# Aggregated-DAG Scheduling for Job Flow Maximization in Heterogeneous Cloud Computing

Boonyarith Saovapakhiran*, George Michailidis†, Michael Devetsikiotis*,
*Department of Electrical and Computer Engineering, North Carolina State University, Raleigh, NC 27695
†Department of Statistics, University of Michigan, Ann Arbor, MI 48109

*Abstract*—**Heterogeneous computing platforms such as Grid and Cloud computing are becoming prevalent and available online. As a result, resource management in these platforms is fundamentally critical to their global performance. Under the assumption of jobs comprised of subtasks forming DAG jobs, we focus on how to increase utilization and achieve near-optimal throughput performance on heterogeneous platforms. Our analysis and proposed algorithm are analytically derived and establish that, by aggregating multiple jobs using *good* scheduling, a near-optimal throughput can be achieved. Consequently, its limit is asymptotically converging to a certain value and can be written in the form of the service time of subtasks. Furthermore, our analysis shows how to explicitly compute the optimal throughput of computing systems, an important task for such a complex scheduling problem. In addition, we derive a simple super-job scheduling and show that its performance in term of throughput is better than the well-known Heterogeneous Earliest-Finish-Time (HEFT) algorithm.**

## I. Introduction

The evolution of the Internet has also led to the rapid interconnectedness of computing resources. As a result, novel computing platforms such as cloud [1] and its precursor, grid computing [2] emerged and have become the standard for computing intensive applications. Hence, the problem of resource management in such environments has received a lot of attention from the research community over the last few years. Many applications run in these environments have a complex structure and can be modeled as Directed Acyclic Graph (DAG) workflows [3]. For instance, scientific workflows and clouds (see [4]) are considered as a new challenge on how to leverage the computing power of cloud and grid computing on science workflow applications. Specifically, each "job" submitted comprises of a number of tasks that are mapped to the nodes of a DAG and hence their execution is subject to given precedence constraints; i.e. the execution of a downstream task can not begin until all its precedent ones have been completed. The problem of DAG scheduling has been extensively investigated in grid computing [5], but significantly less in cloud computing [6].

However, most of the emphasis has been on *static* scheduling algorithms [7], where all the information about the characteristics of the tasks in a job (e.g. workload) together with the processing power of available resources is known a priori at scheduling time. On the other hand, in *dynamic scheduling* [8], such information becomes available only when the job arrives at the system and the scheduling algorithm is required to map tasks to available resources on the fly. Note that static scheduling is closely related to the problem of scheduling DAGs on a multi-processor environment [9], for which a large number of heuristics has been proposed to primarily minimize the makespan of the job.

Another important characteristic of such scheduling algorithms is whether they are *centralized* or *distributed*. In the former case, there exists a global scheduler for assigning the jobs and their tasks to the available resources, while in the latter "local" schedulers control their own resources. Other features that can be incorporated in the job scheduling include business cost minimization [10], resource utilization maximization [11], throughput maximization, load balancing, achieving fairness, and meeting budgetary or deadline constraints [12]. Finally, some recent work focuses on scheduling simultaneously multiple jobs [13].

In this study, our objective is to dynamically schedule DAGs in a cloud/grid computing environment so that the throughput of the system is maximized. The main challenge in dynamic scheduling is that a job is scheduled upon arrival and therefore it may lead to resource underutilization until another job arrives. Our proposed approach is based on the idea that if there are present a number of jobs for scheduling, a better allocation of resources can be achieved (see example in Section III). To achieve our goal of throughput maximization the proposed algorithm schedules jobs in batches, thus eliminating to a large extent such inefficiencies. It shares some features with the work in [14], [15], but the techniques used are substantially different. Hence, we combine centralized static scheduling with a dynamic flow of jobs that makes obtaining analytical results about the performance of our algorithm tractable. The main contribution of this study are two-fold. First, our analytical results allow one to explicitly compute the optimal throughput and capacity region of the computing system, which are very critical for system design. Furthermore, it is established that even the simple super-job scheduling can acheive asymptotically the maximum throughput. Both are our main contributions.

The remainder of the paper is organized as follows: the problem under consideration is presented in Section II. An analytic expression for the throughput is derived in Section III, together with a condition for the system's stability. The proposed scheduling algorithm is presented in Section IV and its performance is illustrated in Section V. Some concluding remarks are drawn in Section VI.

## II. Problem Formulation

We start by denoting by $\mathbf{R} = \{P_1, P_2, ..., P_{|R|}\}$ the set of all available resources (e.g. processors or virtual machines), with $P_j$ corresponding to the $j$-th resource, and $|\mathbf{R}|$ the total number of resources. In this work, the communication cost between resources is incorporated in the processing time of the jobs. Further, there exists a global scheduler mapping the incoming jobs (and their tasks) to the available resources. The processing power of the $j$-th resource can be thought of as instructions per unit of time (e.g. second).

We specify next the structure of incoming jobs. They arrive to the global scheduler at time $\{t_i\}$ where $i \in \mathbf{Z}$. Each job is characterized by a DAG graph $G_i = (V_i, E_i)$ where $V_i = \{n_i^1, n_i^2, ..., n_i^{|V_i|}\}$ is the set of nodes/subtasks with $n_i^j$ denoting the $j$-th node/subtask of job $G_i$, and $E_i = \{(n_i^j, n_i^k) | n_i^j, n_i^k \in V_i$ the edge set. An edge between two subtasks exists if there is a directed link from $n_i^j$ to $n_i^k\}$. For any $(n_i^j, n_i^k) \in E_i$, $n_i^j$ is called an *Ancestor* node of $n_i^k$, while $n_i^k$ a *Child* node of $n_i^j$. Nodes without Ancestors are called *Entry* nodes. In contrast, nodes without Children are called *Exit* nodes. Due to a precedence constraint, $(n_i^j, n_i^k) \in E_i$, subtask $n_i^k$ can be processed only if the execution of task $n_i^j$ is completed.

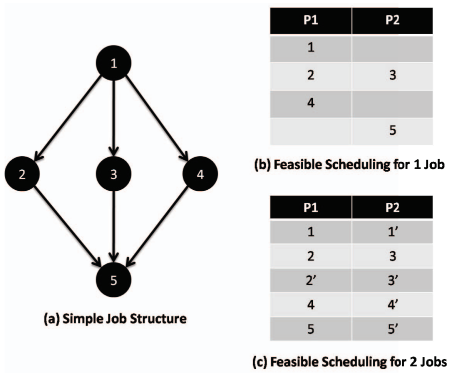
Let $|n_i^j|$ be the size of this subtask of $G_i$ expressed in instruction (processing) units. The service time of the subtask $n_i^j$ assigned to resource $P_\ell$ is $\sigma(n_i^j, P_\ell) = \frac{|n_i^j|}{|P_\ell|}$. Note that in principle service times could include communication times between processors, as well as migration times for moving a subtask from one resource to another. In this study, such times are incorporated in the service times and ignored in the remainder.

Next we define the system dynamics. First, the $i$-th arriving job has task graph $G_i$ and is randomly drawn from a finite collection of $L$ feasible graphs in the set $\mathbf{G} = \{g_1, g_2, ..., g_L\}$.; i.e. it is assumed that the types of jobs submitted to the system can belong to one of $L$ types. It is further assumed that all members of $\mathbf{G}$ have finite number of nodes/subtasks and also that $|n_i^j|$ is bounded for all $i, j$. Let $\sigma_{G_i} = (\sigma_i^1, ..., \sigma_i^{|V_i|})$ be the vector of service times if they were assigned to a benchmark resource with processing power *one instruction per unit time*; i.e. $\sigma_i^j = |n_i^j|$. The arrival process of jobs to the system $\mathbf{A} = \{(\tau_j, G_j, \sigma_{G_j})\}_{j=1}^{\infty}$ for $j \in \{1, 2, ...\}$ is assumed to be a random marked point process, where $\tau_j$ is the inter-arrival time between the $(j-1)$-th and $j$-th arrivals and $(G_j, \sigma_{G_j})$ its mark [16]. Further, we assume that the $\{\tau_j\}$'s are independent and identically distributed for all $j$ from a general distribution with finite first moment; i.e. $\mathbf{E}[\tau_j] < \infty$ and we define the arrival rate as $\lambda = \frac{1}{\mathbf{E}[\tau_j]}$. Finally, we denote by $\xi_j$ the total service time of job $j$; i.e. $\xi_j = \sum_{i=1}^{|V_j|} \sigma_j^i$. It is also assumed that $\mathbf{E}[\xi_j] < \infty$ and that the arrival process is stationary with respect to time shifts and also ergodic. [16]

Given the stochastic structure of the input to the system, we would first like to determine its *throughput*. Then, based on this theoretical finding, we introduce our batch scheduling algorithm and establish that it asymptotically achieves it.

## III. System's Throughput

To gain insight into the structure of the problem, we examine the following simple example.


1
2
3
4
5
(a) Simple Job Structure
(b) Feasible Scheduling for 1 Job
(c) Feasible Scheduling for 2 Jobs

| P1 | P2 |
|---|---|
| 1 | |
| 2 | 3 |
| 4 | |
| | 5 |

| P1 | P2 |
|---|---|
| 1 | 1' |
| 2 | 3 |
| 2' | 3' |
| 4 | 4' |
| 5 | 5' |

Fig. 1. An example of job scheduling

A job with 5 subtasks is shown in Figure 1, where each subtask represents one unit of instructions. We suppose that only two resources are available. In Figure 1(b), we show a feasible schedule for one job over two resources, Processor 1 and 2 (P1 & P2) when each resource has the same processing power equal to one instruction per time unit. The scheduling in 1(a) requires 4 time units to finish a job; on the contrary, if there are two jobs in the system, then a feasible schedule is shown in 1(c), and we can see that two jobs can be completed within five time units. Obviously, the scheduling choice in 1(c) is more efficient than that in 1(b), since jobs are aggregated and mapped at the same time, and resource utilization increases. Conversely, the *scheduling holes*, i.e., the amount of time that resources are idle after scheduling, decrease. Our proposed scheduling algorithm takes advantage of this feature of the system to achieve asymptotically maximum throughput.

In calculating the throughput of the system, we assume that there exists an *optimal* algorithm for processing $N$ jobs. To consider the rate of processing $N$ jobs under the optimal algorithm, the system first gathers jobs with indices $\{k, k+1, ..., k+N\}$ in the buffer, where $k \in \{1, 2, ...\}$.

*Lemma 3.1:* Let $T_{k,N}^{opt}$ denote the time it takes under the optimal algorithm $|R|$ processors to execute jobs with indices $\{k, k+1, ..., k+N\}$. As the number of jobs $N$ goes to infinity, we have $\lim_{N\to\infty}[\frac{T_{k,N}^{opt}}{N}] = \lim_{N\to\infty}[\frac{\mathbf{E}[T_{k,N}^{opt}]}{N}] = \inf_{m>0}[\frac{\mathbf{E}[T_{k,N}^{opt}]}{N}] = \gamma$. That is, under an optimal scheduling algorithm, a normalized version of the optimal execution time of $N$ jobs converges to a limit.

*Proof:* First, we consider the key fact that $\{T_{k,N}^{opt}\}_{k=1}^{\infty}$ is subadditive. Specifically, it is easy to see that

$$T_{k,N}^{opt} \leq T_{k,l}^{opt} + T_{l,N}^{opt}, \text{ pathwise } \forall k, l, N \text{ s.t. } k \leq l \leq N$$

The result says that it is more efficient to schedule $N$ jobs together, rather than split them into two groups. By our assumption, the arrival process $\mathbf{A}$ is stationary and ergodic, so $T_{1,N}^{opt}$ and $T_{k,k+N}^{opt}$ have the same distribution and are also ergodic. Further, as $N$ increases, $T^{opt}_{k,N}$ is non-decreasing. Thus, $T^{opt}_{k,N}$ is a stationary, ergodic, path-wise and non-negative increasing function of $N$. The above result follows from an application of Kingman's sub-additive ergodic theorem [16]. Hence, we get $\lim_{N\to\infty}[\frac{T^{opt}_{k,N}}{N}] = \inf_{m>0}[\frac{\mathbf{E}[T^{opt}_{k,N}]}{N}] = \gamma$. ∎

We have established the existence of a limit for a normalized version of the execution time of jobs under an optimal scheduling algorithm. Next, we relate this limit $\gamma$ to the number of resources $|R|$ and the expected value of the job service times $\mathbf{E}[\xi]$.

*Lemma 3.2:* For a system where all the resources $|R|$ have unit processing power, we have that $\gamma = \frac{\mathbf{E}[\xi_i]}{|R|}$.

*Proof:* We observe that as long as there are more than $|R|$ jobs in the system, then the following holds:

$$\frac{1}{|R|}\sum_{j=1}^{N-1}\xi_j \le T^{opt}_{k,N}$$

Letting $Q_N = \max_{j\in\{k,k+1,...,k+N\}}\{\xi_j\}$, then we have

$$\frac{1}{|R|}\sum_{j=1}^{N}\xi_j + |R|Q_N \ge T^{opt}_{k,N}$$

Thus,

$$\frac{1}{|R|}\sum_{j=1}^{N-1}\xi_j \le T^{opt}_{k,N} \le \frac{1}{|R|}\sum_{j=1}^{N}\xi_j + |R|Q_N$$

Then, dividing both sides by $N$ and take the limit as $N$ go to $\infty$, we get

$$\lim_{N\to\infty}\frac{\frac{1}{|R|}\sum_{j=1}^{N-1}\xi_j}{N} \le \lim_{N\to\infty}\frac{T^{opt}_{k,N}}{N}$$

and

$$\lim_{N\to\infty}\frac{T^{opt}_{k,N}}{N} \le \lim_{N\to\infty}\frac{\frac{1}{|R|}\sum_{j=1}^{N}\xi_j}{N} + |R|\lim_{N\to\infty}\frac{Q_N}{N}$$

However,

$$\lim_{N\to\infty}\frac{1}{|R|}\frac{\sum_{j=1}^{N-1}\xi_j}{N-1}\frac{(N-1)}{N} = \frac{\mathbf{E}[\xi]}{|R|}$$

which follows from ergodicity of [16]. Similarly, $\lim_{N\to\infty}\frac{1}{|R|}\frac{\sum_{j=1}^{N}\xi_j}{N} = \frac{\mathbf{E}[\xi]}{|R|}$. Further, we will show that $\lim_{N\to\infty}\frac{Q_N}{N} = 0$.

We have that $Q_N$ is non-decreasing in $N$ as the maximum of the $\{\xi_j\}$ for all $j \in \{k, ..., k+N\}$, by definition. Let the jump points be at time $t_N$, e.g., $t_1 = \tau_1$, but this implies $Q_N = \xi_{t_N}$ for every integer $\in [t_N, t_{N+1})$. Then, if we take any increasing subsequence $\{N_s\}$ such that $\lim_{s\to\infty} N_s = \infty$ of the original sequence $\{N\}$ and define a point $a_s$ in the interval $[N_{a_s}, N_{a_{s+1}})$, we get

$$0 \le \lim_{s\to\infty}\frac{Q_{N_s}}{N_s} \le \lim_{s\to\infty}\frac{\xi_{N_{a_s}}}{N_{a_s}} = 0$$

Since $\xi_j$ is bounded and also $\{N_s\}$ is arbitrary, then the above is true for every subsequence, which completes the proof of the Lemma. ∎

*Remark:* Note that in the presence of resources with heterogeneous power, the above result becomes $\gamma = \frac{\mathbf{E}[\xi]}{\sum_{\ell=1}^{|R|}|P_\ell|}$, where $P_\ell$ denotes the processing power of the $\ell$-th resource.

*Proposition 3.3:* Let $\lambda$ be the arrival rate of jobs. If $\frac{1}{\lambda} < \gamma = \frac{\mathbf{E}[\xi]}{|R|}$, then the system is unstable.

*Proof:* First, we define $T^S_t$ as the completion time of jobs that have arrived by time $t$ under some scheduling algorithm $S$. Next, let $j_o(t)$ be the index of the last job arriving before time $t$ and no jobs are accepted after time $t$, so $j_o(t) = \max\{j \in \mathbf{Z} : \sum_{i=1}^{j-1}\tau_i < t\}$. We first show that

$$T^S_t \ge T^{opt}_{1,j_o(t)} - t$$

which follows from the definition of $T^{opt}_{1,j_o(t)}$. Therefore,

$$\begin{aligned}\liminf_{t\to\infty}\frac{T^S_t}{t} &\ge \liminf_{t\to\infty}\frac{T^{opt}_{1,j_o(t)}}{t} - 1\\ &\ge \liminf_{t\to\infty}[\frac{T^{opt}_{1,j_o(t)}}{j_o(t)}][\frac{j_o(t)}{t}] - 1\\ &\ge \gamma\liminf_{t\to\infty}[\frac{j_o(t)}{t}] - 1\end{aligned}$$

Recall that $\liminf_{t\to\infty}\frac{T^{opt}_{1,j_o(t)}}{j_o(t)} = \gamma$ as previously proved. In addition, we know that $t = \sum_{i=1}^{j}\tau_i$ and $\tau_i$ forms a stationary and ergodic sequence; hence, $\liminf_{t\to\infty}\frac{j_o(t)}{t} = \liminf_{j\to\infty}\frac{j-1}{\sum_{i=1}^{j}\tau_i}$. By the same technique as in lemma 3.2, we have $\liminf_{t\to\infty}\frac{j_o(t)}{t} = \lambda$. As a result, $\liminf_{t\to\infty}T^S_t = \infty$ as $t \to \infty$.

∎

We have calculated the capacity region of an optimal scheduling algorithm based on batch processing, as $N \to \infty$. Our next task is to design a practical and easy to implement algorithm that achieves the optimal throughput. Our scheduling algorithm, denoted henceforth by $S^*$, works as follows. The system starts empty at time 0 and waits until $N$ jobs have arrived. It then employs a low complexity algorithm described in Section IV to schedule them to the resources. While the first set of $N$ jobs is being processed, the second set of $N$ jobs is gathered. Once the first group finishes execution and the second set of size $N$ has been gathered, the algorithm proceeds in scheduling, while another set of $N$ jobs is being gathered, and so forth. Henceforth, we call a set of $N$ jobs a super-job; the $k$-th super-job consists of jobs with indices $j$ between $kN+1$ and $(k+1)N$ where $k \in \{1,2,...\}$. To determine the value of $N$ notice that the jobs in each group are scheduled according to schedule $S^*$, so the expected execution time of $k$-th $N$-group is given by

$$\mathbf{E}[T^{S^*}_{kN,(k+1)N}] = \mathbf{E}[T^{S^*}_{1,N}] < N\mathbf{E}(\tau_i)$$

Thus, $N$ needs to be chosen such that this relationship holds. The latter gives some guidance on the size of the super-job, but its exact value needs to be determined through simulation. For the proposed batch scheduling algorithm, we next establish that it is throughput maximizing.

*Proposition 3.4:* If $\frac{1}{\lambda} < \gamma$, then there exists a non-negative random variable $\psi$ such that

$$\lim_{k\to\infty} \mathbf{Pr}[T^{S^*}_{t_{k,N}} \in \mathbf{B}] = \mathbf{P}[\psi \in \mathbf{B}]$$

where $\mathbf{B}$ is a Borel measurable set of $\mathbf{R}^+$ and $t_{k,N}$ is arrival time of the $k$-th $N$-group job. Furthermore,

$$\lim_{k\to\infty} \frac{T^{S^*}_{t_{k,N}}}{k} = 0$$

hence, $T^{S^*}_{t_{k,N}}$ becomes zero infinitely often (i.o.) or almost surely (a.s.).

*Proof:* The $k$-th group starts processing at $t_{k,N}$ (last job of $(k-1)$-th group) and ends at time $t_{k+1,N}$ (arrival of last job). Thus, the expected time between each group is

$$\mathbf{E}\{t_{k+1,N} - t_{k,N}\} = \mathbf{E}\{\sum_{l=kN-1}^{(k+1)N-1} \tau_l\} = N\mathbf{E}(\tau_i)$$

Moreover, the service time of each group according to schedule $S^*$ is

$$\mathbf{E}\{T^{S^*}_{kN+1,(k+1)N}\} = \mathbf{E}\{T^{S^*}_{1,N}\} < N\mathbf{E}(\tau_i)$$

As a result, each group of jobs is processed in a First-Come-First-Serve (FCFS) fashion, so the system behaves like a $G/G/1$ queue, with each group corresponding to the *customers* in that paradigm. Then, the proposition is derived from the classical results on $G/G/1$ and all stated results follow [16].

■

*Remark:* For the asymptotic optimality, in terms of throughput, of the batch scheduling to hold, it requires that the static algorithm that schedules the $N$ jobs in the batch be of low computational complexity. Specifically, it requires that the scheduling time is sublinear in the number of jobs $N$ in the batch, which is achieved by our static scheduling algorithm discussed next.

## IV. Super-job Static Scehduling Algorithm

In this section, we design an efficient static scheduling algorithm for allocating the subtasks of $N$ batch jobs to the available resources, so that we minimize the makespan. We define the total makespan or scheduling length of a super-job comprising of $N$ jobs as $T^{S^*}_N$.

We note that the static version of the problem (scheduling subtasks of a DAG with unit service requirements to a finite number of processors) is NP-hard [9]. Our strategy is to decompose the main problem into multiple sub-problems through a divide-and-conquer strategy.

For any job $G_i$, it can easily be seen that subtask $n_i^{k_1}$ is only dependent on $n_i^{k_2}$ for some $k_1 \neq k_2$ and $k_1, k_2 \in V_i$ whereas $n_i^{k_1}$ is independent of $n_j^{k_2}$ for any $i \neq j$ and $i,j \in \{1,2,...,N\}$. This implies that we can aggregate these unrelated subtasks of the $N$ jobs into a batch and efficiently allocate them to proper resources to minimize $T^{S^*}_N$. The question is which subtasks we should select to aggregate in order to minimize the makespan. Indeed, every DAG graph can be divided into different depth-levels. For instance, let all entry nodes be in the first depth-level, its children be in the second depth-level, its grand children be in the third depth-level, and so forth. Since nodes are only dependent on the parent nodes, then, to maximize the degree of parallelism, it can be seen that nodes in the same depth level can be aggregated into a batch and scheduled at the same time. After all aggregated subtasks in each depth-level are scheduled, the descendent aggregated subtasks can be scheduled afterwards without a dependency concern, because all parent nodes of the current subtasks have finished execution. Hence, our first step is to categorize the subtasks from the $N$ super-job into multiple depth-level and aggregate all subtasks in the same depth-level together into a list corresponding to its depth-level.

After collecting substasks from the same depth-level together, we want to minimize the makespan in every depth-level. First, we redefine our notation as follows: let $\mathbf{R}(d)$ be the set index of all available resources in the $d$-th depth-level, $\zeta_i(d)$ is denoted to be the set index of subtasks in the $d$-th depth-level of job $G_i$ for all $i \in \{1,2,...,N\}$ where $a \neq b$ for all $a \in \zeta_i(d), b \in \zeta_j(d)$ for any $i \neq j$, and $\zeta(d) = \cup_i \zeta_i(d)$ is defined as the set index of all aggregated subtasks in $d$-th level. Moreover, $\sigma_{a,b}$ is the service time when the $b$-th nodes is assigned to the $a$-th resource where $a \in \mathbf{R}(d)$ and $b \in \zeta(d)$, and we have control variables $z_{a,b}(d)$ in $d$-th level and $z_{a,b}(d) = 1$ if $b$-th subtask is assigned to $a$-th resource and 0 otherwise. Then, the optimization subproblem in each $d$-th depth-level can be formulated as follows.

$$\begin{array}{ll} \text{Minimize :} & T \\ \text{subject to:} & \\ \quad (1) & \sum_{a\in\mathbf{R}(d)} z_{a,b}(d) = 1 \\ \quad (2) & \sum_{b\in\zeta(d)} (\sigma_{a,b} \times z_{a,b}(d)) \leq T \\ \quad (3) & z_{a,b}(d) \in \{0,1\} \end{array} \qquad (1)$$

This subproblem is classical and well-known under various names such as $Q||C_{max}$ and $R||C_{max}$ depending on the assumptions made regarding the processing capability of resources [17]. The problem under our consideration is the minimization of the makespan on *uniform parallel machines*, denoted by $Q||C_{max}$. It is well-known that even the simplest case $P_2||C_{max}$ is an NP-hard problem [17]. As a matter of fact, we can see that even the subproblem of the multiple DAG scheduling is still proven to be highly nontrivial. Most previous work is based on approximate and heuristic algorithms. As discussed in [17], there exists a polynomial-time $(1+\rho)$-approximate algorithm for $\rho > 0$ for $Q||C_{max}$.

Additionally, the problem can be interpreted as in the non-cooperative game-theoretic perspective as in [18]. The rules of the game are that, at every round, only one task (player) has the right to choose the preference resource (strategy) to optimize its cost function respectively. Each task can greedily migrate from the current to the resource that minimize its total makespan. A Nash equilibrium is reached when there is no further improvement from migrating any tasks from current resources to others. Also, a maximum task size with best-reply policy can be considered, where precisely at every iteration, a task with the maximum size is migrated to the target machine accomplishing minimal total makespan. We will call this policy the "max-load-best-reply policy" and implement it to heuristically solve the subproblem because it is shown in Theorem 4.3 of [18] that it can reach the Nash equilibrium within polynomial time, and so it can be deployed as a good approximation for locating a local optimal.

Thus, in brief, our Simple Superjob Scheduling Algorithm can be summarized as the follows. For all $i \in \{1, 2, ..., N\}$,

```
Step 1 : Aggregate subtasks at $d$-th level into $\zeta(d)$
for each $G_i$ do
    $\zeta(d) \leftarrow \zeta(d) \cup \zeta_i(d)$
end for
Step 2 : At every $d$-th level
for each $d \leq maxdepth$ do
    sort($\zeta(d)$) in descending order of task size
    for each $task_j \in \zeta(d)$ do
        allocate $task_j$ to $resource_k$ that min total makespan
        or $z_{k,j} \leftarrow 1$ for $k, j$ that $\min_{k \in R(d)} \sum_{j \in \zeta(d)} \sigma_{k,j} z_{k,j}$
    end for
    compute makespan of $d$-th level
    $AFT(d) \leftarrow \max_{i \in R(d)} AFT_i(d)$
end for
Step 3 : compute total makespan
makespan $\leftarrow \sum_{d=1}^{maxdepth} AFT(d)$
```

where $maxdepth$ is the maximum depth level for all $G_i$ job and $AFT_i(d)$ is the Actual Finished Time of $i$-th resource at $d$-th level.

## V. Simulation and Numerical Study

In this section, a simulation study is conducted to evaluate and verify that the proposed Simple Super-job Scheduling, denoted by "3S" in short, can achieve superior throughput performance. For benchmarking purposes, we compare it to the HEFT (Heterogeneous Earliest-Finish-Time) algorithm as proposed in [8], which is a widely-adopted dynamic scheduling algorithm.

### A. Simulation Setup

In our simulation, the DAG graph of $G_i$ is generated by applying the so-called *Layer-by-Layer* method as in [19], which is simple and efficient. First, the integer number of layers is uniformly generated from $[L_{min}, L_{max}]$, where $L_{min}$ and $L_{max}$ are the lower and upper bound of allowed layers. At every layer, an integer number of tasks is also uniformly generated from $[W_{min}, W_{max}]$, where $W_{min}$ and $W_{max}$ are the lower and upper bounds of the number of tasks in each layer. Then, a random adjacency DAG matrix is created according to the Layer-by-Layer algorithm with the probability of connecting two nodes equal to $p$. Next, $|K|$ finite DAG job patterns are generated, and the processing rate of $|R|$ resources is uniformly chosen from the integer set of $[S_{min}, S_{max}]$, where $S_{min}$ and $S_{max}$ is the minimum and maximum processing power (instructions per unit of time). Also, the size of each subtask for all $G_i$ is uniformly chosen from the integer set of $[T_{min}, T_{max}]$, where $T_{min}$ and $T_{max}$ are the minimum and maximum of feasible task size.

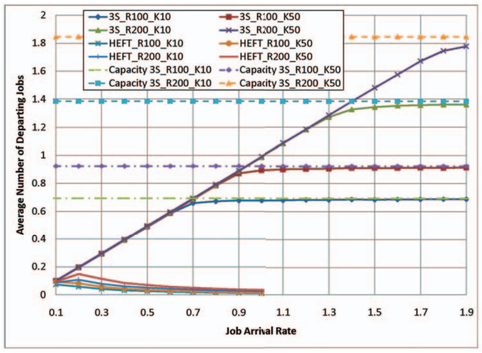
After all resources and job patterns are set, the event-driven simulation is run. Each job arrives to the system according to Poisson process of rate $\lambda \in \mathbf{R}^+$. When a job arrives to the system, it is immediately scheduled in the case of the HEFT algorithm. In contrast, super-job scheduling waits until $|N|$ jobs are aggregated and scheduled at once by the algorithm presented in Section IV. In our simulation, all parameters were set as follows: $[L_{min}, L_{max}] = [1, 5]$, $[W_{min}, W_{max}] = [1, 5]$, $p = 0.8$, $[S_{min}, S_{max}] = [1, 5]$, $[T_{min}, T_{max}] = [10, 50]$, $|K| \in \{10, 50, 1000\}$, $|R| \in \{100, 200\}$ and $|N| = 500$.

### B. Simulation Results

Our simulations are classified into the following two cases. In case I, we consider all resources either have an identical unit (homogeneous) or randomly generated (heterogeneous) processing power as shown in Figures 2(a) and 2(b). For both of them, we consider the following settings: number of resources $|R| \in \{100, 200\}$, number of job patterns $|K| \in \{10, 50\}$ and group size of aggregated jobs before scheduling is $|N| = 500$.

According to Figure 2(a), it can be seen that the maximum throughput that HEFT can achieve is around 0.2 for both $|R| = 100$ and $|R| = 200$. In contrast, the average throughput of 3S is significantly higher than HEFT's and it is around 0.9,1.7 for $|R| = 100, 200$ respectively when $|K| = 50$, because 3S has the advantage of increasing the utilization of available resources, due to the structure of the super-job. To match with our theoretical results, the average of total service time of random generated jobs for cases $|K| = 10$ and $|K| = 50$ are computed and equal to 144.4 and 108.34 respectively. By Lemma 3.1 and 3.2, with a *good* scheduling, the optimal throughput is given by $1/\gamma \equiv \beta = \frac{|R|}{\mathbf{E}\{\xi\}}$. As a result, by hand-computation, $\beta = 0.69, 0, 92, 1.385$, and 1.846 when $(|R| = 100, |K| = 10)$, $(|R| = 100, |K| = 50)$, $(|R| = 200, |K| = 10)$ and $(|R| = 200, |K| = 50)$, respectively. It can be easily seen that our theoretical results match the numerical result in Figure 2(a) with the 3S algorithm employed. Similarly, in Figure 2(b), the obtained results show that 3S still clearly outperforms HEFT even in the presence of heterogeneous processing power resources. The maximum throughput of HEFT is around 0.3 whereas 3S throughput is continuously increasing beyond that.

For case II, we also consider to find the capacity region of 3S and show that the numerical results match those from proposition 3.3. Thus, for better accuracy, a larger number of job patterns is generated and uniformly chosen for each arrival. In addition, the simulation is performed on identical resources with unit rate for our convenience to compute service time of jobs, then $|K|$ is set to 1000 and $|N|$ is set to 10. According to our random set of DAG patterns, the average total service time of jobs is 107.44. With these patterns, the number of resources are varied to see what the capacity regions are, and $|R|$ is set to be 10,25 and 80 as shown in Figure 3. According to our proposition 3.3, if $\lambda > \frac{|R|}{\mathbf{E}[\xi]}$, the system is unstable and



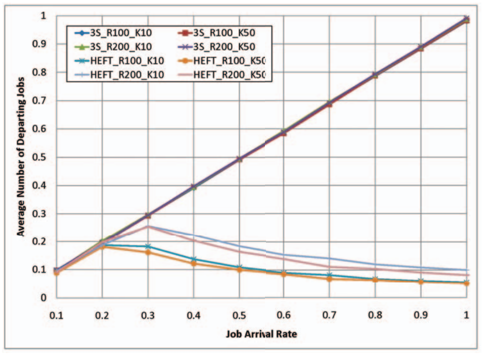


Fig. 2. Case I : Average Throughput for (a) Homogeneous resources (b) Heterogeneous resources

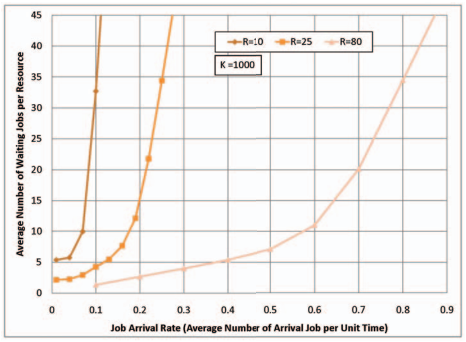


Fig. 3. Case II : Capacity Region for 3S Scheduling

number of waiting jobs is exploded. By hand-computation, if $\lambda > 0.093, 0.2326$, and $0.744$ for $|R| = 10, 25$ and $80$ respectively, the system will be unstable. As depicted in figure 3, it is easy to see that the simulations match the previously obtained theoretical results.

## VI. Conclusion and Future Work

Our batch-based DAG scheduling increases throughput and utilization, as supported by both our analytical and numerical results. We have shown that even simple super-job scheduling can achieve superior performance, in terms of throughput, than existing algorithms such as HEFT. Our method allows one to compute explicitly the throughput, which is an important contribution in itself. Furthermore, the idea of batching, as used in our approach, takes advantage of not letting processors become idle. In this paper we establish that this approach leads to an asymptotically optimal throughput. In our future work, we are aiming to design an algorithm that can achieve both high throughput and lower delay, when communication costs and time constraints are included.

## References

[1] R. Buyya, C. S. Yeo, S. Venugopal, J. Broberg, and I. Brandic, "Cloud computing and emerging it platforms: Vision, hype, and reality for delivering computing as the 5th utility," *Future Gener. Comput. Syst.*, vol. 25, pp. 599–616, June 2009.

[2] I. Foster, Y. Zhao, I. Raicu, and S. Lu, "Cloud computing and grid computing 360-degree compared," in *Grid Computing Environments Workshop, 2008. GCE '08*, 2008, pp. 1 –10.

[3] J. Yu and R. Buyya, "A taxonomy of workflow management systems for grid computing," *Journal of Grid Computing*, vol. 3, pp. 171–200.

[4] G. Juve and E. Deelman, "Scientific workflows and clouds," *Crossroads*, vol. 16, no. 3, pp. 14–18, 2010.

[5] F. Dong and S. G. Akl, "Technical report no. 2006-504 scheduling algorithms for grid computing: State of the art and open problems," 2006.

[6] C. G. Chaves, D. M. Batista, and N. L. S. da Fonseca, "Scheduling grid applications on clouds," in *GLOBECOM*, 2010, pp. 1–5.

[7] A. H. Alhusaini, V. K. Prasanna, and C. S. Raghavendra, "A unified resource scheduling framework for heterogeneous computing environments," in *Proceedings of the Eighth Heterogeneous Computing Workshop*, ser. HCW '99, 1999, pp. 156–165.

[8] H. Topcuouglu, S. Hariri, and M.-y. Wu, "Performance-effective and low-complexity task scheduling for heterogeneous computing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 13, pp. 260–274, March 2002.

[9] Y.-K. Kwok and I. Ahmad, "Static scheduling algorithms for allocating directed task graphs to multiprocessors," *ACM Comput. Surv.*, vol. 31, pp. 406–471, December 1999.

[10] "Maximizing business value by optimal assignment of jobs to resources in grid computing," *European Journal of Operational Research*, vol. 194, no. 3, pp. 856 – 872, 2009.

[11] J. Yu, M. Li, Y. Li, F. Hong, and M. Gao, "A framework for price-based resource allocation on the grid," in *Parallel and Distributed Computing: Applications and Technologies*, ser. Lecture Notes in Computer Science, K.-M. Liew, H. Shen, S. See, and W. Cai, Eds. Springer Berlin / Heidelberg, vol. 3320, pp. 63–100.

[12] J. Yu and R. Buyya, "Scheduling scientific workflow applications with deadline and budget constraints using genetic algorithms," *Sci. Program.*, vol. 14, pp. 217–230, December 2006.

[13] L. Bittencourt and E. Madeira, "Towards the scheduling of multiple workflows on computational grids," *Journal of Grid Computing*, vol. 8, pp. 419–441.

[14] J. Barbosa and B. Moreira, "Dynamic scheduling of a batch of parallel task jobs on heterogeneous clusters," *Parallel Computing*, vol. In Press, 2011.

[15] G. Malewicz and A. L. Rosenberg, "Batch-scheduling dags for internet-based computing," in *Euro-Par 2005 Parallel Processing*.

[16] F. Baccelli and P. Bremaud, *Elements of Queueing Theory: Palm Martingale Calculus and Stochastic Recurrences*. Springer, 2003.

[17] P. Schuurman and T. Vredeveld, "Performance guarantees of local search for multiprocessor scheduling," *INFORMS J. on Computing*, vol. 19, pp. 52–63, January 2007.

[18] E. Even-Dar, A. Kesselman, and Y. Mansour, "Convergence time to nash equilibrium in load balancing," *ACM Trans. Algorithms*, vol. 3, August 2007.

[19] D. Cordeiro, G. Mounié, S. Perarnau, D. Trystram, J.-M. Vincent, and F. Wagner, "Random graph generation for scheduling simulations," in *Proceedings of the 3rd International ICST Conference on Simulation Tools and Techniques*, ser. SIMUTools '10, 2010, pp. 60:1–60:10.