A Divide and Conquer Algorithm for DAG Scheduling under Power Constraints

Gökalp Demirci
Department of Computer Science
University of Chicago
Chicago, USA
demirci@cs.uchicago.edu

Ivana Marincic
Department of Computer Science
University of Chicago
Chicago, USA
imarincic@cs.uchicago.edu

Henry Hoffmann
Department of Computer Science
University of Chicago
Chicago, USA
hankhoffmann@cs.uchicago.edu

***Abstract*—We consider the problem of scheduling a parallel computation—represented as a directed acyclic graph (DAG)—on a distributed parallel system with a global resource constraint—specifically a global power budget—and configurable resources, allowing a range of different power/performance tradeoffs. There is a rich body of literature on the independent problems of (1) scheduling DAGs and (2) scheduling independent applications under resource constraints. Very little, however, is known about the combined problem of scheduling DAGs under resource constraints. We present a novel approximation algorithm using a divide-and-conquer method for minimizing application execution time. We prove that the length of the schedule returned by our algorithm is always within O($\log n$)-factor of the optimum that can be achieved with any selection of configurations for the tasks. We implement and test our algorithm on simulations of real application DAGs. We find that our divide-and-conquer method improves performance by up to 75% compared to greedy scheduling algorithms.**

***Index Terms*—scheduling, DAG, precedence, power, resource, configuration**

I. Introduction

Future High Performance Computing (HPC) systems are expected to maximize workload performance subject to strict power constraints [1]–[3]. These workloads are effectively modeled as directed acyclic graphs (DAGs) that capture precedence requirements between tasks. DAG representations describe diverse workloads such as scientific computing applications (e.g. representing data flow between MPI calls), scheduling assignments of HPC jobs, and data-driven scientific workflows managed by frameworks like Spark. Scheduling DAGs for minimum execution time (without power constraints) is an old and well-understood problem. Likewise, scheduling independent workloads (without precedence) under power constraints has been gaining more focused attention in the last few years. However, the combined problem of two restricted special cases—resource (i.e. power) and precedence constrained (i.e. DAG) scheduling—is not well understood. Further complicating the problem, future HPC systems are expected to be highly configurable, allowing them to operate at a range of power and performance tradeoffs by tuning hardware or system software properties [4]. Distributed scheduling algorithms must take advantage of these configurations to minimize DAG execution time while respecting the system-wide power constraint.

Intuitively, minimizing DAG execution time under a power cap is the problem of packing tasks into an open-ended strip. The strip's height is the power cap; the length it reaches is the execution time to be minimized. On modern computing systems, we can configure the combination of a task (DAG node) and machine to operate at different power/performance tradeoffs. Thus, each task can be represented by different rectangles whose heights are the individual task's power and the length is the runtime. Even before we consider precedence constraints, this packing problem is known to be NP-hard [5]. Intuitively, to minimize the strip's length (i.e. overall execution time), one should minimize the total uncovered area of the strip and also minimize the areas of each task's rectangle. In other words, we can approach the problem by

1) *Trying to maximize power utilization*, so that at any time instant as much work as possible is being accomplished and the individual rectangles represented by the scheduled tasks are as close to the power limit as possible; or
2) *Trying to maximize energy efficiency (or task performance over power)*, so that each task's individual rectangle has the smallest possible area.

The first strategy favors configurations that correspond to tall, skinny rectangles for individual tasks, while the second strategy favors minimal area rectangles.

Greedy scheduling strategies can easily be developed to favor one of these approaches. For example, a simple greedy strategy that attempts to maximize power utilization would wait for a machine to go idle and just select the next largest task and configuration that takes up as much of the remaining power budget as possible. In particular, greedy algorithms for the problem we consider can be obtained by taking a state-of-the-art greedy algorithm satisfying either the precedence constraint or the power constraint and simply extending it to handle the other as well. We find that such greedy algorithms achieve poor results in practice for the combined problem of DAG scheduling under power constraints and that *the key insight to efficient scheduling is to find a good tradeoff between maximizing power utilization and maximizing individual tasks' energy efficiency.*

Based on this insight, we propose a divide-and-conquer algorithm for scheduling DAGs under power constraints when

SC18, November 11-16, 2018, Dallas, Texas, USA
978-1-5386-8384-2/18/$31.00 ©2018 IEEE

power/performance tradeoffs are configurable. The key principle of this divide-and-conquer algorithm is: *to concurrently schedule tasks whose completion time (rectangle length) is similar when they run in their most energy-efficient configuration.* This principle allows our algorithm to achieve higher power utilization without deviating too far from executing tasks in an energy efficient manner (keeping individual task rectangles small). Our divide-and-conquer approach extends prior work that considers a more restricted scheduling problem in which the tasks have fixed power and performance characteristics [6]. In this paper, we consider the harder version of working with a discrete configuration set involving different power/performance trade-offs per configuration per task.

This paper makes the following contributions:

- A divide-and-conquer algorithm based on the principle of finding and co-scheduling tasks with similar execution times in their most energy-efficient configurations. Specifically, we extend prior work [6] to intelligently incorporate configuration selection. We then improve this algorithm with a randomized variant.
- A proof that the algorithm is a $(2+\log n)$-approximation for the problem where n is the number of tasks; i.e., the algorithm returns a schedule whose length is always within a factor $2+\log n$ of the optimum.
- Discussion of variations on the greedy algorithms that form the basis for today's state-of-the-art schedulers with an empirical demonstration that the divide-and-conquer algorithm is superior.

We evaluate our algorithm using both real-world and synthetic DAGs of up to 10,000 nodes using a simulator[1]. Our algorithm is DAG-agnostic and capable of scheduling DAGs of any shape and size. It outperforms greedy approaches especially well in severely power-restricted systems with significant energy savings and improving performance up to 75%.

We use simulation infrastructure for this evaluation because our goal is to compare fundamentally different algorithmic strategies (greedy versus divide-and-conquer), rather than to evaluate practical implementation issues. All modern job management systems (of which we are aware) are based on greedy strategies. Performance-aware schedulers minimize execution time for DAGs—on both distributed (e.g. Spark [7] and MapReduce [8]) and multi-core systems (e.g. Cilk++ [9] and TBB [10])—based on greedy strategies derived from that first proposed and analyzed by Graham [11]. State-of-the-art power-aware schedulers also use greedy schedules to maximize throughput for independent tasks [12]–[15]. Whether constrained by precedence or power, all these recent scheduling efforts share a common algorithmic foundation: *they adopt greedy strategies and innovate for practical concerns*. In this paper, we present a fundamentally different algorithmic strategy: divide-and-conquer. Our simulation-based evaluation factors out implementation concerns to focus on the potential benefits of divide-and-conquer over the greedy strategy that underlies all known schedulers.

[1]Available at: https://github.com/PowerCapDAG/PowerCapDagScheduling

II. Related Work

The problem of power-aware scheduling has recently attracted a lot of interest. In this paper, we consider the problem of configuring multiple processors to maximize performance under a power constraint in a multiprocessor setting where there are precedence constraints between tasks. We first discuss related work in energy-aware scheduling and then the most closely related work: scheduling *malleable tasks*.

A. Power-aware Scheduling

In the scheduling literature, most power-aware algorithms assume the only configurable part of the system is processor speed. They then model the speed scaling using the standard assumption proposed by Yao, Demers, and Shenker (YDS) [16]: there is a continuous *power function* $P(s)$, such that a processor running at speed s has power consumption $P(s) = s^\alpha$ for some $\alpha > 1$. Unfortunately, this assumption ignores the fact that real processors have both minimum and maximum speeds; i.e., in real computing systems one cannot arbitrarily increase energy efficiency $(s/P(s))$ by slowing down computation [17].

Nevertheless, many researchers have proposed extensions and improvements of the YDS algorithm. These include algorithms that account for non-zero idle-power states [18] and algorithms that consider zero power sleep states that require additional energy and time to restart computation [19]. Numerous other improvements on the original YDS algorithm exist in the scheduling literature [20]–[28].

Most approaches, however, schedule only independent jobs and maintain the YDS assumption about the relationship between power and speed. Pruhs, van Stee, and Uthaisombut [29] consider speed scaling to minimize total energy for multiprocessor settings with precedence constraints between tasks. Bunde assumes that jobs arrive over time and considers continuous power functions that are convex, strictly convex, or discrete [30]. He develops algorithms that minimize the makespan for uniprocessor and multiprocessor settings where the processors have a shared energy budget. This technique is not applicable to our multiprocessor setting with a power bound, rather than an energy bound—i.e., a bound on the maximum *rate* of energy consumption rather than total energy consumption—as the combinations of such configurations may violate the power constraint.

Power bounds, however, are essential for future generation high-performance computing systems as noted by several recent studies [1]–[3]. Recent heuristic schedulers account for power by attempting to predict a workload's critical path and reduce power where it will not affect execution time [31], [32]. Patki et al. propose building HPC systems that are specifically *over-provisioned* and could easily draw much more power than is safe to deliver to the system at once [33]. Sarood et al. have shown similar results: hardware over-provisioning increases performance given a power cap [34]. These hardware over-provisioning approaches acknowledge that compute resources are no longer the primary factor limiting cluster size—power is. Furthermore, these approaches require implementing

severely restrictive power caps, where the system-level power budget is significantly below the maximum total power draw of the available machines. This scenario is precisely the realm where our proposed divide-and-conquer algorithm provides the biggest advantages over greedy approaches.

As existing scheduling implementations are heuristic in nature, Bailey et al. recently proposed a mathematical optimization framework for analyzing job schedules under a power constraint [35]. This formulation considers configurability, power constraints, and precedence constraints, but it is an offline algorithm and requires imposing a total order on the DAG representing the workload to be scheduled. For these reasons, the Bailey et al. approach is best suited for post hoc analysis of heuristic schedules, to determine how far they deviate from optimal. In contrast, our divide-and-conquer algorithm works directly on the partial order of the DAG and is suitable to be used as a scheduler with a time complexity comparable to greedy algorithms.

B. Scheduling Malleable Tasks

Malleable task scheduling with precedence constraints resembles the problem we consider here in all three aspects—configurability, resource limits, and precedence constraints. For malleable tasks, each task is allotted a number of 1 to m processors, where m is the total available processors. A task's execution time is a function of the processors allotted. A feasible schedule ensures that co-scheduled tasks are allotted no more than m machines while respecting the DAG dependencies. It is often assumed that tasks' execution times are non-increasing and the work (the number of processor times the execution time) is non-decreasing with increasing processors. Under such assumptions, the problem admits constant-approximations [36]–[38].

While similar, malleable task scheduling differs from our problem in several key ways. Most importantly, we model the resource (power) as a continuous variable; at any point in a feasible schedule the tasks' power requirements may sum to any real number between 0 and the power cap. In contrast, in malleable task scheduling, there are only m states a schedule can be in at any point. This discrete modeling of resource (together with the assumptions on the function of execution time) is the key for obtaining a constant-approximation for the malleable task scheduling with precedence constraints. The continuous nature of the resource in our problem renders the same methods less useful. Moreover, Demirci et al. [6] point to an instance of our problem which shows that simple lower bound analysis methods and straightforward extensions of LPs designed for precedence constrained scheduling all have an $\Omega(\log n)$ gap. Nevertheless, we find that a method previously used in scheduling independent malleable tasks [39] to be useful in proving an $O(\log n)$ approximation bound for our algorithm (see Section III-C).

The malleable task problem clearly differs from that considered in this work from a systems point of view as well. Malleable task scheduling reduces configurability to simply the number of processors, which determines a task's execution time. In contrast, we do not impose any assumptions on the machine/task pairs' configuration spaces. Additionally, our configurations are not necessarily directly comparable with each other (but comparable in terms of their resource–power–requirement). For example, one configuration may correspond to four active cores on a lower DVFS setting with no hyperthreading and two memory controllers and another one may correspond to two active cores on a higher DVFS setting with hyperthreading and a single memory controller (see Section V-A).

Recently, Demirci, Hoffmann, and Kim (DHK) proposed the first approximation algorithm with non-trivial bounds for scheduling DAGs under power constraints [6]. This algorithm solves a restricted, theoretical version of DAG scheduling under a power constraint—specifically, when tasks have predetermined power requirements and execution times. That is, DHK does not consider configurable power/performance tradeoffs, available on all real systems. In this work, we modify the DHK algorithm to cover configurable tradeoffs. *Our modification and extension allow us to prove that DAG scheduling under a power cap on configurable systems has a* $2 + \log n$ *approximation.* Note that they also allow us to slightly improve the approximation bound of $2 + 2\log(n+1)$ given in [6] for the non-configurable version of the problem.

III. The Divide-and-Conquer Strategy

We formalize our problem statement, discuss the divide-and-conquer strategy, prove an upper bound on the approximation, and then present a randomized variant of divide-and-conquer.

A. Problem Formulation

Our problem is stated formally as follows. We want to schedule a set of tasks $\mathcal{T}$ on m machines given a DAG representing the dependencies between tasks. Each task $t \in \mathcal{T}$ running on machine i has a discrete configuration set $\mathcal{C}_{t,i}$ associated with the task-machine pair. Each configuration $c \in \mathcal{C}_{t,i}$ has a power requirement $\mathrm{p}(c)$ (beyond the machine's idle power) and an execution time $\mathrm{t}(c)$ representing the power/performance trade-off of running task t on machine i in configuration c. Let P denote the *power cap*: the total power beyond idle that the running tasks can consume at any time point. There are two special configurations: the fastest *race* configuration, i.e. $c_{t,i}^{race} = \operatorname{argmin}_{c \in \mathcal{C}_{t,i}} \mathrm{t}(c)$, and the most energy-efficient *pace* configuration, i.e. $c_{t,i}^{pace} = \operatorname{argmin}_{c \in \mathcal{C}_{t,i}} \mathrm{t}(c)\mathrm{p}(c)$. In the rest of the paper, we assume that the machines are identical, so the configuration set depends only on the task and is denoted as $\mathcal{C}_t$. A feasible schedule needs to choose a configuration $c \in \mathcal{C}_t$ for each task t, dedicate $\mathrm{p}(c)$ amount of the power budget to it, and run t on one of m machines for $\mathrm{t}(c)$ amount of time without interruption (i.e. we do not allow preemption).

Our formulation yields an optimization problem with two objectives: maximize power utilization and maximize energy efficiency (of individual configurations). We propose a unifying principle that optimizes these objectives simultaneously: to concurrently schedule tasks with similar completion times

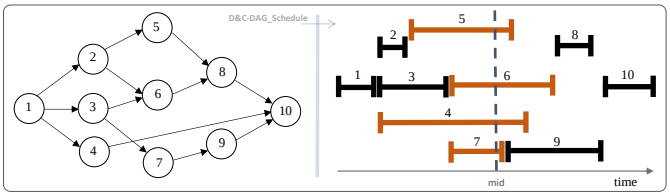

Fig. 1. The intermediate schedule (right) obtained in the first step by **D&C-DAG_Schedule** from the DAG (left). The intermediate schedule satisfies the DAG dependencies, but not necessarily the power constraints. **D&C-Recursive** takes this intermediate schedule and groups and schedules the tasks crossing its mid-point (orange) in parallel. It then recursively obtains schedules for the tasks remaining on the first half and second half of the intermediate schedule.

in their pace configuration. In this section, we give a high-level overview of our divide-and-conquer algorithm, explain why it is an excellent candidate for implementing this unifying principle, and provide pseudocodes for its main subroutines. The algorithm works in two steps: 1) **D&C-DAG_Schedule** and 2) **D&C-Recursive**.

B. The Divide-and-Conquer Algorithm

Subroutine 1 D&C-DAG_Schedule$(\mathcal{T}, m, G, time(\cdot))$

Input: A set of tasks $\mathcal{T}$ with execution times $time(\cdot)$, a DAG $G = (\mathcal{T}, E)$ on these tasks, and m machines.
Output: An intermediate schedule IS of the tasks respecting the DAG.

$ReadyTasks \leftarrow \{t \in \mathcal{T} : \nexists t' \in \mathcal{T} \text{ s.t. } (t', t) \in E\}$
$ReadyMachines \leftarrow \{i : 1 \leq i \leq m\}$
$time \leftarrow 0$
while $\mathcal{T} \neq \emptyset$ **do**
  if $ReadyTasks \neq \emptyset$ and $ReadyMachines \neq \emptyset$ **then**
    $t \leftarrow ReadyTasks[1]$ and $i \leftarrow ReadyMachines[1]$
    Assign task t on machine i:
    $IS \leftarrow IS \cup \{(t, i, time, time + time(j))\}$
    $ReadyTasks \leftarrow ReadyTasks \setminus \{t\}$
    $ReadyMachines \leftarrow ReadyMachines \setminus \{i\}$
  else
    $(endT, endM, beg, end) \leftarrow \underset{(t,i,s,f) \in IS \,:\, f \geq time}{\operatorname{argmin}} f$
    $ReadyMachines \leftarrow ReadyMachines \cup \{endM\}$
    $\mathcal{T} \leftarrow \mathcal{T} \setminus \{endT\}$
    $ReadyTasks \leftarrow \{t \in \mathcal{T} : \nexists t' \in \mathcal{T} \text{ s.t. } (t', t) \in E\}$
    $time \leftarrow end$
  end if
end while
return IS

D&C-DAG_Schedule (Subroutine 1) produces an intermediate schedule satisfying the DAG dependencies. We take the task run-times in their pace configurations and produce a schedule with no restrictions on the power or available machines. This intermediate schedule may have more than m tasks running in parallel and it is essentially a "schedule representation" of the DAG with respect to the pace configurations. Figure 1 illustrates the DAG of a small example on the left and the respective intermediate schedule on the right.

Subroutine 2 D&C-Recursive$(T, m, P, \{\mathcal{C}_t\}_t, IS, beg, end)$

Input: A set of tasks $T \subseteq \mathcal{T}$ each with its configuration space $\mathcal{C}_t$, m machines, a power cap P, and an intermediate schedule IS with beginning and end time points beg, end.
Output: A schedule S of the tasks respecting the DAG and the power cap.

if T is $\emptyset$ **then**
  return Empty schedule
end if
Let mid be a time-point s.t. $|T_{bef}| \leq |T|/2$ and $|T_{aft}| \leq |T|/2$ w.r.t. definitions below
$T_{bef} \leftarrow \{t \in T : beg < IS_t.s < IS_t.f \leq mid\}$
$T_{mid} \leftarrow \{t \in T : beg < IS_t.s \leq mid < IS_t.f \leq end\}$
$T_{aft} \leftarrow \{t \in T : mid < IS_t.s < IS_t.f < end\}$
$S_{bef} \leftarrow$ **D&C-Recursive**$(T_{bef}, m, P, \{\mathcal{C}_t\}_t, IS, beg, mid)$
$time \leftarrow \max_{t \in T_{bef}} S_{bef}(t).f$
$S_{mid} \leftarrow$ **Schedule-Independent**$(T_{mid}, m, P, \{\mathcal{C}_t\}_t)$
Offset the times in S_{mid} by $time$.
$time \leftarrow \max_{t \in T_{mid}} S_{mid}(t).f$
$S_{aft} \leftarrow$ **D&C-Recursive**$(T_{aft}, m, P, \{\mathcal{C}_t\}_t, IS, mid, end)$
Offset the times in S_{aft} by $time$.
return $S_{bef} \cup S_{mid} \cup S_{aft}$

D&C-Recursive (Subroutine 2) converts the intermediate schedule into a final schedule satisfying the power constraints. This algorithm considers the tasks crossing the mid-point of the intermediate schedule. These tasks have similar run-times and will be scheduled in parallel separate from the rest of the tasks in the final schedule. Since the precedence-aware intermediate schedule has all these tasks running in parallel at the mid-point, they cannot have dependences among them. Thus, what remains is to satisfy the power constraints when scheduling this subset of tasks. **D&C-Recursive** then applies the same technique on the remaining tasks by two recursive calls on two halves of the intermediate schedule.

D&C-DAG_Schedule's intermediate schedule may have more than m tasks crossing the mid-point. **D&C-Recursive** uses another subroutine **Schedule-Independent** that sorts these tasks in order of their execution times and schedules them to run in this order in groups of at most m. (We describe **Schedule-Independent** in detail in Section III-C.) Thus, the algorithm has two main tools for grouping tasks with similar execution times. First, it is more likely that a point in the middle of the intermediate schedule will be crossed by relatively "longer" tasks than by a diverse group of tasks containing, in particular, both "short" and long tasks. Second, these tasks are sorted with respect to execution times and considered for groups in this particular order to separate long tasks from the shorter ones as much as possible. To justify the first point, that it is more likely for a longer task to cross a point in the middle, suppose we place a long and a short task uniformly at random within a finite time horizon. For an ϵ close to 0 and smaller than the lengths of these tasks, the

Algorithm 1 **D&C**$(\mathcal{T}, G, m, P, \{\mathcal{C}_t\}_t)$

Input: A set of tasks $\mathcal{T}$ each with its configuration space $\mathcal{C}_t$, a DAG G, m machines, and a power cap P

Output: A schedule S of the tasks respecting the DAG and the power cap

1: Get pace configurations $c_t^{pace} \in \mathcal{C}_t$ for each task t
2: $IS \leftarrow$ **D&C-DAG_Schedule**$(\mathcal{T}, G, |\mathcal{T}|, \{\mathrm{t}(c_t^{pace})\}_t)$
3: $end \leftarrow \max_{t \in IS} IS(t).finish$
4: $S \leftarrow$ **D&C-Recursive**$(\mathcal{T}, m, P, \{\mathcal{C}_t\}_t, IS, 0, end)$
5: **return** S

probabilities that each of these tasks will cross the time point ϵ are equal and independent of the lengths of the tasks. However, the probability that such a randomly placed task will cross a point in the middle of the time horizon (sufficiently far from the both endpoints) is proportional to the task's length. Thus, a greedy algorithm—that increments the time-line starting from 0—will encounter a variety of tasks in each time increment, whereas our divide-and-conquer method is more likely to first cut the longer tasks in the first call and cut relatively shorter and shorter tasks deeper in the recursion.

Consider the example in Figure 1. A greedy algorithm starting with task 1 would next schedule task 2—the shortest task—together with 3 and 4—the longest task. While a greedy algorithm will often co-schedule both short and long tasks, it is more likely that a point sufficiently far from both ends of the intermediate schedule will be crossed by relatively longer tasks. Note that the remaining tasks' lengths become shorter with deeper recursion on both sides. Thus, the first level of the recursion cuts mostly the relatively longer tasks, and the recursive calls on the lower levels of the recursion tree work on the smaller tasks contained in tiny fragments of the intermediate schedule. Finally, we get a fine-grained grouping of the tasks with respect to the length of their pace execution times. Note that **D&C-DAG_Schedule** and **D&C-Recursive** take the pace configurations as the point of reference (either for execution time or for power) because, in the end, we would like to schedule all the tasks close to (or ideally in) their pace configurations as energy efficiency is one of the objectives for minimizing the total run-time under a power-cap.

D&C (Algorithm 1) is the main subroutine which uses **D&C-DAG_Schedule** and **D&C-Recursive** to obtain the final schedule. It first calls **D&C-DAG_Schedule** to obtain an intermediate schedule. Then **D&C** calls **D&C-Recursive** with the full set of tasks $\mathcal{T}$, number of machines m in the original problem, the power cap P, the intermediate schedule IS, and the beginning and end times of IS. **D&C-Recursive** recursively schedules the tasks respecting the power cap and returns this schedule.

C. An $O(\log n)$ Approximation Algorithm

We complete the algorithm given in Section III by describing **Schedule-Independent** and prove that our algorithm is always within a $2 + \log n$ factor of the optimal solution by selecting configurations that are close enough to the configurations employed in an optimal solution. We modify and extend the DHK proof that solves the problem for fixed configurations [6] by carefully adapting a method (inspired by [39]) to argue that the configurations chosen by our algorithm are comparable to the configurations in an optimal solution.

The DHK algorithm has fixed configurations (and, therefore, execution times), so its intermediate schedule is recursively partitioned around the mid-point of the intermediate schedule's length. This strategy guarantees that the algorithm recurses on equally partitioned halves so the recursion depth is bounded by $O(\log n)$. **D&C-Recursive**, however, cannot follow the same steps because the length of the intermediate schedule obtained using the pace configurations is not necessarily a lower bound on the optimal schedule length—which may use configurations other than pace for the same tasks.

Thus, we use another method to limit the depth of the recursion. Instead of equally partitioning with respect to intermediate schedule length, **D&C-Recursive** partitions the tasks with respect to the number of tasks left on each side. This goal motivates the particular definition in Line 4 of **D&C-Recursive**. Now, we prove that the intermediate schedule has at least one point that guarantees that both sides contains at most half of all tasks.

Lemma 1. *There is at least one time point mid such that, when **D&C-Recursive** partitions the tasks in intermediate schedule around mid, $|T_{bef}| \leq |T|/2$ and $|T_{aft}| \leq |T|/2$.*

Proof: Consider a sweep starting from the end point of the intermediate schedule towards its starting point and the first critical point cp where $|T_{bef}| \leq |T|/2$. Immediately to the right of cp, let the size T_{bef} be $\lfloor |T|/2 \rfloor + i$ for some $i \geq 1$. On cp, $|T_{mid}| \geq i$ and $|T_{bef}| = \lfloor |T|/2 \rfloor + i - |T_{mid}|$. Then, $|T_{aft}| = |T| - |T_{mid}| - |T_{bef}| = |T| - |T_{mid}| - \lfloor |T|/2 \rfloor - i + |T_{mid}| = \lceil |T|/2 \rceil - i \leq |T|/2$. Since there are $O(|T|)$ critical points, such a sweep can be done efficiently. ∎

A simple subroutine called **NFDH**—used and analyzed for strip packing [5]—is directly adapted by DHK for scheduling the independent tasks in T_{mid}. It greedily schedules the tasks in T_{mid} in the order of non-increasing execution times on "shelves" respecting the power cap. If the addition of the next task in this order makes the current shelf violate the power cap, the subroutine closes the current shelf and opens another containing this next task. This subroutine guarantees a schedule of length proportional to the sum of energy requirements of the tasks in T_{mid} plus the maximum execution time among all the tasks in T_{mid}. In our case, however, power and execution time both depend on a task's configuration. To use the same subroutine to schedule independent tasks in T_{mid}, we need to make the configuration selection carefully so that we can relate the configurations our algorithm selects to the configurations selected for the same tasks in an optimal solution.

Let $t_1, t_2, \cdots, t_k$ be the tasks in T_{mid}. Let $c_1^*, c_2^*, \cdots, c_k^*$ be their corresponding configurations in an optimal solution with a makespan of length OPT. Note that, in an optimal solution, these tasks do not necessarily run together. We define a procedure that produces a set of configurations $\{c_i\}_{1 \leq i \leq k}$

for these tasks with two specific guarantees:

1) $\max_{1\le i\le k} \mathrm{t}(c_i) \le \max_{1\le i\le k} \mathrm{t}(c_i^*)$
2) $\mathrm{t}(c_i)\mathrm{p}(c_i) \le \mathrm{t}(c_i^*)\mathrm{p}(c_i^*), \quad \forall 1 \le i \le k$

Note that these two guarantees correspond directly to the two intuitive goals specified in the introduction. The first guarantee corresponds to the desire to maximize power utilization and minimize execution time (configure tasks for tall, narrow rectangles). The second guarantee corresponds to the desire to minimize each tasks' energy (configure all tasks for their pace configurations, for minimal area rectangles).

The procedure works iteratively, producing a different set of configurations in each iteration by modifying those from the previous iteration. It starts with the pace configurations in the first iteration $\{c_i^1 = \mathrm{argmin}_{c\in\mathcal{C}_{t_i}} \mathrm{t}(c)\mathrm{p}(c)\}_{1\le i\le k}$. In iteration $j \ge 2$, the configurations are kept the same as the previous iteration $c_i^j = c_i^{j-1}$ for all tasks except that with the longest execution time. The longest executing task is assigned (if possible) a shorter configuration that minimizes the energy among all such configurations. Formally, let $i_{max} = \mathrm{argmax}_{1\le i\le k} \mathrm{t}(c_i^{j-1})$ and, then,

$$c_i^j = \begin{cases} \underset{c\in\mathcal{C}_{t_i}, \mathrm{t}(c)<\mathrm{t}(c_i^{j-1})}{\mathrm{argmin}} \mathrm{t}(c)\mathrm{p}(c) & \text{if } i = i_{max} \\ c_i^{j-1} & \text{otherwise} \end{cases}$$

The procedure produces a new set of configurations until there is no shorter configuration for the longest task. Since we do not know the configurations $\{c_i^*\}_i$ in the optimal schedule, we have no straightforward way of checking if both Conditions 1 and 2 are satisfied by $\{c_i^j\}_i$ at some iteration j. Thus, we run the procedure until no new iterations are possible.

Lemma 2. *The above procedure produces at least one set of configurations satisfying Conditions 1 and 2 at the same time.*

Proof: The configurations of the first iteration $\{c_i^1\}_i$ satisfy Condition 2 by definition. If the procedure never produces a set of configurations violating Condition 2, then the last set of configurations also satisfy Condition 1. This condition is true because the longest running task has no other shorter configuration which means the same task in an optimal solution cannot get a shorter configuration either. Then, let j be the first iteration to violate Condition 2. It is not hard to see that the configurations $\{c_i^{j-1}\}_i$ of $j-1^{st}$ iteration satisfy Condition 1 as well. Let $i_{max} = \mathrm{argmax}_{1\le i\le k} \mathrm{t}(c_i^{j-1})$. Given the facts that $\mathrm{t}(c_{i_{max}}^*)\mathrm{p}(c_{i_{max}}^*) < \mathrm{t}(c_{i_{max}}^j)\mathrm{p}(c_{i_{max}}^j)$ and $c_{i_{max}}^j = \underset{c\in\mathcal{C}_{t_i}, \mathrm{t}(c)<\mathrm{t}(c_i^{j-1})}{\mathrm{argmin}} \mathrm{t}(c)\mathrm{p}(c)$, it cannot be the case that $\mathrm{t}(c_{i_{max}}^*) < \mathrm{t}(c_{i_{max}}^{j-1})$ (because, otherwise, $c_{i_{max}}^j$ would be assigned $c_{i_{max}}^*$). Then $\mathrm{t}(c_{i_{max}}^{j-1}) \le \mathrm{t}(c_{i_{max}}^*) \le \max_{1\le i\le k} \mathrm{t}(c_i^*)$. ∎

This procedure's iteration count is limited by $\sum_{1\le i\le k} |\mathcal{C}_{t_i}|$. **Schedule-Independent** applies **NFDH** on the set of configurations produced after each iteration and, in the end, outputs the one resulting in the minimum schedule length for T_{mid}.

Let $\{c_t^*\}_{t\in T}$ be the set of configurations of tasks in $T \subseteq \mathcal{T}$ in the optimum solution and let $OPT(T)$ denote the length of the shortest continuous time interval that fully contains all the tasks in T in the optimum schedule. Note $OPT(\mathcal{T})$ is OPT by definition.

Theorem 1. *The length of the schedule returned by* ***D&C-Recursive*** *on a set of tasks T is at most $2\sum_{t\in T} \mathrm{t}(c_t^*)\mathrm{p}(c_t^*)/P + OPT(T)\log|T|$.*

Proof: We prove this by induction on the size of T. The base case is $|T| = 0$ and the inequality holds. The total schedule length is the sum of the lengths of schedules S_{bef} and S_{aft} obtained recursively and S_{mid} obtained by applying **NFDH** on each set of configurations produced by the procedure above and taking the shortest. Since T_{mid} is not empty, we have the following guarantee on the lengths of S_{bef} and S_{aft} by the inductive hypothesis:

$$|S_{bef}| \le 2\sum_{t\in T_{bef}} \mathrm{t}(c_t^*)\mathrm{p}(c_t^*)/P + OPT(T_{bef})\log|T_{bef}|$$
$$|S_{aft}| \le 2\sum_{t\in T_{aft}} \mathrm{t}(c_t^*)\mathrm{p}(c_t^*)/P + OPT(T_{aft})\log|T_{aft}|$$

Let $\{c_t\}_{t\in T_{mid}}$ be the set of configurations produced by the procedure above resulting in the minimum length schedule of T_{mid} and $\{c'_t\}_{t\in T_{mid}}$ be the configurations produced satisfying Conditions 1 and 2 simultaneously as suggested by Lemma 2. The length of S_{mid} guaranteed by **NFDH** applied repeatedly on the output configurations of the above procedure is then bounded by

$$\begin{aligned} |S_{mid}| &\le 2\sum_{t\in T_{mid}} \mathrm{t}(c_t)\mathrm{p}(c_t)/P + \max_{t\in T_{mid}} \mathrm{t}(c_t) \\ &\le 2\sum_{t\in T_{mid}} \mathrm{t}(c'_t)\mathrm{p}(c'_t)/P + \max_{t\in T_{mid}} \mathrm{t}(c'_t) \\ &\le 2\sum_{t\in T_{mid}} \mathrm{t}(c_t^*)\mathrm{p}(c_t^*)/P + \max_{t\in T_{mid}} \mathrm{t}(c_t^*). \end{aligned}$$

A crucial observation is that the fragments in any feasible schedule containing the tasks in T_{bef} and the tasks in T_{aft} are disjoint because they are separated by the mid point of the intermediate schedule. Thus, all tasks in T_{aft} have at least one predecessor in T_{bef}, implying that $OPT(T_{bef}) + OPT(T_{aft}) < OPT(T)$. Given $|T_{bef}| \le |T|/2, |T_{aft}| \le |T|/2$ from Lemma 1 and $\max_{t\in T_{mid}} \mathrm{t}(c_t^*) \le OPT(T)$ from the definition of $OPT(T)$, we have

$$\begin{aligned} |S_{bef}| + |S_{mid}| + |S_{aft}| &\le \frac{2}{P}\sum_{t\in T} \mathrm{t}(c_t^*)\mathrm{p}(c_t^*) \\ &\quad + (OPT(T_{bef}) + OPT(T_{aft}))\log(|T|/2) + OPT(T) \\ &\le \frac{2}{P}\sum_{t\in T} \mathrm{t}(c_t^*)\mathrm{p}(c_t^*) + OPT(T)(\log(|T|/2) + 1) \\ &= \frac{2}{P}\sum_{t\in T} \mathrm{t}(c_t^*)\mathrm{p}(c_t^*) + OPT(T)\log|T| \end{aligned}$$

∎

Since $\sum_{t\in T} \mathrm{t}(c_t^*)\mathrm{p}(c_t^*)/P$ is a lower bound on the length of any schedule running tasks in configurations $\{c_t^*\}_{t\in T}$, the

length of the schedule returned by **D&C-Recursive** on the initial set of tasks $\mathcal{T}$ is at most $2\sum_{t\in\mathcal{T}} \mathrm{t}(c_t^*)\mathrm{p}(c_t^*)/P + OPT(\mathcal{T})\log|\mathcal{T}| \leq 2OPT + OPT\log|\mathcal{T}|$. This completes the proof that our algorithm is always within $(2+\log|\mathcal{T}|)$-factor of the optimum solution.

D. A More Efficient Randomized Algorithm

We note that finding a point mid exactly as described in **D&C-Recursive** and having **Schedule-Independent** repeatedly apply **NFDH** on each set of configurations as described in Section III-C both seem to be required only to overcome the technical challenges in proving Theorem 1. We replace these methods with more efficient ones that also reinforce the main principle of the divide-and-conquer algorithm: scheduling tasks with similar run times in pace configurations together. We take mid to be the point maximizing $|T_{mid}|$ among several random picks within a reasonable range around the mid point of the intermediate schedule (in terms of its length). This definition is consistent with the principle above since a large T_{mid} often means better grained set of mid-tasks and more degree of freedom for **Schedule-Independent** to match similar tasks together. Moreover, we let **Schedule-Independent** apply **NFDH** on only the pace configurations $(\{c_i^1\}_i)$ of T_{mid}. Since **NFDH** schedules these tasks in shelves greedily and shelves are always kept disjoint, there is still room for improvement within the individual shelves. In each shelf, we first use up all the unused power budget by choosing a shorter—higher power—configuration for the longest task. We do this incrementally because the longest task may not be the longest anymore after using a portion of the available power budget. Then, similarly, we take power away from the shortest task—making it longer—and use this freed power to make the longest task shorter until no more improvements are possible. These operations essentially try to make the tasks running in a shelf start at the same time, use almost all of the power budget all throughout their execution, and finish around the same time. If we have already matched tasks that have very similar execution times in their initial pace configurations, then the deviation from the pace configurations is minimum and we achieve high power budget utilization with energy efficient configurations. The implemented **D&C** in the rest of the paper refers to this version with randomized mid selection and improved **Schedule-Independent**.

IV. Greedy Algorithms

A. Greedy Strategies

The problem we study and the solution we offer with the divide-and-conquer algorithm is not about the intricacies of scheduling introduced in the systems level such as data locality or network delays, but it is about the essentials of scheduling a DAG under a power-cap. (It could also be argued that some of these additional complexities are easier to deal with in a greedy algorithm and some are easier for a divide-and-conquer algorithm.) For this reason, we compare our algorithm to greedy algorithms that form the basis of today's state-of-the-art schedulers.

One such algorithm for DAG scheduling under a power cap is obtained by extending existing DAG schedulers to take the power cap into account. The list scheduling algorithm proposed initially by Graham [11] has been the essential DAG scheduling algorithm. There is, in fact, a hardness of approximation result [40] stating that one cannot get a better generic algorithm for DAG scheduling. In the list scheduling algorithm, the DAG is extended to a total order (a list) and, as soon as a machine is available, the next task in this list is scheduled on this machine. The analysis in [11] states that an arbitrary extension to a total order is as good as any extension for DAG scheduling. However, with the introduction of the power cap and requirements, we note that this extension can have an effect on the algorithm's performance. Thus we consider two greedy variants differing in the choice of the next tasks to be scheduled among all tasks whose predecessors are all completed:

- **G1:** Always take the next task to be the one whose dependencies completed the first.
- **G2:** Take the "best fit" among the tasks whose dependencies are completed. We define the best fit to be the task whose power requirement in its most energy efficient configuration is closest to the free power budget. This definition aims to minimize the deviation from pace configurations in the final schedule.

We let **G3** refer to a greedy algorithm obtained by extending a power-cap-aware scheduler to also obey the DAG dependencies. It employs the strategy of greedily scheduling the tasks in the non-increasing order of their running time, which is previously used and analyzed in the context of strip packing, resource constrained scheduling, and malleable task scheduling [5], [39]. Since the running time of a task depends on the configuration selected for it, to be able to compare tasks in terms of their run time before scheduling them, **G3** uses their run times in their most energy efficient (pace) configuration. In order to make it also satisfy the DAG dependencies, it only considers the tasks whose predecessors all have been completed.

In all three algorithms described above, the next task is scheduled on the available machine in a configuration that will use the maximum amount of the free power budget. This makes sure that all three algorithms work towards at least one of two goals stated in Section I to achieve better performance: maximize power budget utilization. Note that **G2** also makes the choice for the next task to achieve the second goal simultaneously: maximize energy efficiency of the individual task's configurations.

B. State-of-the-art Schedulers

As mentioned in the introduction, the state-of-the-art in scheduling is adapting greedy scheduling to practical implementation concerns. For example, Spark is a distributed workload management system that is precedence aware, but not power-aware. Spark adapts a greedy scheduler to reduce IO overhead [41]. Cilk++—a language and runtime for multi-threaded workloads—uses a sophisticated work-stealing

scheduler that distributes the scheduling load across all participating processors, while maintaining the fundamentally greedy strategy [9]. Several recent advances have dealt with the practical reality that often precise power and performance tradeoffs are not known, so they augment greedy schedulers with machine learning to estimate the power and time of each configuration and task [42]–[48] or to deal with tasks that take longer than expected (also known as "stragglers") [8], [49]. Even modern power-aware schedulers use greedy strategies, including both commonly deployed solutions [12] and cutting-edge research solutions that deal with serious practical concerns like unknown power/performance relationships [14], manufacturing variability across super-computer nodes [15], and balancing the needs of independent IO- and compute-bound applications [13]. Our evaluation elides these practical details to focus on the fundamental differences between greedy strategies (as described above) and the divide-and-conquer strategy presented in this paper.

V. Evaluation and Results

We evaluate our divide-and-conquer algorithm on a mixture of real and synthetic workloads, using a high-level simulator. We compare our algorithm **D&C** against greedy strategies **G1**, **G2**, and **G3** on a variety of both random and real-world DAG structures varying in size and parallelism depth. We compare the four algorithms against a theoretical lower bound on execution time. We show how different power caps impact the algorithms and that the divide-and-conquer algorithm performs especially well with more stringent power caps. Finally, we show that the divide-and-conquer approach utilizes more of the available power than greedy, and we demonstrate that our approach scales well on large DAGs containing up to 10,000 nodes.

A. Experimental Setup

In this section we describe the simulator used in our experiments, the DAG properties, and the power- and performance measurements obtained. The simulator is a Python package that can be used to implement and test any DAG scheduling algorithm under power constraints. It takes a DAG, power cap and number of machines as input and runs the desired algorithm. The output is the schedule produced by the algorithm and an optional visualization of the power utilization over time.

Our algorithm is DAG-agnostic and can handle any shape and size of the DAG. To demonstrate, we chose a diverse ensemble of DAGs. We consider DAGs from real scientific applications and randomly generated DAGs (using DAGGEN [50]). We consider two simple Swift/T [51] applications with DAGs generated directly by Swift. We have also selected applications from the Rodinia benchmark suite [52] and the NAS Parallel Benchmarks (NPB) [53], whose DAGs were obtained by combining compiler-generated callgraphs using LLVM [54].

The simulator takes a set of configurations for each DAG node that correspond to real measurements of time and power of whole applications obtained from a single machine. The mapping of configurations to DAG structures described above is performed in a random fashion and is configurable. In total we have power and performance measurements of 26 applications obtained from a real system. The applications are a combination of Rodinia and MineBench [55] benchmarks. Power and performance of each application were measured on a dual-socket Linux 3.2.0 system with a SuperMICRO X9DRL-iF motherboard and two Intel Xeon E5-2690 processors. Each processor has 8 cores with hyperthreading, and 16 DVFS settings. In addition, each processor has its own memory controller. Therefore, in total there are 1024 user-accessible configurations (16 cores, 2 hyperthreads, 2 memory controllers, 16 DVFS settings). This system's idle power is approximately 90 W. A user may wish to collect their own power and performance measurements and use them in our simulator instead.

TABLE I
Description of DAGs with node count n and height h

Application	Description	n	h
synth-lg-long	random, generated	$10,000$	701
synth-lg-wide	random, generated	$10,000$	7
swift1	sleep tasks called iteratively	461	3
swift2	iterative calls to Tcl	$4,195$	3
npb-is	NPB kernel: integer Sort	43	5
npb-dc	NPB benchmark: data cube	188	7
backprop	backpropagation algorithm	79	5
kmeans	clustering algorithm	60	6

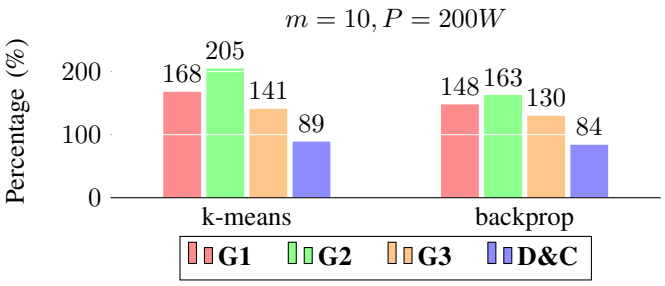

Fig. 2. Percent overhead to theoretical lower bound.

B. Results

For each simulation, we obtain a theoretical lower bound on the execution time. A schedule with all tasks in pace configurations and with a perfect power utilization runs for $\sum_t \mathrm{t}(c_t^{pace})\mathrm{p}(c_t^{pace})/P$ time. This bound can be even smaller than what is attainable by an algorithm because it does not consider the precedence constraints imposed by the DAG. Nevertheless, this lower bound on the execution time is the best that any algorithm can get. We then obtain the additional time of **D&C**, **G1**, **G2**, and **G3** beyond this lower bound in terms of percentages for each simulation. We refer to this additional time as the *overhead* of the strategy compared to the lower bound. With each combination of m, P and DAG parameters, we run a hundred simulations and take the geometric mean of the percent overheads.

Figure 2 shows the mean percent overheads of all four algorithms with $m = 10$ machines under $P = 200W$ power

TABLE II
PERCENT IMPROVEMENT OF **D&C** OVER THE BEST OF **G1**, **G2**, AND **G3** IN EACH CASE WITH 10 AND 20 MACHINES AND VARIOUS POWER CAPS.

m=10	backprop	kmeans	npb-dc	npb-is	swift1	swift2	synth-lg-long
P=100W	60.3	60.0	67.4	64.6	63.9	72.0	32.7
P=200	35.4	36.9	36.2	43.8	40.4	46.2	13.4
P=300	5.8	18.5	9.9	15.5	4.5	3.6	5.0
P=400	3.8	3.1	-7.2	11.8	-12.2	0.0	2.3
P=500	-8.2	5.3	-15.7	-3.0	-14.3	2.4	-18.8
m=20							
P=200W	62.3	61.5	59.8	72.8	74.6	64.3	
P=400	39.4	37.8	49.0	39.2	27.7	30.0	
P=600	2.3	12.9	-8.7	8.2	10.0	-13.2	
P=800	1.9	3.4	-27.2	5.6	6.8	-20.4	
P=1000	-25.5	-10	-15.5	-9.3	-2.3	-24.7	

cap. On the kmeans DAG, for example, **D&C** has on average 89% overhead to the theoretical lower bound whereas the best out of three greedy algorithms, **G3**, has 141%. Thus, we can consider **D&C** to have a $\frac{(141-89)}{141} = 36.9\%$ improvement over the best greedy algorithm in this setting.

Table II summarizes our main results in terms of percent improvement of **D&C** over the *best* greedy algorithm in each setting. When power is severely constrained ($P \leq 300$ for $m = 10$, $P \leq 600$ for $m = 20$) we find that **D&C** averages 35% improvement over the best of greedy algorithms. When power constraints are loose **D&C** tends to have lower performance because the problem starts to become more like DAG scheduling with no power constraints for which the greedy algorithms have been known to perform very well for 50 years [11].

Different global power cap settings impact the efficacy of any DAG scheduling algorithm under power constraints. When the power cap is too liberal, the problem reduces to a DAG scheduling problem with essentially no power cap. To avoid this, for each combination of graph and number of machines, we set the power cap within a range of the mean statistics of the pace configurations' power requirements because, as explained before, any such effective scheduling algorithm should aim for the pace configurations.[2]

Table II demonstrates the superiority of our divide-and-conquer method under strict power caps.[3] Improvements get to as high as 74.6% in the case of $m = 20, P = 200W$ on the swift1 DAG. Negative percentages indicate **D&C** underperforming the best greedy algorithm in that case. **D&C**'s performance over each of there greedy algorithm separately can be found in Table III. A self-evident pattern in these results is the diminishing improvements of **D&C** as the power cap gets bigger. In all experiments, there is a *turning point* for the power cap where **D&C** performs significantly better for power caps below this point and, when the power cap is above this point, either one of the greedy algorithms starts to perform better or all four algorithms tend to perform similarly depending on the particular DAG. For example, npb-is DAG on $m = 20$ machines has this turning point somewhere between $P = 800W$ and $P = 1000W$. As mentioned earlier, DAG scheduling under power constraints starts to resemble DAG scheduling with no power constraints as the power cap increases. Since our divide-and-conquer method is particularly designed for scheduling under power constraints, the observed diminishing improvements of **D&C** as the power cap increases are exactly what is expected.

We stated that the principle of concurrently scheduling similar pace tasks gives our divide-and-conquer method an edge in better managing the trade-off between conflicting approaches of choosing energy efficient configurations and utilizing the power budget. We also note that this trade-off is biased towards the power budget utilization. Our results for energy efficiency of the configurations chosen by all four algorithms and for the power utilization attained by them support this claim. In majority of the cases where **D&C** outperforms greedy algorithms, it utilizes significantly more power budget on average than these algorithms. In the combinations of machine numbers, power caps, and DAGs reported in Table II, greedy algorithms waste 44.2% of the power budget on average whereas **D&C**'s non-utilized power budget is only 30% of the total available budget. Still, these values may seem considerably high for both algorithms. As one would expect, both algorithms manage to utilize a much bigger portion of the power budget when the power cap is low compared to the cases when the power cap is high. On the npb-dc graph with $m = 10$ machines, for example, **G3** and **D&C** fail to utilize 10.9% and 3.8% when the power budget is $P = 100W$, respectively, but these numbers get as high as 36.6% and 32.8% when it is $P = 500W$. Neither algorithm can significantly utilize a power budget set by a high power cap, because of two reasons: applications have realistic configuration sets in which arbitrarily increasing the power requirements of a task—thereby speeding up the task—is not possible, and all these DAGs have parts with insufficient parallelism to utilize the whole power budget. Moreover, the poor power utilization by both algorithms is another indicator that the problem on high power caps starts to turn into DAG scheduling with no power cap.

When **D&C** cannot utilize more power than the greedy algorithms, it still has an advantage by scheduling tasks closer to pace configurations than the greedy algorithms can. On swift2 with $m = 10$ and $P = 100W$, for example, **D&C** and **G3** fail to utilize 3.4% and 1.3% of the power budget, respectively, but **D&C** still manages to get better final execution times by choosing configurations with total energy only 4% more than the best possible with pace configurations.

[2]Mean power requirement of the pace configurations in different simulations ranges from $15W$ to $20W$ above idle.

[3]As the DAG synth-lg-long is a "long" graph with very limited parallelism, we only report $m = 10$ for synth-lg-long.

TABLE III
PERCENT OVERHEAD OF **G1**|**G2**|**G3**|**D&C**, RESPECTIVELY, ON THE THEORETICAL LOWER BOUND. LOWER IS BETTER. FOR EACH ENTRY THE BEST RESULT IS IN BOLD AND THE SECOND BEST IS ITALIC.

m=10	backprop	kmeans	npb-dc	npb-is	swift1	swift2	synth-lg-long
P=100W	*73*|100|78|**29**	*115*|172|128|**46**	*46*|64|*46*|**15**	149|215|*127*|**45**	*36*|46|41|**13**	26|39|*25*|**7**	60|90|*55*|**37**
P=200	148|163|*130*|**84**	168|205|*141*|**89**	62|65|*58*|**37**	197|216|*176*|**99**	*47*|50|*47*|**28**	*26*|*26*|*26*|**14**	182|280|*149*|**129**
P=300	178|193|*155*|**146**	228|255|*195*|**159**	94|111|*81*|**73**	329|323|*245*|**207**	45|*44*|*44*|**42**	29|29|*28*|**27**	293|394|*221*|**210**
P=400	201|206|*182*|**175**	227|234|*195*|**189**	*83*|88|**77**|*83*	378|397|*321*|**283**	*37*|*37*|**36**|41	*35*|*35*|**33**|**33**	280|339|*215*|**210**
P=500	319|361|**259**|*282*	372|427|*323*|**306**	*118*|129|**113**|134	304|356|**262**|*270*	*62*|*62*|**60**|70	43|43|*42*|**41**	351|429|**277**|*341*
m=20							
P=200W	224|315|*220*|**83**	252|403|*281*|**97**	*92*|136|100|**37**	*383*|491|398|**104**	*67*|92|*67*|**17**	29|39|*28*|**10**	
P=400	247|247|*213*|**129**	378|416|*328*|**204**	123|124|*104*|**53**	558|630|*492*|**299**	74|95|*65*|**47**	31|32|*30*|**21**	
P=600	420|491|*344*|**336**	471|565|*412*|**359**	180|194|**137**|*150*	508|525|*379*|**348**	85|83|*70*|**63**	**33**|**33**|**33**|*38*	
P=800	480|563|*429*|**421**	659|725|*527*|**509**	*156*|176|**134**|184	832|985|*709*|**669**	*117*|132|313|**109**	*44*|*44*|**43**|54	
P=1000	513|499|**369**|*495*	728|780|**583**|*648*	361|410|**288**|*341*	924|971|**708**|*781*	**127**|131|137|*130*	**58**|*59*|**58**|77	

TABLE IV
PERCENT OVERHEAD OF **G1**|**G2**|**G3**|**D&C**, RESPECTIVELY, ON THE THEORETICAL LOWER BOUND.

m=50	swift1	swift2	synth-lg-wide
P=500W	132|165|*93*|**69**	*34*|46|36|**18**	*29*|43|**28**|32
P=800	*219*|450|399|**120**	*43*|47|44|**32**	**36**|48|*38*|53
P=1000	317|605|*260*|**161**	**41**|50|*42*|**41**	*43*|66|**36**|74
P=1500	*541*|927|746|**259**	**32**|*33*|**32**|53	*58*|88|**41**|123
P=2000	535|846|*515*|**270**	**54**|*55*|**54**|97	*50*|57|**42**|124
m=100			
P=500W	124|141|*82*|**60**	*31*|38|39|**14**	**29**|37|40|*33*
P=800	*215*|441|292|**114**	*40*|55|49|**27**	*48*|92|**45**|67
P=1000	312|574|*271*|**154**	*52*|85|58|**41**	*48*|86|**45**|71
P=2000	613|897|*612*|**252**	**62**|101|*66*|82	*97*|119|**66**|103
P=3000	925|1153|*916*|**349**	46|*43*|**41**|96	*180*|212|**125**|217

In the same setting, **G3**, the best greedy algorithm among all three in this particular setting, chooses configurations with total energy 24% more than the energy of pace configurations. Thus, in this case **D&C** is saving significant energy compared to the best greedy approach.

In addition to improvement of **D&C** over the best greedy algorithm in each case presented in Table II, we present each algorithms' overhead to theoretical lower bound in Table III. In Table IV, we provide additional results in support of the scalability of our algorithms. We present the settings with $m = 50$ and $m = 100$ on the DAGs that are large enough and that have enough parallelism to benefit from the increase in the number of machines. One might consider increasing the range of power caps proportional to the number of machines m. However, a power cap set this way for $m = 100$ machines to $P = 6000W$, for example, will reduce the problem to essentially a DAG scheduling with no power cap at all times when the DAG cannot utilize all 100 machines. Also recent research indicates it may be more efficient overall to more severely cap large systems and make more use of parallelism than speed of individual nodes [33], [34]. We report in the range $500 - 2000W$ for 50 machines and in the range $500-3000W$ for 100 machines. We observe the same behavior of **D&C** having diminishing improvements as we increase the power cap. We note that the size of any particular DAG does not change in our simulations as we increase m from 10 to 100 and P from $100W$ to $3000W$; i.e., we are simulating strong scaling results.

VI. CONCLUSION

This paper addresses the problem of scheduling parallel workloads (represented as DAGs) under power constraints on distributed systems with configurable resources. We present a novel divide-and-conquer algorithm that finds and concurrently schedules similar tasks to maximize the power utilization while keeping energy efficiency high. We prove that our algorithm is always within $O(\log n)$ factor of the optimum solution. We also present three variations on greedy algorithms that form the basis of todays state-of-the-art schedulers. We show that under strict power caps (which are expected to become common in the next generation of exascale supercomputers) the divide-and-conquer algorithm's ability to achieve both high power utilization and high energy efficiency results in much lower runtimes for a number of DAG workloads. Furthermore, this algorithm scales up to work on DAGs with 10,000 nodes. The divide-and-conquer algorithm presented here extends the best known algorithm for scheduling under both precedence and power constraints by incorporating the practical concern of considering configurable resources. Therefore, we believe this algorithm has the potential to significantly impact future scheduler implementations for HPC systems.

ACKNOWLEDGMENT

We thank anonymous referees—especially our shepherd—for their insightful comments. This research was supported by the DARPA BRASS program, a DOE Early Career award, and the NSF (CCF-1439156).

REFERENCES

[1] K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, J. Hiller, S. Karp, S. Keckler, D. Klein, R. Lucas, M. Richards, A. Scarpelli, S. Scott, A. Snavely, T. Sterling, R. S. Williams, K. Yelick, K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, J. Hiller, S. Keckler, D. Klein, P. Kogge, R. S. Williams, and K. Yelick, "Exascale computing study: Technology challenges in achieving exascale systems peter kogge, editor & study lead," 2008.

[2] V. Sarkar, S. Amarasinghe, D. Campbell, W. Carlson, A. Chien, W. Dally, E. Elnohazy, M. Hall, R. Harrison, W. Harrod, K. Hill, J. Hiller, S. Karp, C. Koelbel, D. Koester, P. Kogge, J. Levesque, D. Reed, R. Schreiber, M. Richards, A. Scarpelli, J. Shalf, A. Snavely, and T. Sterling, "Exascale software study: Software challenges in extreme scale systems," 2009, dARPA IPTO Study Report for William Harrod.

[3] J. Shalf, S. Dosanjh, and J. Morrison, "Exascale computing technology challenges," in *VECPAR*, 2011.

[4] J. Ang, R. Barrett, R. Benner, D. Burke, C. Chan, D. Donofrio, S. Hammond, K. Hemmert, S. Kelly, H. Le, V. Leung, D. Resnick, A. Rodrigues, J. Shalf, D. Stark, D. Unat, and N. Wright, "Abstract machine models and proxy architectures for exascale computing," Sandia National Laboratories, Tech. Rep., 2014.

[5] J. E. G. Coffman, M. R. Garey, D. S. Johnson, and R. E. Tarjan, "Performance bounds for level-oriented two-dimensional packing algorithms," *SIAM Journal on Computing*, vol. 9, no. 4, pp. 808–826, 1980.

[6] G. Demirci, H. Hoffmann, and D. H. K. Kim, "Approximation Algorithms for Scheduling with Resource and Precedence Constraints," in *35th Symposium on Theoretical Aspects of Computer Science (STACS 2018)*, ser. LIPIcs, vol. 96, 2018, pp. 25:1–25:14.

[7] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *2nd USENIX Workshop on Hot Topics in Cloud Computing, HotCloud'10, Boston, MA, USA, June 22, 2010*, 2010.

[8] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," in *OSDI'04: Proc. of the 6th conf. on Opearting Systems Design & Implementation*, 2004.

[9] C. E. Leiserson, "The cilk++ concurrency platform," in *2009 46th ACM/IEEE Design Automation Conference*, July 2009, pp. 522–527.

[10] J. Reinders, *Intel Threading Building Blocks*, 1st ed. Sebastopol, CA, USA: O'Reilly & Associates, Inc., 2007.

[11] R. L. Graham, "Bounds for certain multiprocessing anomalies," *Bell Labs Technical Journal*, vol. 45, no. 9, pp. 1563–1581, 1966.

[12] SLURM, "The slurm workload manager," Online document, https://slurm.schedmd.com/.

[13] L. Savoie, D. K. Lowenthal, B. R. d. Supinski, T. Islam, K. Mohror, B. Rountree, and M. Schulz, "I/o aware power shifting," in *IPDPS*, May 2016.

[14] T. Patki, D. K. Lowenthal, A. Sasidharan, M. Maiterth, B. L. Rountree, M. Schulz, and B. R. de Supinski, "Practical resource management in power-constrained, high performance computing," in *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '15. New York, NY, USA: ACM, 2015, pp. 121–132. [Online]. Available: http://doi.acm.org/10.1145/2749246.2749262

[15] Y. Inadomi, T. Patki, K. Inoue, M. Aoyagi, B. Rountree, M. Schulz, D. Lowenthal, Y. Wada, K. Fukazawa, M. Ueda, M. Kondo, and I. Miyoshi, "Analyzing and mitigating the impact of manufacturing variability in power-constrained supercomputing," in *SC*, 2015.

[16] F. F. Yao, A. J. Demers, and S. Shenker, "A scheduling model for reduced cpu energy," in *FOCS*, 1995.

[17] D. H. K. Kim, C. Imes, and H. Hoffmann, "Racing and pacing to idle: Theoretical and empirical analysis of energy optimization heuristics," in *CPSNA*, 2015.

[18] S. Albers, "Algorithms for dynamic speed scaling," in *STACS*, 2011, pp. 1–11.

[19] S. Irani, S. Shukla, and R. Gupta, "Algorithms for power savings," *ACM Trans. Algorithms*, vol. 3, no. 4, Nov. 2007.

[20] E. Bini, G. C. Buttazzo, and G. Lipari, "Minimizing cpu energy in real-time systems with discrete speed management," *ACM Trans. Embedded Comput. Syst.*, vol. 8, no. 4, 2009.

[21] H. Yun, P.-L. Wu, A. Arya, C. Kim, T. F. Abdelzaher, and L. Sha, "System-wide energy optimization for multiple dvs components and real-time tasks," *Real-Time Systems*, vol. 47, no. 5, 2011.

[22] P. Pillai and K. G. Shin, "Real-time dynamic voltage scaling for low-power embedded operating systems," in *SOSP*, 2001.

[23] H. Aydi, P. Mejía-Alvarez, D. Mossé, and R. Melhem, "Dynamic and aggressive scheduling techniques for power-aware real-time systems," in *RTSS*, 2001.

[24] S. Albers and A. Antoniadis, "Race to idle: new algorithms for speed scaling with a sleep state," in *SODA*, 2012.

[25] N. Bansal, H.-L. Chan, T. W. Lam, and L.-K. Lee, "Scheduling for speed bounded processors," in *ICALP*, 2008.

[26] N. Bansal, D. P. Bunde, H.-L. Chan, and K. Pruhs, "Average rate speed scaling," *Algorithmica*, vol. 60, no. 4, 2011.

[27] H.-L. Chan, J. W.-T. Chan, T. W. Lam, L.-K. Lee, K.-S. Mak, and P. W. H. Wong, "Optimizing throughput and energy in online deadline scheduling," *ACM Transactions on Algorithms*, vol. 6, no. 1, 2009.

[28] N. Bansal, H.-L. Chan, and K. Pruhs, "Speed scaling with an arbitrary power function," *ACM Transactions on Algorithms*, vol. 9, no. 2, 2013.

[29] K. Pruhs, R. van Stee, and P. Uthaisombut, "Speed scaling of tasks with precedence constraints," in *WAOA*, 2006.

[30] D. P. Bunde, "Power-aware scheduling for makespan and flow," in *SPAA*, 2006.

[31] N. Kappiah, V. W. Freeh, and D. K. Lowenthal, "Just in time dynamic voltage scaling: Exploiting inter-node slack to save energy in mpi programs," in *SC*, 2005.

[32] A. Marathe, P. E. Bailey, D. K. Lowenthal, B. Rountree, M. Schulz, and B. R. de Supinski, "A run-time system for power-constrained hpc applications," in *ISC*, 2015.

[33] T. Patki, D. K. Lowenthal, A. Sasidharan, M. Maiterth, B. L. Rountree, M. Schulz, and B. R. de Supinski, "Practical resource management in power-constrained, high performance computing," in *HPDC*, 2015. [Online]. Available: http://doi.acm.org/10.1145/2749246.2749262

[34] O. Sarood, A. Langer, L. Kale, B. Rountree, and B. de Supinski, "Optimizing power allocation to cpu and memory subsystems in over-provisioned hpc systems," in *CLUSTER*, 2013.

[35] P. E. Bailey, A. Marathe, D. K. Lowenthal, B. Rountree, and M. Schulz, "Finding the limits of power-constrained application performance," in *SC*, 2015.

[36] R. Lepère, D. Trystram, and G. J. Woeginger, "Approximation algorithms for scheduling malleable tasks under precedence constraints," in *Proceedings of the 9th Annual European Symposium on Algorithms*, ser. ESA '01, 2001, pp. 146–157.

[37] K. Jansen and H. Zhang, "An approximation algorithm for scheduling malleable tasks under general precedence constraints," *ACM Trans. Algorithms*, vol. 2, no. 3, pp. 416–434, Jul. 2006.

[38] ——, "Scheduling malleable tasks with precedence constraints," *Journal of Computer and System Sciences*, vol. 78, no. 1, pp. 245 – 259, 2012, jCSS Knowledge Representation and Reasoning.

[39] J. Turek, J. L. Wolf, and P. S. Yu, "Approximate algorithms scheduling parallelizable tasks," in *Proceedings of the Fourth Annual ACM Symposium on Parallel Algorithms and Architectures*, ser. SPAA '92, 1992, pp. 323–332.

[40] O. Svensson, "Conditional hardness of precedence constrained scheduling on identical machines," in *Proceedings of the Forty-second ACM Symposium on Theory of Computing*, ser. STOC '10. New York, NY, USA: ACM, 2010, pp. 745–754.

[41] S. Venkataraman, A. Panda, G. Ananthanarayanan, M. J. Franklin, and I. Stoica, "The power of choice in data-aware cluster scheduling," in *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014.*, 2014.

[42] C. Delimitrou and C. Kozyrakis, "Paragon: Qos-aware scheduling for heterogeneous datacenters," in *ASPLOS*, 2013.

[43] ——, "Quasar: Resource-efficient and qos-aware cluster management," in *ASPLOS*, 2014.

[44] N. Mishra, J. D. Lafferty, and H. Hoffmann, "Esp: A machine learning approach to predicting application interference," in *ICAC*, 2017.

[45] N. Mishra, H. Zhang, J. D. Lafferty, and H. Hoffmann, "A probabilistic graphical model-based approach for minimizing energy under performance constraints," in *ASPLOS*, 2015.

[46] N. Mishra, C. Imes, J. D. Lafferty, and H. Hoffmann, "CALOREE: Learning Control for Predictable Latency and Low Energy," in *ASPLOS*, 2018.

[47] H. Zhang and H. Hoffmann, "Maximizing Performance Under a Power Cap: A Comparison of Hardware, Software, and Hybrid Techniques," in *ASPLOS*, 2016.

[48] ——, "Performance & energy tradeoffs for dependent distributed applications under system-wide power caps," in *ICPP*, 2018.

[49] N. J. Yadwadkar, B. Hariharan, J. E. Gonzalez, and R. Katz, "Multi-task learning for straggler avoiding predictive job scheduling," *Journal of Machine Learning Research*, vol. 17, no. 106, pp. 1–37, 2016. [Online]. Available: http://jmlr.org/papers/v17/15-149.html

[50] "DAGGEN: A Synthetic Task Graph Generator," 2013. [Online]. Available: https://github.com/frs69wq/daggen

[51] J. M. Wozniak, T. G. Armstrong, M. Wilde, D. S. Katz, E. Lusk, and I. T. Foster, "Swift/t: large-scale application composition via distributed-memory dataflow processing," in *CCGrid*, 2013.

[52] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *IISWC*, 2009.

[53] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber *et al.*, "The nas parallel benchmarks," *International Journal of High Performance Computing Applications*, vol. 5, no. 3, pp. 63–73, 1991.

[54] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*. IEEE, 2004, pp. 75–86.

[55] R. Narayanan, B. Özisikyilmaz, J. Zambreno, G. Memik, and A. Choudhary, "Minebench: A benchmark suite for data mining workloads," in *Workload Characterization, 2006 IEEE International Symposium on*. IEEE, 2006, pp. 182–188.