execution. As we can expect if we decrease the size of each queue the errors increase. For this example we can observe that for large enough queue size (up to 12 buffer positions), the errors are less than 1% for each node. For very limited queue size (i.e. 3, 2 and 1 positions), the errors are less than 8, 11 and 19% for each node. Therefore we can conclude that steady-state analysis gives quite precise results w.r.t the simulations for queue size large enough.

3 Cyclic Computation Graphs: analytical treatment

In the previous section we saw that Queueing Networks are a sufficiently powerful methodology in order to determine the performance modeling of acyclic computation graphs. The analytical treatment of Queueing Systems, in terms of probability distributions of inter-arrival and service times, is mainly necessary for cyclic graph computations exhibiting a request-reply behavior.

Let us consider a system in which client modules $C_1, C_2, \ldots, C_N$ transmit to a server module S a set of requests and need to wait for an explicit reply

in order to continue their elaboration. An example of these graphs is shown in Figure 12. As we can see this interaction yields to cycles in the communication pattern. The parameters of interest in evaluating the server performance are:

- The *average queue length*, L_q: the average number of client requests in the waiting queue;
- The *average request number* in the system, N_q: with respect to the queue length, it includes the number of currently served requests;
- The *average waiting time* in queue, W_q: the average time spent by a request in the waiting queue of the server;
- The *average response time*, R_q: with respect to W_q includes the time spent on the currently served request(s). It is also called response time, and consists in the average time needed from a client to receive the result for the requested service.

These parameters are related each other in several ways. One of the most useful results is the ***Little's law***[9]:

$$L_q = \frac{W_q}{T_A} \qquad N_q = \frac{R_q}{T_A} \tag{8}$$

This law is a fundamental long-term relationship which ties together the concept of waiting and response time and the concept of population size of the queue. T_A is the aggregate inter-arrival time to the server S, so its inverse represents the average frequency of arrivals. Other important relations are the following:

$$\begin{aligned} N_q &= L_q + \rho_s \\ R_q &= W_q + L_s \end{aligned} \tag{9}$$

The former holds since the average number of requests currently in the service phase is equal to the utilization factor of the server $\rho_s = T_S/T_A$, where T_S is the average service time of S. The latter means that we add, to the average time spent in the waiting queue, the average computation latency L_s of a service phase. This aspect is very important especially if the server module is internally parallel. As we know, several structured parallelizations of the server could be adopted, with different impacts on the average service time and on the average computation latency. Therefore in the response time expression is of great importance to consider the computation latency per request, which may be different from the average service time of the server.

The graph depicted in Figure 12 needs to be properly modeled from the performance viewpoint in order to evaluate a meaningful parameter: i.e. the average response time of the server. For this reason one possible approach consists in modeling this graph as a closed queueing network. Two important considerations emerge from the semantics of the request-reply behavior:

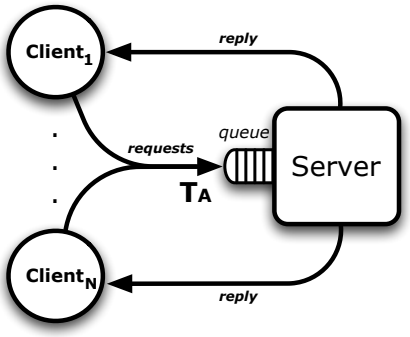


Figure 12: Client-Server computation modeled by a closed queueing network.

- each client generates the next request only when the result of the previous one of the same client has been received. This means that from a modelistic point of view it is equivalent to consider a finite population of tasks, as many as the global number of clients N, that circulate continually and never leave the network;
- the network has a *self-stabilizing behavior*: i.e. a temporary increase in the inter-arrival time has the effect of a decrease in the server response time that tends to lower the inter-arrival time itself. In such kind of systems we cannot speak of bottlenecks, or at least with the same meaning exposed for acyclic graphs since ρ_s is always smaller than one. Anyway the stable behavior of the server queue makes it possible the analytical studying of the response time distribution and its average value.

Closed queueing networks are a quite complex argument of Queueing Theory that requires further discussions and a more exhaustive description. In the following part we will provide the basic formulation inherited from[1] for modeling the performance of cyclic computation graphs of parallel modules, interacting by exchanging request-reply pairs. Notable cases of this interaction pattern are client-server parallel applications.

3.1 Performance modeling of Client-Server parallel computations

Let us assume for simplicity that all clients have an identical behavior. Let T_C be the steady-state inter-departure time from each client, T_G its service time and T_S the server service time. We need a performance model which is able to quantify the effective steady-state inter-departure time from each client module. This inter-departure time is certainly greater than the theoretical service time: the request-reply behavior imposes that, after the transmission of a request (which are generated by each client with a theoretical frequency equals to $1/T_G$),

the client waits for an explicit reply from the server. Only after the reception of the reply a further request can be produced by the client. Therefore we expect that T_C is increased w.r.t T_G by the response-time of the server (i.e. time spent in the waiting queue and the time for completing the service). The following system of equations models the performance of this class of computations:

$$\begin{cases} T_C = T_G + R_q \\ R_q = W_q(\rho_s, T_S, T_A) + L_s \\ \rho_s = \dfrac{T_S}{T_A} \\ T_A = \dfrac{T_C}{N} \end{cases} \tag{10}$$

where the last equality is derived by applying the expression 5 in which N indicates the number of clients. The solution of the system is subject to constraint $\rho_s < 1$, as discussed in the previous section. Proper waiting time (W_q) expressions, of second or higher order in ρ_s, can be derived by applying well-known results in Queueing Theory[9]. These expressions depend on the particular probability distributions of the server service time and of the inter-arrival time from clients. Formulas can be derived for different cases:

M/M/1 queue : exponential distribution of the inter-arrival time from clients and exponential distribution of the server service time. In this case the average waiting time in queue is given by:

$$W_q = \frac{T_S^2}{T_A - T_S} \tag{11}$$

M/G/1 queue : exponential distribution of the inter-arrival time from clients and general distribution for the server service time. The average waiting time is given by the Pollaczek-Khinchnine formula:

$$W_q = \frac{\sigma_s + T_S^2}{2\,T_A - 2\,T_S} \tag{12}$$

We can notice that σ_s in the M/G/1 expression indicates the service time variance. A notable case is when the service time distribution is deterministic (the server provides a constant service time). In this case we speak about the M/D/1 queue, and the average waiting time can be obtained from (12) with a zero variance, i.e. $\sigma_s = 0$.

As stated in[1], by using the proper W_q expression the previous system of equations admits one and only one real, positive solution satisfying $\rho_s < 1$, which is an approximation of the average server response time.

The previous formulas, formally derived for infinite queues, are valid also for finite FIFO queues with good approximation, especially for relatively large values of the physical queue positions. When this approximation is not sufficient, exact formulas also exist for the finite queue[6].

4 Conclusion

This report provides performance modeling tools for parallel applications based on basic results on Queueing Theory and Queueing Networks. Our standpoint considers complex graph-based applications in which each module may be internally parallel. Two classes of parallelism have been identified: (i) intra-module parallelism, which is expressed by well-known parallelization schemes as task-farm and data-parallel ones; (ii) modules can cooperate (inter-module parallelism) in complex general computation graph structures. In this report we have described a unified performance modeling approach: starting from the theoretical service times of notable parallelism schemes, that can be encapsulated inside parallel application modules, we have provided an automatic procedure for calculating the steady-state performance behavior of a general class of computation graphs.

References

[1] M. Vanneschi, "High performance computing systems and enabling platforms." Course Notes of the Master Program in Computer Science and Networking, University of Pisa, Jan. 2011.

[2] E. D. Lazowska, J. Zahorjan, G. S. Graham, and K. C. Sevcik, *Quantitative system performance: computer system analysis using queueing network models*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1984.

[3] J. Mincer-Daszkiewicz, "Program i/o behavior: Models and their applications," *Journal of Systems and Software*, vol. 14, no. 1, pp. 51–62, 1991.

[4] L. Kerbache and J. M. Smith, "Queueing networks and the topological design of supply chain systems," *International Journal of Production Economics*, vol. 91, no. 3, pp. 251–272, 2004.

[5] M. Hassan, R. Egudo, and J. Breen, "A closed queueing network model for retransmission based transport protocols over cell-switching networks," in *Communications, Computers, and Signal Processing, 1995. Proceedings. IEEE Pacific Rim Conference on*, pp. 86 –89, May 1995.

[6] H. G. Perros, *Queueing networks with blocking*. New York, NY, USA: Oxford University Press, Inc., 1994.

[7] *Introduction to algorithms*. Cambridge, MA, USA: MIT Press, 2nd ed., 2001.

[8] D. of electronics and computer science polytechnic of Milan, "Java Modelling Tools." `http://jmt.sourceforge.net/`, 2011.

[9] L. Kleinrock, *Theory, Volume 1, Queueing Systems*. Wiley-Interscience, 1975.