2011 IEEE International Parallel & Distributed Processing Symposium

Scheduling Functionally Heterogeneous Systems with Utilization Balancing

Yuxiong He, Jie Liu

Microsoft Research
yuxhe;liuj@microsoft.com

Hongyang Sun

Nanyang Technological University
sunh0007@ntu.edu.sg

***Abstract*—Heterogeneous systems become popular in both client and cloud. A parallel program can incur operations on multiple processing resources such as CPU, GPU, and vector processor units. This paper investigates scheduling problems on functionally heterogeneous systems with the objective of minimizing the completion time of parallel jobs.**

We first present performance bounds of online scheduling and show that any online algorithm is at best around $(K+1)$-competitive with respect to job completion time, where K is the total number of resource types. There exist "bad" jobs that prevent any online algorithms from obtaining good interleaving of heterogeneous tasks. This lower bound suggests that the relative performance of online algorithms versus an offline optimal could degrade linearly as types of heterogeneous resources increase.

The limitation of online scheduling motivates our study of how additional offline or lookahead information can help improve scheduling performance. We propose a Multi-Queue Balancing algorithm (MQB) that effectively transforms the problem of minimizing completion time to one of maximizing utilization of heterogeneous resources. It promotes interleaving of heterogeneous tasks through balancing the task queues of different types. Our simulation results suggest that MQB reduces the execution time of online greedy algorithms up to 40% over various workloads and outperforms other offline schemes in most cases. Furthermore, MQB can use limited and approximated offline information to improve scheduling decisions.

***Keywords*-completion time; DAG; functional heterogeneity; heterogeneous systems; offline; online; parallel; scheduling; utilization balancing;**

I. Introduction

Scheduling parallel jobs on multiprocessors, an important area of research in computer science, has been extensively explored on homogeneous resources [4], [6], [9], [17], [23], [27], [31]. However, heterogeneous systems are becoming popular in both client and cloud with the increasing popularity of various accelerators, such as GPU, FPGA, and vector processing units, as well as better programming support across heterogeneous platforms [13], [18], [24], [25].

Heterogeneity in computer systems can be characterized in two ways. *Performance heterogeneity* exists in systems that contain general-purpose processors of different speeds. A task can execute on any processor, but it will execute faster on some than others. Executing parallel programs on processors with different speeds has been studied intensely in scheduling theory [4], [8], [10], [11], [31].

Functional heterogeneity, on the other hand, exists in systems that contain various types of functional units, and not all tasks of an application can be executed on all functional units. A system that mixes different physical processors and co-processors such as CPU, GPU, FPGA, DSP, and vector processing units is functionally heterogeneous. However, functional heterogeneity is not limited to hardware resources. For example, security and privacy needs may restrict some application computations that access sensitive data on local machines, while others can be performed in a data center. Software licenses can impose constraints that differentiate the functionality of machines as well.

We motivate our problem further by a map-reduce style parallel data analysis framework, known as Cosmos [2], that runs on a cluster of tens of thousands of commodity servers in the back end of the Bing search engine from Microsoft. User programs in Cosmos are written in Scope [7], a mash-up language that mixes SQL-like queries with native code. The Scope compiler transforms a program, called a job, into a workflow, which is a directed acyclic graph. Each node of the graph represents a set of tasks that runs in parallel to perform a computation on different parts of the input stream (i.e., data parallelism). A task, assigned to a server based on data availability, will read its input over the network if not locally available. Task outputs are always written to the local disk. Each edge that joins two nodes represents a data production and consumption relationship. A job can have about 20 nodes on average, and each node can involve thousands of servers. In a typical day, Cosmos handles over a thousand jobs, with a total computation time over 50 server-year.

We address the scheduling problem for functionally heterogeneous systems (FHSs) like Cosmos. To minimize the job completion time, schedulers should assign tasks to machines based on data locations, dependence patterns and cluster-wide resource availability. As a simplified abstraction, we assume that the servers are clustered into classes based on their data allocation and tasks are not assigned across classes. Thus, we treat server classes as heterogeneous functional units.

In an FHS, an application can perform operations on multiple processing resources. We model such systems and jobs as follows. Processors are categorized into K

1530-2075/11 $26.00 © 2011 IEEE
DOI 10.1109/IPDPS.2011.113

types. A parallel job can consist of different types of tasks, and each task type can be executed only on the matching type of resources. We model the execution of a parallel job as a directed acyclic graph (DAG). We extended the conventional DAG in prior work on homogeneous resources [5], [6], [14], [22], [26] to a parallel job with heterogeneous tasks as a K-resource DAG, or K-DAG in short. We strive to find a schedule to complete the job as soon as possible.

Although functional heterogeneity is quite common in both client and cloud, we found little prior work on this problem. Early work on FHSs uses job-shop [3], [16], [23], [31] or DAG-shop scheduling [31] to minimize the completion time of multiple jobs. Neither of these approaches allows concurrent execution of tasks in the same job. Despite the many prior results on DAG scheduling of homogeneous systems [9], [17], [31], little quantitative research exists on how these schemes perform on functionally heterogeneous ones.

This paper presents scheduling algorithms for functionally heterogeneous systems to minimize job completion time. To achieve this goal, key insight is that a scheduler must achieve good interleaving of different types of tasks. In other words, an effective scheduler wants to balance resource utilization to keep all types of resources as busy as possible. We elaborate the importance and usage of utilization balancing in both online and offline algorithms.

This paper first describes performance and limitations of online scheduling algorithms on FHS. An online scheduler lacks knowledge of future job information such as remaining work and tasks. Let K represent the total number of resource types and P_α represent the number of type α resources where $1 \leq \alpha \leq K$. We show that any online algorithm is at best

$$K + 1 - \sum_{\alpha=1}^{K} \frac{1}{P_\alpha + 1} - \frac{1}{P_{max}}$$

competitive with respect to job completion time, where $P_{max} = \max_{\alpha=1,\ldots,K} P_\alpha$. This lower bound suggests that the relative performance of online algorithms against an offline optimal could degrade linearly with increased types of heterogeneous resources: the lack of lookahead information can prevent any online algorithm from efficiently interleaving different types of tasks and make them fail to balance the utilization of heterogeneous resources.

We present a simple online greedy algorithm — KGreedy — that provides the performance guarantee of $(K+1)$-competitiveness. This greedy scheduler is close to the best that any online algorithm can get. This carries two messages. On one side, it saves us effort in finding better online algorithms, because they can at most lead to marginal performance improvements. For less time-critical applications where $(K+1)$-competitiveness of completion time is acceptable, KGreedy offers a simple approximate. On the other side, for time-critical applications, the limitations of online scheduling motivate our use of offline/lookahead information to improve scheduling performance.

To achieve efficient offline scheduling, we propose a Multi-Queue Balancing algorithm (MQB) that minimizes job completion time by balancing the task queues of different types of heterogeneous resources to maximize task interleaving. It effectively transforms the problem of minimizing completion time to one of balancing and maximizing system utilization. If there are multiple ready tasks, MQB gives priority to those whose execution can potentially activate more descendants that can use under-utilized types of resources to achieve better balanced system utilization. We conduct extensive simulations to evaluate the performance of the online KGreedy algorithm with MQB and four other common offline heuristics. Experimental results suggest that MQB reduces the execution time of online KGreedy by more than 40% over various workloads and consistently outperforms other offline schemes. Even with limited and approximated offline information of a job, MQB can make use of them to improve scheduling decisions.

The remainder of this paper is organized as follows. Section II formulates the K-DAG scheduling problem, and Section III presents online results. We introduce our MQB algorithm and describe four other offline algorithms in Section IV. Section V presents the experimental results of comparing online and offline algorithms for various workloads. Section VI explores related work, and we offer concluding remarks in Section VII.

II. Problem Formulation

Our scheduling problem maps a job that consists a collection of tasks with varied types and precedence constraints. The processors and tasks are categorized into K types, and a task can be executed only on a processor with the matching type. For simplicity, we define the processors belonging to category α as ***α-processors***, and the tasks running on α-processors as ***α-tasks***. For each category $\alpha \in \{1, \ldots, K\}$, there are P_α α-processors in the system. This section formalizes the job model and present the optimization criteria for job completion time.

Job Model

We model the execution of a parallel job J with heterogeneous tasks as a K-DAG such that $J = (V(J), E(J))$, where $V(J)$ and $E(J)$ represent the sets of J's tasks (or nodes) and edges respectively. A K-DAG can have up to K types of tasks; each α-task v has its work (execution time), denoted as $T_1(v, \alpha)$, that can be executed only on an α-processor. The notation $V(J, \alpha)$ represents the set of α-tasks of the job. The value of ***α-work*** $T_1(J, \alpha)$ indicates the total amount of work of the α-nodes in the K-DAG, i.e., $T_1(J, \alpha) = \sum_{v \in V(J,\alpha)} T_1(v, \alpha)$. Each edge $e \in E(J)$ from nodes u to v represents a dependency between the two tasks, regardless of their types. A task

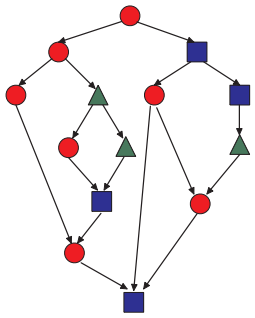


Figure 1. A job represented by a K-DAG with 3 different types of tasks drawn as circles, squares, and triangles respectively.

becomes *ready* to execute if all of its parents have been completed. The precedence relationship $u \prec v$ holds if and only if there exists a path from task u to v in $E(J)$. The *span* — or the *critical path length* $T_\infty(J)$ — corresponds to the amount of work on the longest chain of the precedence dependencies. We consider both online and offline schedulers. An offline scheduler knows in advance the entire K-DAG of a job. For online scheduling, the K-DAG is unfolded dynamically during the job's execution; at any moment, an online scheduler knows only the tasks either in execution or completed, as well as the tasks that are ready. The work of an executing or a ready task, however, is unknown to the online scheduler.

Figure 1 shows an example of a K-DAG J, with $K = 3$ represented by the shapes of the nodes. Each task has unit-size work. Let the circular, square and the triangular nodes denote α_1-tasks, α_2-tasks and α_3-tasks of the job, respectively. The work of each type is $T_1(J, \alpha_1) = 7$, $T_1(J, \alpha_2) = 4$ and $T_1(J, \alpha_3) = 3$. The span of the job is $T_\infty(J) = 7$.

Objective Function

Our scheduler uses completion time as the performance measure. The *completion time* $T(J)$ of job J under a schedule is the time at which the schedule completes J's execution.

We use *competitive analysis* to show online scheduler's performance relative to an optimal offline algorithm. Let $\mathrm{T}^*(J)$ denote the completion time produced by an optimal algorithm on job J, and let $\mathrm{T}^A(J)$ denote the completion time produced by a deterministic algorithm A for the same job. Algorithm A is said to be ***c*-*competitive*** in terms of completion time if there exists a constant b such that $\mathrm{T}^A(J) \le c \cdot \mathrm{T}^*(J) + b$ holds for any job. Similarly, let $^R(\ \ )$ denote the completion time

III. Online Scheduling

This section presents lower and upper bounds for online scheduling on an FHS. In previous work [20], He, Sun and Hsu showed that any *deterministic* online algorithm can be no better than $(K + 1 - 1/P_{max})$-competitive with respect to completion time for K-DAG scheduling, where $P_{max} = \max_{\alpha=1,\ldots,K} P_\alpha$. This section shows that any *randomized* online algorithm is at best

$$K + 1 - \sum_{\alpha=1}^{K} \frac{1}{P_\alpha + 1} - \frac{1}{P_{max}}$$

competitive. In the worst case, these two lower bounds suggest that, the completion time produced by an online algorithm degrades linearly with an increase in types of heterogeneous resources. We also present in this section an online greedy algorithm that provides the performance guarantee of $(K + 1)$-competitiveness and discuss its implications.

Performance Lower Bound

This section presents a lower bound of randomized online algorithms on completion time. We start from a technical lemma that will be used in the lower bound analysis.

Lemma 1: There are n balls in a non-transparent box. Of the n balls, r are red and $n-r$ are black. At each step, we can get one ball from the box without replacement, and each ball has the same probability of being chosen. The expected number of steps one needs to get all the red balls is $\frac{r}{r+1}(n+1)$.

Proof: Given n balls and r red ones, there are $\binom{n}{r}$ possible combinations to arrange the r red balls at n total positions, and each combination is equally likely. Let Q denote the total number of steps we need to get all the red balls for a trial. The expected value of Q is equal to the average number of steps to get all red balls over all combinations. The minimum value of Q is r because it takes at least r steps to get r red balls. The probability $\Pr\{Q = r\}$ is $1/\binom{n}{r}$ because only one combination has all red balls in the front among $\binom{n}{r}$ combinations. Similarly, we have $\Pr\{Q = r + i\} = \binom{r+i-1}{i}/\binom{n}{r}$ because $\binom{r+i-1}{i}$ combinations give $Q = r + i$, where the last red ball is at the $(r + i)$-th position and i black balls are in any of the $r + i - 1$ positions in the front.

Therefore, the expected value of Q is

$$\begin{aligned} E[Q] &= \sum_{i=r}^{n} i \times \Pr\{Q = i\} \\ &= \sum_{i=0}^{n-r} (r+i) \times \Pr\{Q = r+i\} \\ &\quad \left(\quad^{n-r} \qquad \left(\quad^{+i} \quad 1 \right) \right) \left(\quad \right) \end{aligned}$$

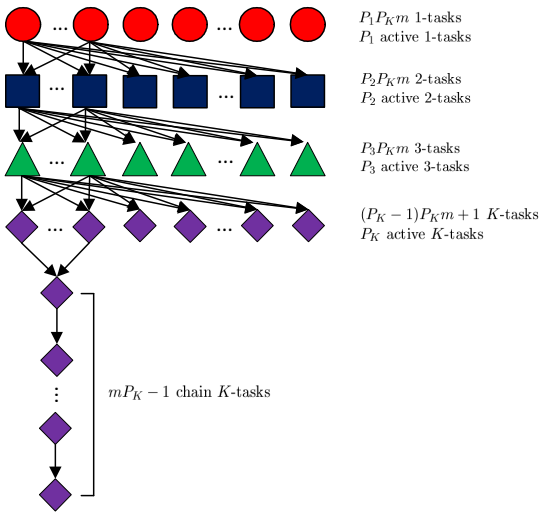


Figure 2. An example of a job instance used in the analysis of Theorem 2.

We now prove Theorem 2.

Theorem 2: The competitive ratio of any randomized online algorithm cannot be better than

$$K+1-\sum_{\alpha=1}^{K}\frac{1}{P_\alpha+1}-\frac{1}{P_{max}}$$

with respect to completion time for K-DAG scheduling, where $P_{max}=\max_{\alpha=1,...,K}P_\alpha$.

Proof: We use Yao's technique [32] in the analysis. We show a lower bound on the performance ratio of any deterministic algorithm on a job instance drawn from a probabilistic distribution. According to Yao's theorem, the same lower bound holds for all randomized algorithms.

We first describe the probabilistic distribution of jobs that we use in this proof. An example of such a job is illustrated in Figure 2. Without loss of generality, let us assume $P_K=P_{max}$. The job has $P_\alpha P_K m$ number of α-tasks for each type $\alpha=1,...,K$, where m is a positive integer constant. For α-tasks where $\alpha=1,...,K-1$, P_α of them have outgoing edges to all $(\alpha+1)$-tasks; we call these active ***α-tasks***. The remaining $P_\alpha P_K m-P_\alpha$ have no outgoing edges. The active α-tasks follow uniform random distribution on all α-tasks. For K-tasks, mP_K-1 of them form a chain, where each task but the last has one outgoing edge. These tasks are called *chain tasks*. Among the remaining $P_KP_Km-mP_K+1$ K-tasks, P_K active ones have outgoing edges to the first of the chain tasks. The remaining K-tasks do not have outgoing edges. Active K-tasks are uniformly distributed among all K-tasks that are not chain tasks. All the tasks have unit-size work.

Given a job instance J from the distribution, for $\alpha=2,...,K$, any α-tasks can get ready to run only after all the active $(\alpha-1)$-tasks have completed. The first chain tasks get ready only after all the active K-tasks have completed. We use S_α to denote the number of steps from the time any α-tasks get ready to the time all active α-tasks have completed. The job completion time is at least the summation of S_α for all $\alpha=1,...,K$ plus the time t_c taken to complete the chain. So we have

$$\mathrm{T}(J)\geq\sum_{\alpha=1}^{K}S_\alpha+t_c\ . \tag{1}$$

To schedule J, an optimal offline scheduler $\mathcal{S}$ always executes the active tasks as soon as possible. Since 1-tasks do not have any incoming edges, all of them are ready at time step 0. An optimal scheduler runs all P_1 active 1-tasks at the first time step. Then, all 2-tasks are ready at time 1. So, we have $S_1=1$. For $\alpha=2,...,K$, all α-tasks get ready at time $\alpha-1$. Once ready, the scheduler always executes active tasks first in one step, i.e., $S_\alpha=1$. K-tasks, the last type of tasks, can be completed in mP_K time steps by executing one unit of K-task on the critical path at each step and executing the remaining K-tasks on the P_K-1 other processors. Therefore, the optimal scheduler $\mathcal{S}$ produces a completion time of $\mathrm{T}^*(\mathcal{J})=K-1+mP_K$.

We now bound S_α for any online deterministic algorithm. We start from $\alpha=1$. Denote Q_1 as the total number of 1-tasks completed when the last active 1-task is done. Computing the expected value of Q_1 is analogous to computing the expected number of balls in Lemma 1 with the collection of balls equivalent to the collection of 1-tasks, i.e., $n=P_1P_Km$ and with red balls corresponding to active tasks, i.e. $r=P_1$. According to Lemma 1, the expected value of Q_1 is $E[Q_1]=(P_1/(P_1+1))(P_1P_Km+1)$. Since at each time step, at most P_1 tasks can be processed, we have

$$E[S_1]\geq E[Q_1]/P_1\geq\frac{P_1}{P_1+1}P_Km \tag{2}$$

Similarly, we can bound $S_2,...,S_{K-1}$ as $E[S_\alpha]\geq(P_\alpha/(P_\alpha+1))P_Km$. For K-tasks, since there are $P_KP_Km-mP_K+1$ non-chain K-tasks with P_K active tasks among them, we have $E[S_K]\geq(P_K/(P_K+1))(P_K-1)m$. The time t_c to complete the chain tasks is at least mP_K-1 since they have a span of mP_K-1.

According to the linearity property of expectation and Inequality (2), we can bound the expected time for any deterministic algorithm to complete J as

$$\begin{aligned}&E[\mathrm{T}(J)]\\ &\geq\ \sum_{\alpha=1}^{K}E[S_\alpha]+E[t_c]\\ &\geq\ \left(K+1-\sum_{\alpha=1}^{K}\frac{1}{P_\alpha+1}\right)mP_K-\frac{P_K}{P_K+1}m-1\ .\end{aligned}$$

Thus, the competitive ratio is given by

$$\begin{aligned}\frac{E\left[\mathrm{T}(J)\right]}{\mathrm{T}^*(J)} &\geq \frac{\left(K+1-\sum_{\alpha=1}^{K}\frac{1}{P_\alpha+1}\right)mP_K-\frac{P_K}{P_K+1}m-1}{K-1+mP_K} \\ &= \frac{K+1-\sum_{\alpha=1}^{K}\frac{1}{P_\alpha+1}-\frac{1}{P_K+1}-\frac{1}{P_K m}}{1+\frac{K-1}{mP_K}}\ . \qquad (3)\end{aligned}$$

Let $m \gg K$, then $\frac{K-1}{mP_K}$ and $\frac{1}{P_K m}$ approach 0. Since $P_K = P_{max}$, according to Inequality (3) we have

$$\frac{E\left[\mathrm{T}(J)\right]}{\mathrm{T}^*(J)} \geq K+1-\sum_{\alpha=1}^{K}\frac{1}{P_\alpha+1}-\frac{1}{P_{max}+1} \qquad (4)$$

According to Yao's theorem, the above bound holds for any randomized algorithms. ■

Theorem 2 gives a performance lower bound for any online algorithms that use K-DAG scheduling. According to the analysis, some "bad" jobs (such as those in Figure 2) prevent any online algorithms from obtaining optimal interleaving and balancing different types of tasks. At any moment, a few types of resources are busy while other types are idle. Since offline optimal algorithms can balance the utilization of different types of resources but online algorithms cannot, we see the $\Omega(K)$ times of performance difference as shown in Inequality (4).

Performance Upper Bound

In homogeneous systems, list scheduling using a greedy algorithm, one of the most well-known online algorithms, represents the first problem for which competitive analysis was presented in 1966 (by Graham [17]). Given P identical machines and a DAG job, a greedy scheduler works as follows: at any time step, if there are more than P ready tasks, the scheduler executes any P of them; if there are at most P ready tasks, it executes them all. Graham showed that this greedy scheduler was $(2-1/P)-$competitive for completion time.

In K-DAG scheduling, we consider a simple extension to this greedy scheduler that we call *KGreedy*. KGreedy uses K number of greedy schedulers, where each works for one type of resource. For example, when $K = 2$, KGreedy has two greedy schedulers working for 1-tasks and 2-tasks. At any time step, if there are more than P_1 ready 1-tasks, KGreedy executes any P_1 of them; if there are at most P_1 ready tasks, it executes them all. KGreedy performs the same greedy scheduling for type 2-tasks.

It is not hard to show that KGreedy is $(K+1)$-competitive for completion time. An extension of Graham's competitive arguments [17] would serve the purpose; refer to Theorem 3 of [20] for a complete proof. This competitive result of KGreedy matches the lower bound in Theorem 2 when there is a large number of processors for each type of resources. Therefore, the greedy scheduler performs close to the best that any online algorithm can get. It saves us effort in finding better online algorithms, which can at most lead to marginal performance improvements in the worst case. For non-time-critical applications that satisfy $(K+1)$-competitiveness of completion time, KGreedy offers a simple solution. However, for time-critical ones, the limitations of online scheduling motivated us to explore whether additional offline/lookahead information on job characteristics could help schedulers make better decisions.

IV. Offline Algorithms

This section presents offline scheduling algorithms on an FHS. Since computing optimal completion time for homogeneous DAG jobs is already NP-hard [15], K-DAG scheduling for minimum completion time is NP-hard. Therefore, our study focuses on developing efficient heuristic algorithms. We now present a new offline algorithm, called Multi-Queue Balancing (MQB), for scheduling K-DAG. MQB seeks to minimize job completion time by balancing system utilization and improving interleaving of different types of tasks. To evaluate MQB performance, we will compare it to online KGreedy and four other offline algorithms. Some of these algorithms are from well-known heuristics in homogeneous scheduling, while others drive from popular heuristics in job-shop scheduling.

A. Multi-Queue Balancing Algorithm

MQB maintains a set of K ready queues in the system, one for each type of resources. At any time, if an α-processor runs out of work, it selects a task to execute from the corresponding ready queue, namely the α-queue. MQB gives priority to tasks whose execution can potentially activate more descendants, using under-utilized resources to achieve better balanced system utilization.

Two important concepts underpin the MQB algorithm. The first is the notion of *balance* for the set of ready queues. Intuitively, we say that a snapshot A of ready queues has better balance than a snapshot B if the shortest queue of A is larger than that of B: the shortest queue will likely be the bottleneck to maximizing system utilization. In the actual algorithm, MQB also considers the number of processors for each resource. We define the *x-utilization* metric for the α-queue to be $r_{\alpha(A)} = l_{\alpha(A)}/P_\alpha$, where $l_{\alpha(A)}$ is the total work of ready tasks in α-queue at snapshot A, and P_α is the total number of α-processors. MQB compares the queues with the smallest x-utilization values to decide the balance. More precisely, let $\pi_A(\cdot)$ denote a non-decreasing order of all queues in terms of the x-utilization value, i.e., $r_{\pi_A(1)} \leq r_{\pi_A(2)} \leq \cdots \leq r_{\pi_A(K)}$. The *balance* R_A of the K ready queues at snapshot A is then given by the ordered set $R_A = \{r_{\pi_A(1)}, r_{\pi_A(2)}, \cdots, r_{\pi_A(K)}\}$. We say that the set of K ready queues at snapshot A has better balance than at snapshot B if we have $R_A > R_B$ in the

lexicographical sense, that is, there exists a $j \leq K$ such that $r_{\pi_A(j)} > r_{\pi_B(j)}$ and $r_{\pi_A(i)} = r_{\pi_B(i)}$ for all $1 \leq i < j$.

The second involves a task's *descendant value*. For each resource type α, a task in the K-DAG maintains a descendant value d_α approximating its descendant workload of type α for each $1 \leq \alpha \leq K$. MQB calculates the descendant value recursively, as follows:

$$d_\alpha(v) = 0 \text{ if task } v \text{ has no children, otherwise },$$
$$d_\alpha(v) = \sum_{u \in \{children(v)\}} (d_\alpha(u) + w_\alpha(u))/pr(u) \ ,$$

where $pr(u)$ represents the number of parents of task u, and $w_\alpha(u)$ is equal to the work of u if u is an α-task and 0 otherwise. In other words, an α-task u with $pr(u)$ parents contributes $1/pr(u)$ of its own descendant value to the descendant value of each of its parents for any resource type, plus an additional $1/pr(u)$ of its own work to d_α of each parent.

MQB works as follows. For each type α, when there are at most P_α ready α-tasks to run at any time, MQB runs them all. When there are more than P_α ready α-tasks, MQB gives priority to tasks whose execution can potentially activate more descendants to achieve better balanced utilization. Specifically, MQB assumes that the work of each ready queue can be increased by the corresponding descendant value of a ready task. It chooses the task leading to the best balance when that task's descendant values are added to the existing ready queues. MQB repeats this process until all processors have been assigned.

B. Other Heuristic Algorithms

The four other offline heuristic algorithms, whose performances we compare to MQB and KGreedy in Section V include. These four algorithms are

- Longest span first (LSpan)
- Maximum descendants first (MaxDP)
- Different type first (DType)
- Shifting bottleneck (ShiftBT)

The first two heuristics, LSpan and MaxDP, well-known for scheduling homogeneous resource, can be directly applied to heterogeneous resource scheduling by favoring tasks with the longest span and the maximum descendants, respectively. Specifically, when an α-processor runs out of tasks, it uses the following rules to select a ready α-task.

- LSpan picks an α-task with the longest remaining span. If a task does not have a child, its remaining span is its remaining work. Otherwise, its span can be computed as the sum of its remaining work and the longest span among its children.
- MaxDP picks an α-task with the largest descendant. A task without children has descendant value 0. A task u with $pr(u)$ parents contributes $1/pr(u)$ of its own descendant and its own work to the descendant of each of its parents. The descendant calculation for MaxDP resembles that for MQB. However, MaxDP does not differentiate the descendant values of different types.

While LSpan and MaxDP are straightforward extensions of the homogeneous resource scheduling heuristics without considering the presence of different resource types, the last two heuristics, DType and ShiftBT, do consider the resource heterogeneity. Specifically, they select a ready α-tasks using these rules:

- DType picks an α-task with the smallest different-child distance, where a task's different-child distance is the shortest distance to any descendant with a different type. The DType thus prioritize tasks that are parents or ancestors of tasks of other types.
- ShiftBT, an extension of the well-known shifting bottleneck heuristic [1] in job-shop scheduling, works as follows. For each resource type α, assuming all other types of resources have infinite number of processors, ShiftBT finds a schedule that tries to minimize the maximum lateness of all tasks. *Lateness* of a task measures the difference between its actual completion time and due date; the *due date*, the latest time to start a task without delaying other tasks, is computed as the total span of the job minus the remaining span of the task. The larger the lateness, the worse a task could delay the execution of other tasks. In order to find a schedule that minimizes the maximum lateness, ShiftBT uses a heuristic that always gives priority to tasks with earlier due date. For a resource type α, let's use L_α to represent the maximum lateness value of all tasks in the schedule produced by earliest due date heuristic. ShiftBT finds the resource type k that maximizes the lateness value, i.e., $k = argmax\{L_\alpha | \alpha = 1..K\}$. This resource, considered to be the biggest bottleneck resource, gets priority to be scheduled first. ShiftBT then repeats this process with the remaining resource types until all types have been considered.

All scheduling algorithms described in this paper can work in either non-preemptive or preemptive mode. A non-preemptive scheduler decides task allocation when a processor is idle, and the task runs through completion on its assigned processor without preemption. A preemptive scheduler makes decisions for each processor at the beginning of every scheduling quantum, and a task can be preempted at one processor and reallocated to another.

V. Experimental Evaluation

We conducted the following experiments to evaluate the performance of our six scheduling algorithms (KGreedy, LSpan, DType, MaxDP, ShiftBT, and MQB):

- *Algorithm performance experiments* to compare the performance of the six algorithms over different workloads
- *Changing K experiments* to investigate the impact of changing K values from 1 to 6, where K represent the number of different resource types

- *Skewed load experiments* to study the impact of biased workloads where some resources are more heavily loaded than others
- *Preemptive scheduling experiments* to compare performance of non-preemptive algorithms to their preemptive counterparts
- *Approximated information experiments* to evaluate the influence of partial and imprecise job information on scheduling

We first discuss our simulation setup (Section V-A) and workload (Section V-B) before presenting experimental results (from Section V-C to Section V-G).

A. Simulation Setup

We built a discrete-time simulator using C# to evaluate the scheduling algorithms. Composed of resources, jobs, schedulers, our simulator simulates the interactions of tasks on the resources. For preemptive scheduling, we ignore the overhead caused by processor reallocation. Our experiments assumes non-preemptive scheduling as default except for preemptive scheduling experiments. We use a default number of different resource types $K = 4$ except for changing K experiments. In all figures representing experimental results, each per bar/point plot was collected from 5000 instances of jobs.

Given a job J and an algorithm A, we are interested in the completion time generated by algorithm A comparing to an offline optimal. Since obtaining an offline optimal is NP-hard, we compare to the lower bound instead. The lower bound we use is

$$L(J) = \max\left(T_\infty(J), \max_{1\le\alpha\le K}\left(T_1(J,\alpha)/P_\alpha\right)\right) .$$

We call this performance metric the *completion time ratio*, which is calculated as the ratio of the job completion time produced by algorithm A to the lower bound, i.e., $T(J)/L(J)$.

B. Workloads and Resource Configuration

Our experiments use three types of workloads — embarrassingly parallel jobs, trees, and iterative reduction jobs. Each represents a class of parallel programming applications.

Embarrassingly Parallel Workload : An embarrassingly parallel (EP) workload is one for which little or no effort is required to separate the problem into a number of parallel branches, and no dependency exists between those parallel branches. Each branch is represented as a chain of tasks. Monte Carlo calculations and other forms of statistical simulation offer the cleanest example of an EP job, but many applications in physics, image processing, bioinformatics, etc., also fall into this general category, In a heterogeneous environment, different phases of an EP branch can be executed on different resource types.

Figure 3(a) shows an example of an EP DAG. We further divided them into two categories: (1) *layered EP* where each branch includes a fixed sequence of tasks with type from 1 to K, and (2) *random EP* where the type of each task in any branch is decided uniformly at random. To obtain an EP workload, we varied the number of branches, the number of tasks in each branch, and the work and type of each task.

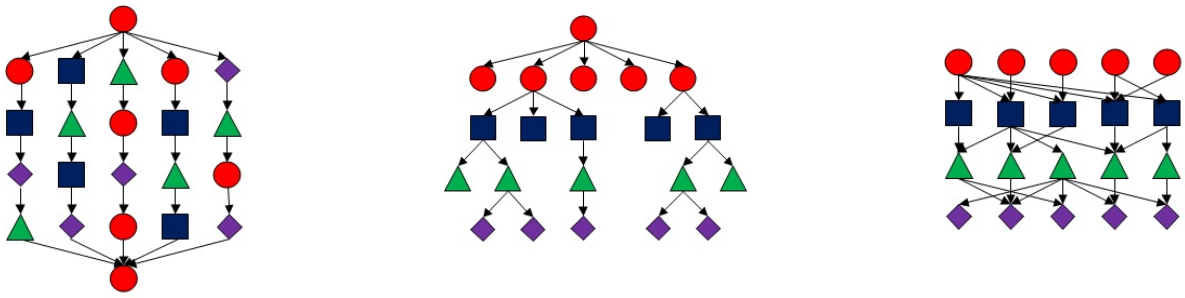
Tree workload : A tree workload starts from a root task and explores parallelism at each task by solving the task using parallel execution of subtasks. A divide-and-conquer parallel program with trivial conquer phases represents the structure of this workload. Many useful applications — such as search, graph traversal, and applications applying speculated parallelism — fall into this category.

Figure 3(b) shows an example of a tree DAG. We further divided the tree workload into two categories: (1) *layered trees* where all the nodes at each level of a tree have the same type, and (2) *random trees* where the type of each task in a tree is decided uniformly at random. A tree workload involves the fanout number m and fanout probability p of any node, i.e., a node has probability p of having m direct children and probability $1-p$ of having no children. To obtain a tree workload, we vary the fanout number, fanout probability, and the work of each task.

Iterative Reduction Workload : An iterative reduction (IR) workload resembles MapReduce [12] workload with multiple iterations of map and reduce operations. The map phase takes inputs, divides them into smaller sub-problems and runs them in parallel. All map tasks in each iteration are independent of one another. A reduce phase combines the answers of all the sub-problems to get the output. There can be many reduce tasks with each one computing the final answers over a sub-range of final results; each reduce task can depend on results from a set of map tasks. Useful in both distributed and parallel systems, MapReduce can be applied to a wide range of applications [28] such as distributed sort, inverted index construction, document clustering, machine learning, etc.

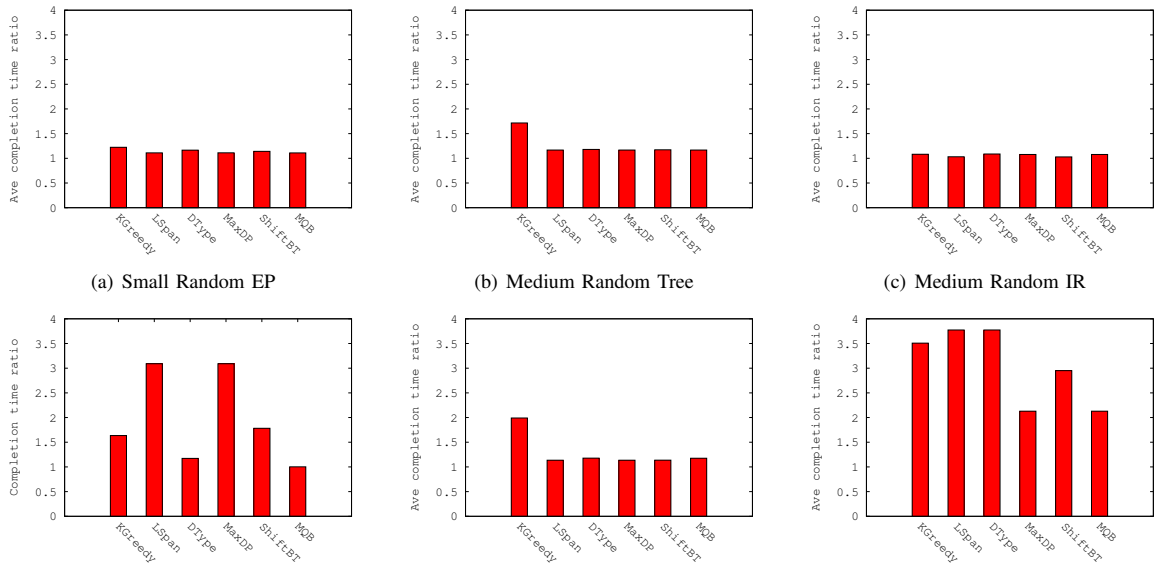
Figure 3(c) shows an example of an IR DAG. We consider the case where a reduce task depends on a subset of all map tasks. Each MapReduce iteration has map tasks with different fanouts. Tasks with a high fanout have a higher probability of providing output to each reduce task. Similarly, some reduce tasks have different fanins. To obtain an IR workload, we varied the probability values, the total number of tasks at each phase, and the work of each task. We also divided this workload into two categories: (1) *layered IR* where all nodes at each iteration of the IR computation have the same type, and (2) *random IR* where the type of each task is decided uniformly at random.

Resource Configuration : We test the workloads on small- and medium-size systems. Our small system had $1-5$ resources per type; if $K = 4$, the system was composed of $4-20$ resources in total. Our medium system had $10-20$ resources per type; if $K = 4$, it was



(a) EP Random (b) Layered Tree (c) Layered IR

Figure 3. Example K-DAG of EP, tree and IR workloads.



(a) Small Random EP (b) Medium Random Tree (c) Medium Random IR

(d) Small Layered EP (e) Medium Layered Tree (f) Medium Layered IR

Figure 4. Algorithms performance on various workload.

composed of $40-80$ resources. Each workload we present in this section combines application workload and machine configurations. For example, "medium layered IR" represents a workload running layered IR K-DAG on medium systems.

C. Algorithm Performance Experiments

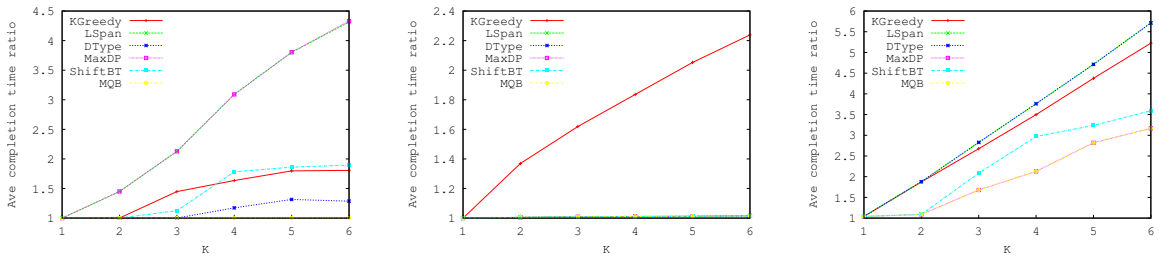
These experiments compare the completion time ratio of the six algorithms on EP, Tree and IR workloads. Figure 4 shows the results.

Figures 4(a), 4(b) and 4(c) present algorithm performance on the three randomized workloads — random EP, random tree, and random IR. The average completion time ratios of the corresponding workloads are close to 1. This suggests that KGreedy performs close to the lower bound, close to optimal, and comparably with other offline approaches. These results match our intuition: the lack of structural information makes it difficult for a scheduling algorithm to take advantage of a random model. Therefore, any "best-effort" algorithm would work just fine, and a simple greedy online algorithm is sufficient for such workloads.

Looking at more structured workloads like layered EP, layered tree and layered IR (Figures 4(d), 4(e) and 4(f)), appropriate offline information offers considerable improvements over KGreedy. Moreover, MQB outperforms KGreedy by reducing completion time ratio by at least 40% in all layered workloads. Moreover, in practice, structured programs occur more often because different tasks and stages could require or better suit different types of resources.

The following discussion about structured workloads presents the comparison results:

- Some offline information helps greatly for simple workloads. For example, all five offline heuristics reduce the average completion ratio of layered trees by half compared to online KGreedy. It is relatively easy in the tree workload to identify tasks whose execution leads to better balancing on heterogeneous resources. These tasks have more descendants, longer remaining



(a) Small Layered EP (b) Medium Layered Tree (c) Medium Layered IR

Figure 5. Performance comparison when varying the total types of resources K from 1 to 6.

spans, and smaller distances to other type of resources. An algorithm capturing any of this information works well.

- A good choice of offline heuristics can be application specific. For example, MaxDP performs well in layered tree and IR workloads, but performs poorly in layered EP ones: for EP jobs, balancing resource utilization depends mostly on the type distribution of descendants, not their number. However, this information is not captured by MaxDP. Similarly, LSpan works well for tree workloads but not for others. An algorithm performing well consistently would be highly desirable.
- A well-known and effective heuristic for job-shop scheduling ShiftBT's performance varies in K-DAG. In particular, it works well in EP and trees but not in IR. This is related to the optimality of the subcomputations on minimizing the maximum lateness. With a chain or a tree of tasks, the earliest due date heuristic that ShiftBT uses to minimize the maximum lateness leads to good solutions. However, there is no effective way to minimize lateness for more general K-DAG. When the quality of the subcomputation is compromised, ShiftBT cannot perform well consistently.
- MQB works well in most of our workloads. Compared to KGreedy and other offline heuristics, it performs always the best, or almost the best. Intuitively, it tries to balance the workload among all resources by balancing the queue sizes. It minimizes completion time by maximizing system utilization over different resource types and effectively transforms a problem of minimizing completion time into one of balancing system utilization.

D. Changing K Experiments

These experiments investigate the impact of changing the number of resource types K from 1 to 6. Figure 5 shows the results. As indicated by the worst-case performance bound in Theorem 2, increasing K linearly degrades the competitiveness of any online algorithm compared to an offline optimal. The competitive ratio of KGreedy shown in Figure 5, although obtained in average sense, nevertheless grows as K increases. The increase of average competitiveness, however, is not always linear in K because Theorem 2 is a worst-case performance bound.

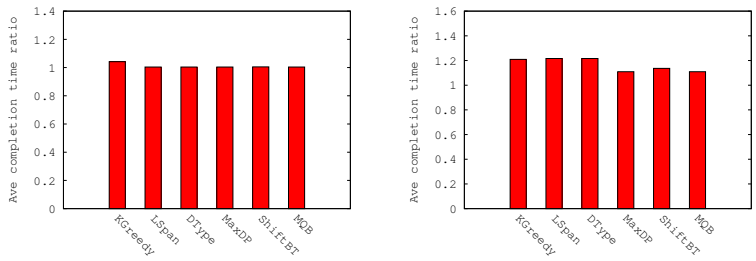
Offline information helps to reduce the performance degradation of increased K, and yields results closer to optimal even when K is large. In layered tree (Figure 5(b)), all offline algorithms perform very close to optimal at any K value. For layered EP (Figure 5(a)), MQB performs close to optimal. In layered IR (Figure 5(c)), although no algorithms perform close to the lower bound, MQB and MaxDP consistently reduce the execution time of a program produced by KGreedy by around half for any $K \geq 2$.

E. Skewed Load Experiments

These experiments study workloads where some resources are more heavily loaded than others. To quantify the resource load, we use a measure called *work-per-processor ratio*. For a job's type α resource, its α-work-per-processor ratio is computed as the α-work divided by the number of α-processors assigned to the job. When a job has similar work-per-processor ratios for each type of resource, its load is considered to be well balanced. Otherwise, the larger the variance, the more skewed the load.

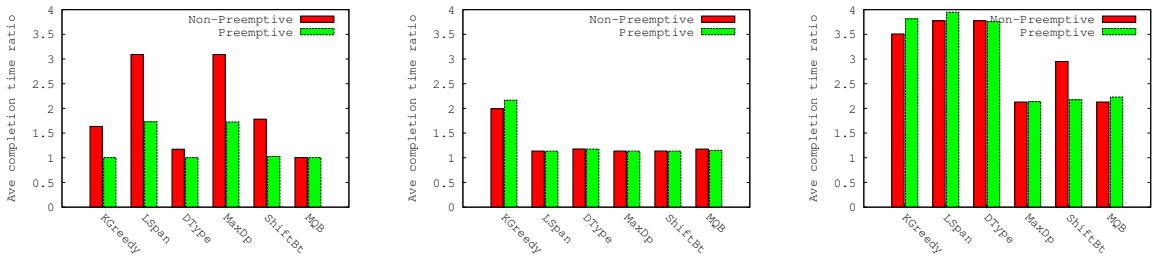
Figure 6 shows the influence of a skewed load. Using the same jobs as those tested in Figures 4(e) and 4(f), we increased the skew by reducing the number of machines for type 1 resources to $1/5$ of the original and keeping the other machines unchanged. From Figures 6(a) and 6(b), we can see that the difference among algorithms shrinks, and KGreedy performs closer to optimal. When a job's load is skewed, one or a few resource types become the bottleneck. This situation resembles a homogeneous case with the difference between algorithms becoming smaller.

Although the scheduling issue is of less concern for skewed load, such loads incur unavoidable waste on resources with small work-per-processor ratios. Assigning proper numbers of resources to a job is an aspect of scheduling decisions that goes beyond the scope of this work. However, on an FHS, achieving high resource utilization depends upon having balanced load for each job.



(a) Medium Layered Tree (b) Medium Layered IR

Figure 6. Impact of scheduling algorithms on jobs with skewed load.



(a) Small Layered EP (b) Medium Layered Tree (c) Medium Layered IR

Figure 7. Comparison of non-preemptive and preemptive scheduling.

F. Preemptive Scheduling Experiments

Previous experiments used non-preemptive scheduling; we now evaluate the impact of preemptive scheduling. Figure 7 compares the performance of both versions of the algorithms. Results suggest that, in many cases, the preemptive version performs comparably with or better than its corresponding non-preemptive one: preemptions offer earlier chances to correct bad scheduling decisions. However, preemption does not solve the performance degradation of online scheduling on heterogeneous systems since the average completion ratio of preemptive KGreedy still greatly exceeds that of good offline algorithms.

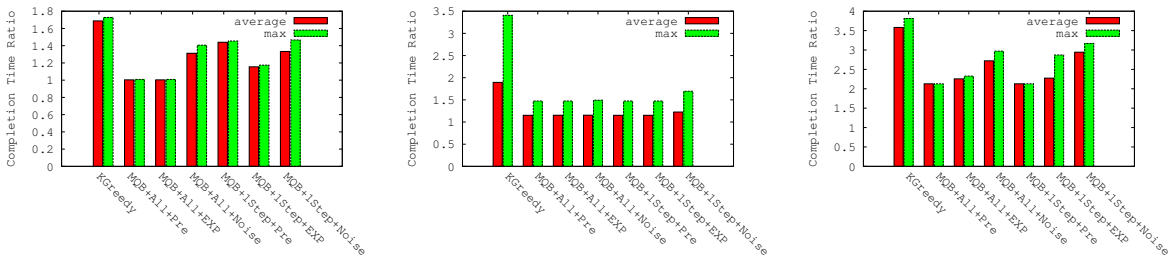
G. Approximated Information Experiments

Our previous experiments suggest that offline information helps shorten job completion time. Although complete information of a job's K-DAG may not be available during online execution, a job's previous execution history, statistics and internal characteristics can provide us an approximation on its future behavior. For example, for some data-intensive tasks, such as those in MapReduce jobs, task processing time scales linearly with the amount of data to be processed. One can collect the scale factor of a task from its previous execution and estimate the amount of data to be processed in a new run to predict task's total processing time. Detailed techniques to obtain job information through static analysis or dynamic prediction exceed scope of this work. We consider the case where estimates are available and investigate how an offline algorithm performs with approximated job information. Since MQB offers the best overall performance, we now show how MQB with partial and imprecise job information performs compared to KGreedy.

We categorize the approximation of job information into two types: (1) partial information, where one can have only limited amount of lookahead into future, and (2) imprecise information, where the approximation has noise and uncertainty. For partial information, we study a restricted case of MQB, called MQB+1Step, that has a single step of lookahead. To schedule a task, MQB+1Step considers only its immediate children. In comparison, the original MQB algorithm uses information of all task descendants, and we call it MQB+All to differentiate.

Imprecise information may come from different sources and appear in various forms such as imprecise workload calibration, user inputs, compiler outputs, etc. While it is not feasible to examine them all, we consider here two simple approximations: MQB+Exp with stochastic uncertainly and MQB+Noise with the interference of noise. In MQB+Exp, the descendant value of a task is a random value following exponential distribution with a mean equal to the true value. In MQB+Noise, the descendant value of a task is its true value interfered by a noise with a multiplicative and an additional term. The multiplicative term is a uniformly distributed random value from 0.5 to 1.5, and the additional term is a uniformly distributed random value from 0 to the average work of the task.

Combining the two cases (All/1Step) of complete or partial information with the three cases (Precise/Exp/Noise) of precise or imprecise information, we



(a) Small Layered EP (b) Medium Layered Tree (c) Medium Layered IR

Figure 8. Comparison of KGreedy and MQB with approximated information.

have six cases of MQB. Figure 8 compares average and maximum completion ratios of KGreedy to these six cases of MQB. Our comparison yields these observations:

- **Partial information**. For some workloads, a scheduler needs only a small amount of lookahead information to obtain substantial performance improvement. The results of MQB+All and MQB+1Step are similar for both trees and IR, which suggests that one-step lookahead offers sufficient information for these two workloads. This matches our intuition since in both cases a good scheduler should give priority to tasks with more children. However, the performance of MQB+1Step is worse than MQB+All at EP since a good schedule of EP can be decided only with more global job information.
- **Imprecise information**. Even with imprecise predicted or estimated information for a job's future, a fair estimate can still help improve scheduling. For example, in all workloads, both MQB+All+Exp and MQB+All+Noise offer performance benefits compared to KGreedy. Even with one-step lookahead and when approximated information can be two times off the true value, MQB still outperforms KGreedy, with 20%-30% improvements in both tree and IR workloads.

VI. Related Work

We start from classical results on scheduling homogeneous resources. The famous greedy scheduling algorithm by Graham [17] guarantees that the completion time of a DAG is no more than $2 - 1/P$ times that of an optimal scheduler, where P is the total number of processors. Shmoys, Wein and Williamson [31] showed a matching lower bound of $2 - 1/P$ on the competitive ratio of any deterministic online algorithm, and this result holds true even if preemption is allowed. They also showed a lower bound of $2 - O(1/\sqrt{P})$ for any non-preemptive randomized online algorithm. For offline scheduling, the longest span first (LSpan) algorithm has been shown to have an approximation ratio of $2 - 1/(P-1)$ for $P \geq 3$ [9]. In the special case where the DAG is an out-tree, Hu [21] showed that LSpan guarantees the optimal schedule. In K-DAG scheduling, however, one can find simple counter-examples to show that LSpan is no longer optimal for out-trees.

Many analytical results on functionally heterogeneous systems have been obtained using the job-shop scheduling model [3], [16], [23], [31]. This model has multiple jobs, each consisting of a chain of heterogeneous tasks, and there is only one machine from each resource type. For the non-preemptive setting, the best known approximation ratio is $O(\log^2 \mu P / \log^2 \log \mu P)$ [16], [31], where μ is the maximum number of tasks in a job; the best ratio for the preemptive setting is $O(\log P / \log \log P)$ [3]. Shmoys et al. [30] generalized job-shop scheduling to DAG-shop scheduling, where the precedence constraints of each job are represented by a DAG instead of a chain, and there are multiple processors for each resource type. However, no two tasks from the same job can be executed simultaneously. Similar results [16], [31] have been derived for this model. Neither job-shop nor DAG-shop scheduling allows concurrent execution of tasks in the same job.

He, Sun and Hsu [20] studied the scheduling of heterogeneous resources under the K-DAG model, which generalizes DAG-shop scheduling by allowing concurrent executions of tasks from the same type. They showed that the completion time of a K-DAG scheduled by an online greedy algorithm is $(K+1-1/P_{max})$-competitive, which matches a lower bound in the deterministic setting. Similar results have been observed in [29] under a different job model. This paper showed that randomization is of little help in improving the performances of online scheduling algorithms.

Besides theoretical results, some empirical work has studied different heuristics for scheduling functionally heterogeneous resources [1], [19]. In particular, shifting bottleneck, a popular heuristic [1], has been shown to have good performance in practice for job-shop scheduling [27]. Hamidzadeh, Lilja and Atif [19] considered both computation and communication costs of assigning a processor to a task and developed a dynamic scheduling heuristic for heterogeneous computing platforms.

VII. Conclusion Remarks

This paper described some limitations of online scheduling on functionally heterogeneous systems and proposed

an efficient algorithm — MQB that uses additional offline information to achieve utilization balancing and minimize completion time of K-DAG jobs.

Scheduling of functionally heterogeneous systems can be extended to a more general context. In a K-DAG model, each task can be executed only on its matching type of processors, which is similar to the case that a compiled binary of a task can only be executed on its matching architecture. Just-In-Time (JIT) compilation brings an interesting new dimension to this problem. With the support of JIT, a task can be compiled to different binaries at run time and flexibly executed on different types of resources. Here, a scheduler requires additional functionality and must choose appropriate resource types to compile the task for and execute it. How to schedule this more flexible job model on functionally heterogeneous systems remains an interesting open problem.

Acknowledgement

We thank Jim Larus, Burton Smith, Dennis Crain, Xenofon D. Koutsoukos, Wen-Jing Hsu and Paul Waterson for helpful discussions. We also thank anonymous reviewers for their valuable comments.

References

[1] J. Adams, E. Balas, and D. Zawack. The shifting bottleneck procedure for job shop scheduling. *Management Science*, 34(3):391–401, 1988.

[2] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris. Reining in the outliers in map-reduce clusters using mantri. In *OSDI*, October 2010.

[3] N. Bansal, T. Kimbrel, and M. Sviridenko. Job shop scheduling with unit processing times. *Mathematics of Operations Research*, 31(2):381–389, 2006.

[4] M. A. Bender and M. O. Rabin. Scheduling Cilk multithreaded computations on processors of different speeds. In *SPAA*, pages 13–21, July 2000.

[5] G. E. Blelloch and J. Greiner. A provable time and space efficient implementation of NESL. In *ICFP*, pages 213–225, 1996.

[6] R. D. Blumofe and C. E. Leiserson. Space-efficient scheduling of multithreaded computations. *SIAM Journal on Computing*, 27(1):202–229, 1998.

[7] R. Chaiken, B. Jenkins, P. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. Scope: Easy and efficient parallel processing of massive datasets. In *VLDB 2008*, August 2008.

[8] C. Chekuri and M. Bender. An efficient approximation algorithm for minimizing makespan on uniformly related machines. *Journal of Algorithms*, 41(2):212–224, 2001.

[9] N. F. Chen and C. L. Liu. On a class of scheduling algorithms for multiprocessor computing systems. In *In SCCPP*, pages 1–16, 1975.

[10] F. A. Chudak and D. B. Shmoys. Approximation algorithms for precedence-constrained scheduling problems on parallel machines that run at different speeds. In *SODA*, pages 581–590, Philadelphia, PA, USA, 1997.

[11] E. Davis and J. M. Jaffe. Algorithms for scheduling tasks on unrelated processors. *Journal of ACM*, 28(4):721–736, 1981.

[12] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. In *OSDI*, 2004.

[13] T. Endo and S. Matsuoka. Massive supercomputing coping with heterogeneity of modern accelerators. In *IPDPS*, pages 1–10, 2008.

[14] Z. Fang, P. Tang, P.-C. Yew, and C.-Q. Zhu. Dynamic processor self-scheduling for general parallel nested loops. *IEEE Transactions on Computers*, 39(7):919–929, 1990.

[15] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., 1979.

[16] Goldberg, Paterson, Srinivasan, and Sweedyk. Better approximation guarantees for job-shop scheduling. In *SODA*, 1997.

[17] R. L. Graham. Bounds on multiprocessing anomalies. *SIAM Journal on Applied Mathematics*, pages 17(2):416–429, 1969.

[18] M. Hall, Y. Gil, and R. Lucas. Self-configuring applications for heterogeneous systems: Program composition and optimization using cognitive techniques. *Proceedings of the IEEE*, 96(5):849–862, 2008.

[19] B. Hamidzadeh, D. J. Lilja, and Y. Atif. Dynamic scheduling techniques for heterogeneous computing systems. *Concurrency: Practice and Experience*, 7(7):633–652, 1995.

[20] Y. He, H. Sun, and W.-J. Hsu. Adaptive scheduling of parallel jobs on functionally heterogeneous resources. In *ICPP*, page 43, 2007.

[21] T. Hu. Parallel sequencing and assembly line problems. *Operations Research*, 9(6):841–848, 1961.

[22] S. F. Hummel and E. Schonberg. Low-overhead scheduling of nested parallelism. *IBM Journal of Research and Development*, 35(5-6):743–765, 1991.

[23] E. L. Lawler, J. K. Lenstra, A. H. G. Rinnooy Kan, and D. B. Shmoys. Sequencing and scheduling: Algorithms and complexity. Technical report, Centre for Mathematics and Computer Science, 1989.

[24] M. D. Linderman, J. Balfour, T. H. Meng, and W. J. Dally. Embracing heterogeneity — parallel programming for changing hardware. In *HotPar*, pages 1–6, 2009.

[25] M. D. Linderman, J. D. Collins, H. Wang, and T. H. Meng. Merge: a programming model for heterogeneous multi-core systems. *SIGOPS Operating Systems Review*, 42(2):287–296, 2008.

[26] G. J. Narlikar and G. E. Blelloch. Space-efficient scheduling of nested parallelism. *ACM Transactions on Programming Languages and Systems*, 21(1):138–173, 1999.

[27] M. L. Pinedo. *Scheduling: Theory, Algorithms, and Systems*. Springer (Third edition), 2008.

[28] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. In *HPCA*, pages 13–24, 2007.

[29] H. Shachnai and J. J. Turek. Multiresource malleable task scheduling to minimize response time. *Information Processing Letters*, 70(1):47–52, 1999.

[30] Shmoys, Stein, and Wein. Improved approximation algorithms for shop scheduling problems. In *SODA*, 1991.

[31] D. B. Shmoys, J. Wein, and D. P. Williamson. Scheduling parallel machines online. In *FOCS*, pages 131–140, 1991.

[32] A. Yao. Probabilistic computations: Toward a unified measure of complexity. In *FOCS*, 1977.