Scheduling Tasks with Precedence Constraints on Multiple Servers

Ramtin Pedarsani, Jean Walrand and Yuan Zhong

***Abstract*— We consider the problem of scheduling jobs which are modeled by directed acyclic graphs (DAG). In such graphs, nodes represent tasks of a job and edges represent precedence constraints in processing these tasks. The DAG scheduling problem, also known as scheduling in fork-join processing networks, is motivated by examples such as job scheduling in data centers and cloud computing, patient flow scheduling in health systems and many other applications. We consider a flexible system, in which servers may process different, possibly overlapping, sets of task types. In this paper, we first discuss the difficulties in designing provably efficient policies for DAG scheduling, which arise due to interactions between the flexibility of the processing environment and the precedence constraints in the system. A major difficulty is the classical synchronization issue, which is further complicated in the presence of system flexibility. Then, we propose two queueing networks to model the scheduling problem that overcome this difficulty. These are virtual queues that enable us to design provably efficient scheduling policies. We show that the well-known Max-Weight policy for these queueing networks is throughput-optimal. Finally, to compare the delay performance of the two queueing networks, we consider a simplified model in which tasks and servers are identical. We characterize their delay performances under a simple first-come-first-serve policy, via a novel coupling argument.**

I. Introduction

Many service and processing systems consist of multiple interconnected tasks served by a set of servers. Scheduling algorithms attempt to maximize the throughput of these systems and minimize their response times. Good algorithms can result in substantial savings in investment and energy costs. Designing such algorithms is complicated because of the presence of both sequential and parallel demand tasks, and the *flexibility* of servers to process different task types as well as the flexibility of tasks that can be processed by different servers.

For instance, in a data center, large-scale computing jobs are divided into smaller tasks, many of which can be processed in parallel. Because of the flexibility of tasks and servers, there are often many options to assign these tasks to different servers. Furthermore, there are also precedence constraints among these tasks that complicate efficient load balancing and scheduling. Another important example arises from patient flow in hospitals. Efficient scheduling schemes can significantly improve patient experience and care quality [1], [18]. Patient flow is often complicated with intricate dependencies among different processing activities (see e.g., Figure 8 in [1]). Some of these activities may be processed in parallel, while others must be processed sequentially. Hospital staff can also be cross-trained to handle multiple types of tasks [1].

Motivated by these applications, we consider a DAG scheduling problem in a flexible network, that faithfully captures the complicated dependencies among workflows in these systems as well as the heterogeneity of the processing environment. The model that we consider is similar to the one proposed in [27] with a major difference. We consider the more realistic case where servers are non-cooperative in processing tasks of a job.

When jobs coming to the system have distinct identities, the so-called synchronization issue arises – parallel tasks of the same job need to be *synchronized* for subsequent tasks to be processed. A naive scheduling policy may need to keep track of all the identities of jobs in the system – a potentially unbounded number – which may be undesirable in practice (see Section II for details). Furthermore, even after keeping tracking of all identities of jobs in the system, it is not clear how to schedule them.

To resolve the synchronization problem, we propose two different solutions, both of which involve the construction of virtual queues. We propose Max-Weight type scheduling policies for these networks, and show that both policies are throughput-optimal. Moreover, we demonstrate that one of the proposed has much smaller scheduling complexity, but it can potentially suffer from larger delay due to additional precedence constraints that are enforced to the system. See Section III for details.

Towards comparing the delay performance of the two virtual queueing networks, we first simplify the network model to make the problem tractable. Second, we propose a throughput-optimal first-come-first-serve scheduling policy, and compare its delay performance for two extreme cases of a serial system versus a parallel system. See Section IV for details.

A. Related Works

DAG scheduling has a wide range of applications, such as parallel processing (see e.g., [22]) and flexible manufacturing (see e.g., [25], [26]). More recently, it also has applications in healthcare systems [1], [18], where the DAG structure is used to model patient flow in hospitals [2], and data centers, where it is used to model data-intensive computations such as MapReduce [12], Dryad [15], Spark [31], etc.

The analytical research considers either *static* or *dynamic* schedules. In a static (a.k.a. offline) scheduling problem, the

R. Pedarsani and J. Walrand are with Department of Electrical Engineering and Computer Science, University of California, Berkeley, CA 94720 USA `email: ramtin@berkeley.edu, walrand@berkeley.edu`

Yuan Zhong is with the Department of Industrial Engineering and Operations Research, Columbia University, New York, NY USA `email: yz2561@columbia.edu`

jobs – modeled by DAGs – are present at the beginning and the problem is to allocate tasks to processors to minimize the expected completion time. In most cases the problem is computationally hard, hence a wide range of heuristics have been proposed. See [22] and the references therein for a review. Static DAG scheduling has a wide range of applications, including reliability analysis of computer systems [28] and performance analysis of PERT (Program Evaluation Review Technique) networks [13].

In a dynamic problem, DAGs arrive to the system and get scheduled dynamically over time. The main difference between these models and ours is that in these models, tasks are assigned to dedicated servers upon arrival, whereas we allow tasks to be *flexible*, and schedulers have the freedom to assign tasks to often more than one server.

System models for the study of the dynamic problems have been called "fork-join networks" [7], [6], [5], [4], [24], [20] in the literature. The most basic question regarding these networks concerns the necessary and sufficient conditions for stability, that is, for the existence of the steady-state probability distribution of the underlying Markov process. Works [7], [6], [5], [8], [3] have established stability conditions for a class of these networks. When the arrival process and the service time sequence form mutually independent renewal processes, the stability conditions reduce to the natural deterministic conditions based on arrival and service rates.

Given the stability results, the next natural question is to compute the steady-state expected completion times of DAGs. Few analytical results are available, except for the simplest models (see e.g., [16], [17], [23]). Performance bounds on the stationary expected job completion time have been derived (see e.g., [7], [6], [5], [24], [3]), but for most models, the tightness of these bounds is not known.

An approach that has proved effective in revealing structural properties of complex queueing networks is so-called "heavy-traffic analysis", where the system state is scaled appropriately and system utilization approaches 1. Works in this direction include [25], [26], [30].

In a related class of models, tasks arrive to an infinite-server system over time, and have probabilistic precedence constraints with tasks waiting to be served in the system [29], [3], [8]. This class of models is also motivated by applications in parallel processing, but are somewhat different from the models described above. Stability conditions of these systems have been established in various cases [29], [3], [8].

II. Network Model

We consider a general processing network, in which jobs are modeled as directed acyclic graphs (DAG). Each job consists of several smaller tasks, among which there are precedence constraints. Each node of the DAG represents one task (or one task type), and each edge represents a precedence constraint. We consider M types of jobs, where each type is specified by one DAG structure.

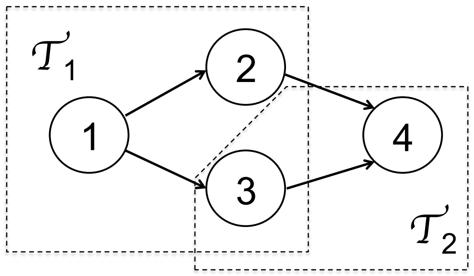

Fig. 1: A simple DAG

Let $\mathcal{G}_m = (\mathcal{V}_m, \mathcal{E}_m)$ be the DAG corresponding to the job of type m, $1 \le m \le M$, where $\mathcal{V}_m$ denotes the set of nodes, and $\mathcal{E}_m$ represents the set of edges of the DAG. We assume that each DAG, $\mathcal{G}_m$, is connected, so that there is an undirected path between any two nodes of $\mathcal{G}_m$. Let the number of tasks of a type-m job be K_m, i.e. $|\mathcal{V}_m| = K_m$, and the total number of task types in the network be K, so that $\sum_{m=1}^{M} K_m = K$. We index the task types in the network by k, $1 \le k \le K$, starting from job type 1 to M. Therefore, task type k belongs to job type $m(k)$ if

$$\sum_{m'=1}^{m(k)-1} K_{m'} < k \le \sum_{m'=1}^{m(k)} K_{m'}.$$

With a slight abuse of notation, we sometimes also use k to denote a task of type k, when the context is clear.

We call task k' a parent of task k if they belong to the same DAG, and $(k', k) \in \mathcal{E}_m$. We now formally define the precedence constraints in the network. Let $\mathcal{P}_k$ denote the set of parents of k. In order to process k, the processing of all the parents of k, $k' \in \mathcal{P}_k$, should be completed. We call task k' a descendant of task k if they belong to the same DAG, and there is a directed path from k to k' in that DAG.

There are J servers in the processing network. Servers are *flexible* in the sense that each server can serve a non-empty set of tasks. Similarly, tasks are also flexible, so that each task can be served by a non-empty set of servers. In other words, servers can have overlap of capabilities in processing one task. For each j, we define $\mathcal{T}_j$ to be the set of tasks that server j is capable of serving. Let $T_j = |\mathcal{T}_j|$, and let S_k be the number of servers that can process task k. Without loss of generality, we assume that $S_k \ge 1$, $T_j \ge 1$ for all k, j, so that each server can serve at least one task, and each task can be served by at least one server.

Example. Figure 1 illustrates the DAG of one job that consists of four tasks $\{1, 2, 3, 4\}$. There are two servers 1 and 2. Server 1 can process the tasks in the set $\mathcal{T}_1 = \{1, 2, 3\}$ and server 2 can process the tasks in the set $\mathcal{T}_2 = \{3, 4\}$. When task 1 is completed, it produces one task 2 and one task 3 both of which have to be completed before task 4 can use the results of these tasks to complete the job.

We consider the system in discrete time. We assume that the arrival process of jobs of type m is a Bernoulli process with rate λ_m, $0 < \lambda_m < 1$, that is, at each time slot a new job of type m arrives to the system with probability λ_m independently over time. We assume that the service times are geometrically distributed and independent of everything else. When server j processes task k, the service completion time has mean μ_{kj}^{-1}. Thus, μ_{kj} can be interpreted as the service rate of task k processed by server j.

A. Job Identity

In our model, jobs and tasks have distinct identities. This is mainly motivated by data center applications as well as health systems. For instance, it is important not to mix up blood samples of different patients in hospital, and to put pictures on the correct webpage in a data center setting. However, in a car manufacturing line, the wheel of the car for example, does not have an identity and can be installed on any car.

An approach to avoid mixing tasks of different jobs is to keep track of job identities of all tasks present in the system at all times. However, this requires potentially unbounded memory from the scheduler, since a task could belong to any one of the previously arrived job. Furthermore, it is not clear how to best make use of such information for designing policies.

In [27], it is assumed that the servers can cooperate on the same task, adding their service capacities. Then, a queueing network model is considered in which each queue corresponds to an edge of the DAG. In this case, there is no synchronization problem since the order of tasks in each queue will be the same as the order of their corresponding jobs arriving to the system; this is because tasks are processed in a first-in-first-out (FIFO) manner. However, in many applications the assumption that servers are cooperative is not practical. For example, two data center servers are usually not able to add their capacities to process a particular task.

Here we describe the queueing network model in [27], and why under the assumption of non-cooperative servers, there is a synchronization problem. Consider the example of Figure 1. The queueing network model in [27] that corresponds to this example maintains one queue for each edge of the graph, and one queue for each root node. This network is shown in Figure 2. The queues are labeled based on the corresponding edge, and the first queue is labeled by $(0,1)$ by convention. In this network, when task 1 of job a from queue $(0,1)$ is processed, tasks 2 and 3 of job a are sent to queues $(1,2)$ and $(1,3)$, respectively. When tasks in queues $(1,2)$ and $(1,3)$ are processed, their results are sent to queues $(2,4)$ and $(3,4)$, respectively. Finally, to process task 4 of job a, one task belonging to job a from queue $(2,4)$ and one task belonging to job a from $(3,4)$ are gathered and processed to finish processing job a. We call this last step as a "join" process which requires synchronization. Note that since tasks are identity-aware, to complete processing task 4, it is not possible to merge any two tasks (let's say of jobs a and b) from queues $(2,4)$ and $(3,4)$. When servers are non-cooperative a synchronization problem can occur as follows. Suppose that job a arrives shortly before job b to the system. Assume that server 1 is much faster than server 2.

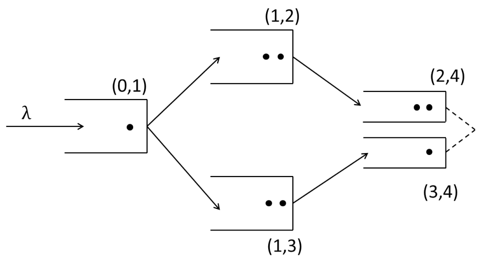

Fig. 2: The queueing network proposed in [27] for the DAG of Figure 1.

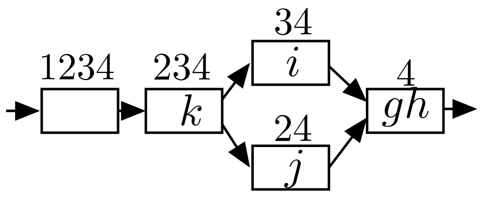

Fig. 3: Queueing Network of Virtual Queues for the DAG in Figure 1

The following sequence of events can happen. First, server 1 processes task 1 of job a in queue $(0,1)$, and then it processes task 1 of job b in queue $(0,1)$ while server 2 starts processing task 3 of job a in queue $(1,3)$. Server 1 finishes processing task 2 of job a and the result is sent to queue $(2,4)$. Then server 1 starts processing task 3 of job b and sends the result to queue $(3,4)$ before task 3 of job a is fully processed by server 2. Now the head of the line task of queue $(2,4)$ belongs to job a and the head of the line task of queue $(3,4)$ belongs to job b. Thus, an identity-oblivious policy may wrongly schedule processing task 4 by merging the results of jobs a and b from queues $(2,4)$ and $(3,4)$.

In this paper, we consider the more realistic scenario that servers are non-cooperative. In the following two subsections, we propose two solutions to keep the memory requirement bounded by considering *virtual* queues for the system in the non-cooperative case.

B. Virtual Queues for Stages of the Job

We define a stage of a job in the system as the set of tasks belonging to that job that are waiting to be processed. We associate a virtual queue with each possible stage of a job. This resolves the synchronization issue, since there is no processing activity which requires merging tasks of two or more virtual queues. The following example clarifies how the network of virtual queues is formed.

Example. Consider again the DAG shown in Figure 1. We consider 5 virtual queues corresponding to 5 possibles stages of the job which are: $\{1,2,3,4\}$, $\{2,3,4\}$, $\{2,4\}$, $\{3,4\}$, $\{4\}$. The queueing network of these virtual queues is illustrated in Figure 3.

Now we formally explain how one forms the network of virtual queues for M types of jobs. Consider M DAGs, $\mathcal{G}_m = (\mathcal{V}_m, \mathcal{E}_m)$, for $1 \leq m \leq M$. We construct M parallel

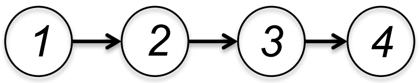

Fig. 4: Queueing Network of Virtual Queues for the serialized DAG

networks of virtual queues as follows. For each non-empty subset $\mathcal{S}_m$ of $\mathcal{V}_m$ consider a virtual queue if and only if for all $i \in \mathcal{S}_m$, all the descendants of i are also in $\mathcal{S}_m$. As an example, subset $\{1, 3\}$ does not correspond to any virtual queue for the DAG in Figure 1, since 2 and 4 are descendants of 1, but not in the subset. Let K' be the number of non-empty subsets that satisfy the mentioned condition. Then, we represent the state of M types of DAGs by K' virtual queues. Clearly, there will be no interaction among the virtual queues corresponding to different DAGs.

Remark 1: In general, the number of virtual queues corresponding to different stages of a job with K tasks can grow exponentially with K since each stage denotes a feasible subset of tasks that require processing. This can significantly increase the complexity of scheduling policies that try to maximize the throughput of the network. For practical purposes, it is important to find a queueing network that has low complexity in terms of number of virtual queues while it also resolves the synchronization problem.

C. Queueing Network based on Additional Precedence Constraints

In this subsection, we propose another network of virtual queues that ensure the synchronization of different tasks of one job type. The queueing network is formed by enforcing additional constraints such that the DAG of type m becomes K_m nodes in series. For instance, for the DAG of Figure 1, we assume that there is a further constraint that task 3 should proceed task 2. Then, the modified DAG will have 4 nodes in series as shown in Figure 4. The nice property of this "serialized" job is that there are only K_m stages of the job since the stage of each job can be uniquely determined by the next task that requires processing. Therefore, the network of virtual queues has only K queues which decreases substantially the complexity of scheduling policies for the network of virtual queues. However, it is not clear how these extra precedence constraints affect the performance of scheduling policies in terms of both throughput and delay. We provide some preliminary analysis of these effects in Section IV.

D. Capacity Region

In this subsection, we characterize the capacity region of the network, defined to be the set of all arrival rate vectors λ where there is a scheduling policy that *rate* stabilizes the network [9]. The network is rate stable if the long-run average arrival and departure rates at each queue are equal. Later we show that the capacity region of the two proposed virtual queueing networks is identical to the original DAG scheduling problem, which we discuss here. This is not surprising, since queueing in general does not decrease the capacity region.

For the original DAG scheduling problem, let p_{kj} be the fraction of capacity that server j allocates for processing available tasks of class k. Let $p = [p_{kj}]$ to be the *allocation vector*. If server j allocates all its capacity to different tasks, then $\sum_{k\in\mathcal{T}_j} p_{kj} = 1$. Thus, an allocation vector p is called *feasible* if

$$\sum_{k\in\mathcal{T}_j} p_{kj} \leq 1, \ \forall\ 1 \leq j \leq J. \tag{1}$$

We introduce a linear program (LP) that characterizes the capacity region of the system. The *nominal* traffic rate of all the tasks of job type m is λ_m. Let $\nu = [\nu_k] \in \mathbb{R}^K_+$ be the set of nominal traffic rate of the queues in the network. Then, $\nu_k = \lambda_m$ if $m(k) = m$, i.e., if $\sum_{m'=1}^{m-1} k_{m'} < k \leq \sum_{m'=1}^{m} k_{m'}$. The LP that characterizes the capacity region of the network makes sure that the total service capacity allocated to each node in the network is at least as large as the nominal traffic rate to that node. Thus the LP, known as the *static planning problem* [19], is defined as follows.

$$\begin{aligned} \text{Minimize} \quad & \rho & (2)\\ \text{subject to} \quad & \nu_k = \sum_{j:k\in\mathcal{T}_j} \mu_{kj}p_{kj}, \ \forall\ 1 \leq k \leq K & \\ & \rho \geq \sum_{k\in\mathcal{T}_j} p_{kj}, \qquad \forall\ 1 \leq j \leq J, & (3)\\ & p_{kj} = 0, \qquad \text{if } k \notin \mathcal{T}_j, & (4)\\ & p_{kj} \geq 0. & (5) \end{aligned}$$

Proposition 1: Let the optimal value of the LP be ρ^*. Then $\rho^* \leq 1$ is a necessary condition for stability of the system.

Proof: Let $\tilde{X}_k^n$ be the number of class k tasks in the system at time n. Consider the system in the fluid limit. (See [9] for more discussion on the stability of fluid models) The fluid level of class k tasks at time t is

$$X_k(t) = \lim_{r\to\infty} \frac{\tilde{X}_k^{\lfloor rt \rfloor}}{r}.$$

The dynamics of the fluid are as follows

$$X_k(t) = X_k(0) + \lambda_{m(k)}t - D_k(t),$$

where $\lambda_{m(k)}t$ is the total number of jobs of type m that have arrived to the system until time t, and $D_k(t)$ is the total number of class k tasks that have been processed up to time t in the fluid limit.

Suppose that $\rho^* > 1$. We show that if $X_k(0) = 0$ for all k, there exists t_0 and k such that $X_k(t_0) \geq \epsilon(t_0) > 0$, which implies that the system is weakly unstable [9]. In contrary, suppose that there exists a scheduling policy that under that policy for all $t \geq 0$ and all k, $X_k(t) = 0$. Pick a regular point[1] t_1. Then, for all k, $\dot{X}_k(t_1) = 0$. This implies that $\dot{D}_k(t_1) = \lambda_m = \nu_k$. On the other hand, at a regular point t_1,

[1]We define a point t to be regular if $X_k(t)$ is differentiable at t for all k.

$\dot{D}_k(t_1)$ is exactly the total service capacity allocated to class k tasks at t_1. This implies that there exists p_{kj} at time t_1 such that $\nu_k = \sum_{j:k\in\mathcal{T}_j} \mu_{kj} p_{kj}$ for all k and the allocation vector $[p_{kj}]$ is feasible, i.e. $\sum_{k\in\mathcal{T}_j} p_{kj} \leq 1$. This contradicts $\rho^* > 1$.

■

Later on we will see that $\rho^* \leq 1$ is also a sufficient condition of rate stability. Thus, the *capacity region* Λ of the network is the set of all $\lambda \in \mathbb{R}_+^M$ for which the corresponding optimal solution ρ^* to the LP satisfies $\rho^* \leq 1$. More formally,

$$\Lambda \triangleq \Big\{ \lambda \in \mathbb{R}_+^M : \exists\ p_{kj} \geq 0 \text{ such that } \sum_{k\in\mathcal{T}_j} p_{kj} \leq 1 \quad \forall\ j, \\ \text{and } \nu_k = \sum_{j:k\in\mathcal{T}_j} \mu_{kj} p_{kj} \quad \forall\ k \Big\}.$$

III. Throughput-Optimal Policy

In this section, we propose Max-weight scheduling policy for the network of virtual queues and show that it is throughput optimal. We mainly focus on the queueing network mentioned in Subsection II-B. Serializing tasks of different job types is a special case of these queueing networks, though it has substantially fewer number of queues. Therefore, the results provided in this section are also valid for the serialized queueing network mentioned in Subsection II-C.

Now we describe the dynamics of the virtual queues in the network. When server j works on task i in a virtual queue corresponding to subset $\mathcal{S}_m$, the result of the process is sent to the virtual queue corresponding to subset $\mathcal{S}_m\backslash\{i\}$, and the processing rate is μ_{ij}. We call the action of processing task i in virtual queue corresponding to $\mathcal{S}_m$ as a service activity. We denote the collection of different service activities in the network as $\mathcal{A}$. Let $A = |\mathcal{A}|$. Define the collection of activities that server j can perform as $\mathcal{A}_j$.

As an example, consider the DAG of Figure 1 and the corresponding queueing network of Figure 3. Note that $\mathcal{T}_1 = \{1,2,3\}$ and $\mathcal{T}_2 = \{3,4\}$. Then, there are 8 activities as follows.

1) Task 1 in virtual queue $\{1,2,3,4\}$ is served by server 1 with rate μ_{11}.
2) Task 2 in virtual queue $\{2,3,4\}$ is served by server 1 with rate μ_{21}.
3) Task 3 in virtual queue $\{2,3,4\}$ is served by server 1 with rate μ_{31}.
4) Task 3 in virtual queue $\{2,3,4\}$ is served by server 2 with rate μ_{32}.
5) Task 2 in virtual queue $\{2,4\}$ is served by server 1 with rate μ_{21}.
6) Task 3 in virtual queue $\{3,4\}$ is served by server 1 with rate μ_{31}.
7) Task 3 in virtual queue $\{3,4\}$ is served by server 2 with rate μ_{32}.
8) Task 4 in virtual queue $\{4\}$ is served by server 2 with rate μ_{42}.

Each activity imposes a departure rate from one queue and an arrival rate to another queue. For example, activity 1 leads to the length of virtual queue $\{1,2,3,4\}$ reducing with rate μ_{11}, and virtual queue $\{2,3,4\}$ increasing with rate μ_{11}, and so on. For the ease of notation, we index the virtual queues by k', $1 \leq k' \leq K'$. We define a drift matrix $D = [d_{k'a}] \in \mathbb{R}^{K'\times A}$, where $d_{k'a}$ is the rate that virtual queue k' changes if activity a is performed. The drift matrix for the above example is presented in Equation (6).

Define length K' arrival vector $e = [e_{k'}]$ such that $e_{k'} = \lambda_m$ if virtual queue k' corresponds to the first stage of m-th job type in which no tasks are yet processed, and $e_{k'} = 0$ otherwise. Now we introduce the LP for this virtual queueing network that characterizes the capacity region of this network. Let $k(a)$ be the task that is processed in activity a. Let $\mathcal{A}_j$ be the set of activities done by server j. Let $q = [q_a] \in \mathbb{R}^A$ be the allocation vector. The LP is then defined as follows.

$$\begin{aligned} &\text{Minimize} && \rho' && (7)\\ &\text{subject to} && e + Dq = \mathbf{0} && \\ & && \rho' \geq \sum_{a\in\mathcal{A}_j} q_a, \qquad \forall\ 1 \leq j \leq J, && (8)\\ & && q_a \geq 0, && (9)\end{aligned}$$

where $\mathbf{0}$ is the vector of zeros, and for length K' vectors x and y, the notation $x < y$ means that $x_{k'} < y_{k'}$ for all k'. The *capacity region* Λ' of the virtual queueing network is the set of all λ (or equivalently the set of corresponding arrival vectors e) for which the solution of the LP satisfies $\rho' \leq 1$.

The arrival vector λ (equivalently the corresponding arrival vector e) is in the interior of capacity region if there exists a feasible capacity allocation vector q^* such that

$$e + Dq^* < \mathbf{0}. \qquad (10)$$

Note that at a first glance, it is not clear that the LP corresponding to the queueing network mentioned in (7) is equivalent to the LP that characterizes the capacity region of the original DAG scheduling problem. In the following, we show the equivalence of these two LPs for the example of the DAG shown in Figure 1 and the queueing network shown in Figure 3. For a general DAG, a similar argument can be made which we do not provide due to space constraints.

It is clear that queueing cannot increase the capacity region. To see this formally, suppose that there exists a λ and feasible q such that $\rho' \leq 1$. Let $\mathcal{A}'_k$ be the set of activities in which task k is served. Then, by conservation of flow, $\lambda = \sum_{j:k\in\mathcal{T}_j} \mu_{kj} \sum_{a\in\mathcal{A}'_k\cap\mathcal{A}_j} q_a$ for all k. Now one can choose a feasible allocation vector p for the original DAG scheduling problem such that $p_{kj} = \sum_{a\in\mathcal{A}'_k\cap\mathcal{A}_j} q_a$, and guarantee that $\rho \leq 1$. Thus, $\Lambda' \subseteq \Lambda$. To show that $\Lambda \subseteq \Lambda'$, consider some $\lambda \in \Lambda$ and its corresponding allocation vector p. We propose an allocation vector q that can support λ as follows. For the example of Figure 3, we design q such that the flow is divided to two equal flows in the branches from virtual queue $\{2,3,4\}$ to virtual queues $\{3,4\}$ and $\{2,4\}$.

$$D = \begin{pmatrix} -\mu_{11} & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \mu_{11} & -\mu_{21} & -\mu_{31} & -\mu_{32} & 0 & 0 & 0 & 0 \\ 0 & 0 & \mu_{31} & \mu_{32} & -\mu_{21} & 0 & 0 & 0 \\ 0 & \mu_{21} & 0 & 0 & 0 & -\mu_{31} & -\mu_{32} & 0 \\ 0 & 0 & 0 & 0 & \mu_{21} & \mu_{31} & \mu_{32} & -\mu_{42} \end{pmatrix}. \tag{6}$$

Then, we have

$$q_1 = p_{11}, q_2 = \frac{1}{2}p_{21}, q_3 = \frac{1}{2}p_{31}, q_4 = \frac{1}{2}p_{32}$$
$$q_5 = \frac{1}{2}p_{21}, q_6 = \frac{1}{2}p_{31}, q_7 = \frac{1}{2}p_{32}, q_8 = p_{42}.$$

It is easy to check that q is feasible (as p is feasible), and also it satisfies the flow conversation equation. Thus, $\Lambda \subseteq \Lambda'$. For a general DAG, in the virtual queueing network, one can similarly divide the flow equally from a branching node in to the following branches, and then construct a feasible q that also satisfies the flow conservation equality.

Now we give a description of the Max-weight policy for our virtual queueing system. Given virtual queue-lengths $Q^n_{k'}$ at time n, Max-weight policy allocates a service vector q that is

$$\arg \max_{q \text{ is feasible}} -(Q^n)^T Dq,$$

where $Q^n = [Q^n_{k'}]$ is the vector of queue-lengths at time n. The Max-Weight policy is the choice of q that minimizes the drift or rate of change of a Lyapunov function $V(Q^n) = \sum_{k'}(Q^n_{k'})^2$. The following example clarifies Max-Weight scheduling in our network.

Example. Consider the network shown in Figure 3. Assume that the virtual queues are indexed by their corresponding subset of tasks. Then, as an example, at time n, the Max-Weight policy assigns server 1 to task 1 in queue $\{1, 2, 3, 4\}$ if activity 1 has the largest weight among the activities that server 1 can perform. That is,

$$\mu_{11}[Q^n_{1234} - Q^n_{234}] > \mu_{21}[Q^n_{234} - Q^n_{34}],$$
$$\mu_{11}[Q^n_{1234} - Q^n_{234}] > \mu_{21}[Q^n_{24} - Q^n_{4}],$$
$$\mu_{11}[Q^n_{1234} - Q^n_{234}] > \mu_{31}[Q^n_{234} - Q^n_{24}],$$
$$\mu_{11}[Q^n_{1234} - Q^n_{234}] > \mu_{31}[Q^n_{34} - Q^n_{4}].$$

We call a scheduling policy throughput-optimal if it stabilizes the network for all the arrival rates in the capacity region. The following theorem, similar to Theorem 4 of [10], shows that the Max-Weight policy is rate stable for all the arrival vectors in the capacity region presented in (8), and it makes the underlying Markov process of the queue-lengths positive recurrent for all the arrival rate vectors that are in the interior of the capacity region.

Theorem 1: Max-weight policy for the network of virtual queues described in Subsection II-B is throughput-optimal.

Proof: [Sketch] Let us consider the problem in the fluid limit. Details of under what conditions and why stability of fluid limit implies stability of stochastic system can be found in [9]. Define the amount of fluid in virtual queue k' as

$$X_{k'}(t) = \lim_{r\to\infty} \frac{Q^{\lfloor rt \rfloor}_{k'}}{r}.$$

Then, the fluid model equations are

$$X(t) = X(0) + et + DT(t),$$

where $X(t) = [X_{k'(t)}]$ is the vector of queue-lengths, $T_a(t)$ is the total time up to t that activity a is served, and $T(t) = [T_a(t)]$ is the vector of total service times of different activities. Further equation incurred by max-weight policy is that

$$\dot{T}(t) = \arg \max_{q \text{ is feasible}} -X^T(t)Dq.$$

Now take $V(X(t)) = \frac{1}{2}X^T X$ as the Lyapunov function. The drift of $V(X(t))$ is

$$\begin{aligned} \dot{V}(X(t)) &= X^T(t)(e + D\dot{T}(t)) \\ &= X^T(t)e - \max_q \left(-X^T(t)Dq\right) \\ &\le X^T(t)(e + Dq^*). \end{aligned} \tag{11}$$

Now if e is in the interior of the capacity region, we have

$$\dot{V}(X(t)) \le X^T(t)(e + Dq^*) < \mathbf{0}.$$

This proves that the fluid model is stable which implies the positive recurrence of the underlying Markov chain [9]. To prove rate stability, it is sufficient to show that the fluid model is *weakly* stable, that is if $X(0) = \mathbf{0}$, then $X(t) = \mathbf{0}$ for all $t \ge 0$. This is again a direct result of Equation (11) since $e + Dq^* \le \mathbf{0}$. ■

Remark 2: The same analysis goes through for the serialized queueing network proposed in Subsection II-C. Indeed, the re-entrant line is a subset of the general queueing network that we considered in this section. Moreover, the capacity region of the re-entrant line is trivially identical to the original DAG scheduling problem since all the queues corresponding to DAG type m have nominal traffic of λ_m, which leads to exactly the same LP as stated in (2).

IV. Towards Analyzing Delay Performance of Different Queueing Networks

Given the throughput optimality of the scheduling policies in preceding sections, it is natural to consider and compare their delay performance, e.g., the steady-state expected sojourn time of jobs under the respective policies. However, as one may expect, the exact delay performance of the general model that we consider is very difficult to analyze, if not impossible, as other than for very simple examples, the underlying Markov chain of the system is almost intractable. A possible approach to get tractable performance bounds is to use tools from heavy-traffic analysis, such as those found in [14], [11]. The complexity of our system model poses challenges in utilizing these tools, and we leave this as future work.

In this section, we provide some illustration about the effect of adding precedence constraints among tasks of a job on the delay of processing jobs. Thus, in order to get some intuition about the effect of precedence constraints on delay, we simplify our model and our scheduling policy as follows.

We simplify our model by considering 1 job type that has K identical tasks, and J identical servers. More precisely, we consider jobs that arrive as a Poisson process with rate λ. Each job consists of K tasks, and there are J servers. Each task can be processed by all servers, and requires an independent exponentially distributed service time with mean 1. We compare job scheduling in the following two extreme cases. In the first case, which we call the "parallel" system, $\mathcal{P}$, the K tasks of a job can be processed in parallel; that is, the servers are allowed to work in parallel on two tasks of the same job. We consider a *First-Come-First-Serve* (FCFS) scheduling policy. In the second case, we consider a serial system, $\mathcal{S}$, in which the servers can work only on one task of a job at a time, and they also do so on a FCFS basis. Thus, the system $\mathcal{S}$ imposes a serial execution of the K tasks of each job; that is, a server must complete task 1 before it can start task 2, and so on, similar to the virtual queueing network considered in Subsection II-C.

We define the state of the parallel system to be

$$Z(\mathcal{P},t) = (X(\mathcal{P},t), Y(\mathcal{P},1,t), \ldots, Y(\mathcal{P},K-1,t)),$$

where $X(\mathcal{P},t)$ is the number of jobs in the parallel system at time t, and $Y(\mathcal{P},k,t)$ is the number of jobs for which k of the K tasks are already completed at time t. Similarly we define

$$Z(\mathcal{S},t) = (X(\mathcal{S},t), Y(\mathcal{S},1,t), \ldots, Y(\mathcal{S},K-1,t)),$$

where $X(\mathcal{S},t)$ is the number of jobs in the serial system at time t, and $Y(\mathcal{S},k,t)$ is the number of jobs for which k of the K tasks are already completed at time t. Note that $(Z(\mathcal{P},t), t \geq 0)$ and $(Z(\mathcal{S},t), t \geq 0)$ are Markov processes.

Clearly, the stability region of the two systems is the following set of arrival rates: $K\lambda \leq J$. Indeed, work arrives at the systems at rate $K\lambda$ and is performed at rate J when there is enough work to be processed.

Theorem 2: The Markov process corresponding to the two systems under FCFS scheduling is positive recurrent if $\lambda < J/K$.

Proof: We use Foster-Lyapunov theorem to show the positive recurrence of the Markov chain. The following analysis holds for both systems. Let $W(t) = KX(t) - \sum_{k=1}^{K-1} kY(k,t)$ denote the number of tasks yet to be processed in the system. Suppose that $\lambda - J/K < -\epsilon$. Then, if $w > J$,

$$E[W(t+dt) - W(t) | W(t) = w] = K\lambda dt - Jdt < -K\epsilon dt.$$

So the drift of the Lyapunov function $W(t)$ is negative outside the bounded set $W(t) \leq J$, and the Markov chain is positive recurrent. ■

The key result of this section is the following.

Theorem 3: Let D_S and D_P be the average delays in the two systems. Then,

$$D_S \leq D_P + \frac{2K-1}{\rho},$$

where $\rho = \frac{K\lambda}{J}$ is the load of the system.

Remark 3: Theorem 3 shows that the delay in the serial system is approximately the same as in the parallel system when the load ρ is not small, and K is not large. This result is of course limited to the case that the servers and tasks are identical, but it suggests that enforcing additional precedence constraints to a DAG scheduling problem does not significantly increase the delay when the load of the network is not small.

Proof: To prove the theorem, we use a coupling argument as follows.

The key observation is that when $X(t) \geq J$, the J servers are all busy in both systems, so that they complete a task at the same rate J. We can then couple the task completion times and the arrivals in the two systems.

Let

$$W(\mathcal{S},t) = KX(\mathcal{S},t) - \sum_{k=1}^{K-1} kY(\mathcal{S},k,t)$$

be the number of tasks to be completed in the serial system and $W(\mathcal{P},t)$ the corresponding value for the parallel system.

Lemma 1: It is possible to couple the systems so that

$$W(\mathcal{S},t) \leq W(\mathcal{P},t) + KJ, \forall t \geq 0.$$

Proof: Assume $W(\mathcal{S},t) = W(\mathcal{P},t) + KJ$. Then there are at least J jobs in system $\mathcal{S}$, so that all its servers are busy. Consequently, the coupling guarantees that if system $\mathcal{P}$ completes a task, so does system $\mathcal{S}$. Hence, in our coupling, it is not possible that $W(\mathcal{S},t) > W(\mathcal{P},t) + KJ$. ■

A direct consequence is as follows.

Lemma 2: Let $c = 2 - 1/K$. It is possible to couple the systems so that

$$X(\mathcal{S},t) \leq X(\mathcal{P},t) + cJ, \forall t \geq 0.$$

Proof: Assume $X(\mathcal{S},t) > X(\mathcal{P},t) + cJ$. Then,

$$\begin{aligned} W(\mathcal{S},t) - W(\mathcal{P},t) &= K(X(\mathcal{S},t) - X(\mathcal{P},t)) \\ &- \sum_{k=1}^{K-1} kY(\mathcal{S},k,t) + \sum_{k=1}^{K-1} kY(\mathcal{P},k,t) \\ &> KcJ - \sum_{k=1}^{K-1} kY(\mathcal{S},k,t) \\ &\geq KcJ - (K-1)J \geq J[K(c-1)+1] \\ &= KJ, \end{aligned}$$

which contradicts the previous lemma.

In the derivation, we used two observations. First,

$$\begin{aligned} \sum_{k=1}^{K-1} kY(\mathcal{S},k,t) &\leq (K-1)\sum_{k=1}^{K-1} Y(\mathcal{S},k,t) \\ &\leq (K-1)J. \end{aligned}$$

This is the case because every job with a partial number of completed tasks occupies a server. Second,

$$K(c-1)+1=K$$

because $c = 2 - 1/K$.

■

We can now complete the proof of the theorem.

From Little's result, we have

$$\begin{aligned} D_S &= \frac{E(X(\mathcal{S}))}{\lambda} \\ &\leq \frac{E(X(\mathcal{P})) + cJ}{\lambda} \\ &= D_P + \frac{cJ}{\lambda}. \end{aligned}$$

■

V. Conclusion and Future Work

We considered the problem of dynamic DAG scheduling in a flexible server system, motivated by applications in areas such as data centers and healthcare. A major challenge to efficient scheduling is the synchronization problem, further complicated by the presence of flexible servers and tasks. We proposed two Max-Weight type policies to resolve the synchronization issue, both of which involve considering virtual queues, and are throughput optimal. Their delay performances are difficult to analyze. To gain insights into the delay properties, we consider a stylized model and compare the delay performances of a serial system and a parallel system. An important future direction is the characterization of delay properties of the policies that we have proposed, as well as the development of scheduling policies with both good performance and low complexity.

References

[1] M. Armony, S. Israelit, A. Mandelbaum, Y. N. Marmor, Y. Tseytlin and G. B. Yom-Tov. Patient flow in hospitals: A data-based queueing-science perspective. Working paper.

[2] R. Atar, A. Mandelbaum and A. Zviran. Control of fork-join networks in heavy traffic. *Allerton*, 2012.

[3] F. Baccelli and Z. Liu. On the execution of parallel programs on multiprocessor systems: A queueing theory approach. *Journal of the ACM*, vol. 37, no. 32, pp. 373–414, 1990.

[4] F. Baccelli and Z. Liu. Stability condition of a precedence based queueing discipline. *Advances in Applied Probability*, vol. 21, pp. 883–898, 1988.

[5] F. Baccelli and A. M. Makowski. Simple computable bounds for the fork-join queue. *Proc. John Hopkins Conf. Inf. Sci.*, John Hopkins Univ., 1985.

[6] F. Baccelli, A. M. Makowski and A. Schwartz. The fork-join queue and related systems with synchronization constraints: Stochastic ordering, approximations and computable bounds. *Journal of Applied Probability*, vol. 21, pp. 629–660, 1989.

[7] F. Baccelli, W. A. Massey and A. Towsley. Acyclic fork-join queueing networks. *Journal of the ACM*, vol. 36, no. 3, pp. 615–642, 1989.

[8] N. Bambos and J. Walrand. On stability and performance of parallel processing systems. *Journal of the ACM*, vol. 38, pp. 429–452, 1991.

[9] J. G. Dai. On positive harris recurrence of multiclass queueing networks: A unified approach via fluid limit models. *Annals of Applied Probability*, vol. 5, pp. 49–77, 1995.

[10] J. Dai and W. Lin. Maximum pressure policies in stochastic processing networks. *Operations Research*, vol. 53, pp. 197–218, 2005.

[11] J. Dai and W. Lin. Asymptotic optimality of maximum pressure policies in stochastic processing networks. *Annals of Applied Probability*, vol. 18, pp. 2239–2299, 2008.

[12] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. *Communications of the ACM*, vol. 51, pp. 107–113, 2008.

[13] B. Dodin. Bounding the project completion time distribution in PERT networks. *Operations Research*, vol. 33, no. 4, pp. 862–881, 1985.

[14] A. Eryilmaz and R. Srikant. Asymptotically tight steady-state queue length bounds implied by drift conditions *Queueing Systems*, vol. 72, pp. 311–359, 2012

[15] M. Isard, M. Budiu, Y. Yu, A. Birrell and D. Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. *Eurosys*, 2007.

[16] L. Flatto and S. Hahn. Two parallel queues created by arrivals with two demands I. *SIAM Journal on Applied Mathematics*, vol. 44, no. 5, pp. 1041–1053, 1984.

[17] L. Flatto and S. Hahn. Two parallel queues created by arrivals with two demands II. *SIAM Journal on Applied Mathematics*, vol. 45, no. 5, pp. 861–878, 1985.

[18] R. W. Hall. (Ed.) *Patient flow: Reducing delay in healthcare delivery.* Springer, 2006.

[19] J. M. Harrison. Brownian models of open processing networks: canonical representation of workload. *The Annals of Applied Probability*, vol. 10, no. 1, pp. 75–103, 2000.

[20] P. Konstantopoulos and J. Walrand. Stationary and stability of fork-join networks. *Journal of Applied Probability*, vol. 26, pp. 604–614, 1989.

[21] P. R. Kumar. Re-entrant lines. *Queueing Systems: Theory and Applications: Special Issue on Queueing Networks*, vol. 13, no. 1, pp. 87–110, 1993.

[22] Y. Kwok and I. Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *Journal of ACM Computing Surveys*, vol. 31, pp. 406–471, 1999.

[23] M. Mandelbaum and B. Avi-Itzhak. Introduction to queueing with splitting and matching. *Israel Journal of Technology*, vol. 6, pp. 288–298, 1968.

[24] R. Nelson and A. N. Tantawi. Approximate analysis of fork/join synchronization in parallel queues. *IEEE Transactions on Computers*, vol. 37, no. 6, pp. 739–743, 1988.

[25] V. Nguyen. Processing networks with parallel and sequential tasks: Heavy-traffic analysis and Brownian motion. *The Annals of Applied Probability*, vol. 3, no. 1, pp 28–55, 1993.

[26] V. Nguyen. The trouble with diversity: Fork-join networks with heterogeneous consumer population. *The Annals of Applied Probability*, vol 4, no. 1, pp 1–25, 1994.

[27] R. Pedarsani, J. Walrand, Y. Zhong. Robust Scheduling in a flexible fork-join network *Proceedings of IEEE Conference on Decision and Control (CDC)*, 2014.

[28] R. A. Sahner and K. S. Trivedi. Performance and reliability analysis using directed acyclic graphs. *IEEE Transactions on Software Engineering*, vol. 13, no. 10, pp. 1105–1114, 1987.

[29] J. N. Tsitsiklis, C. H. Papadimitriou and P. Humblet. The performance of a precedence-pased queueing discipline. *Journal of the ACM*, vol. 33, no. 3, pp. 593–602, 1986.

[30] S. Varma. *Heavy and light traffic approximations for queues with synchronization constraints.* Ph.D dissertation, Department of Electrical Engineering, University of Maryland, 1990.

[31] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. *NSDI*, 2012.