Finding the Limits of Power-Constrained Application Performance

Peter E. Bailey
Aniruddha Marathe
David K. Lowenthal
Dept. of Computer Science
The University of Arizona

Barry Rountree
Martin Schulz
Lawrence Livermore National Laboratory

ABSTRACT

As we approach exascale systems, power is turning from an optimization goal to a critical operating constraint. With power bounds imposed by both stakeholders and the limitations of existing infrastructure, we need to develop new techniques that work with limited power to extract maximum performance. In this paper, we explore this area and provide an approach to find the theoretical upper bound of computational performance on a per-application basis in hybrid MPI + OpenMP applications.

We use a linear programming (LP) formulation to optimize application schedules under various power constraints, where a schedule consists of a DVFS state and number of OpenMP threads for each section of computation between consecutive MPI calls. We also provide a more flexible mixed integer-linear (ILP) formulation and show that the resulting schedules closely match schedules from the LP formulation. Across four applications, we use our LP-derived upper bounds to show that current approaches trail optimal, power-constrained performance by up to 41.1%. This demonstrates the untapped potential of current systems, and our LP formulation provides future optimization approaches with a quantitative optimization target.

1. INTRODUCTION

With power consumption becoming a critical—if not *the* critical—supercomputer operating constraint, future systems must adhere to strict power limits. For example, in the US, exascale systems currently have a target power consumption—set by the Department of Energy—of 20 MW, other agencies world-wide have set similar limits. We anticipate that total machine power will be divided across multiple simultaneous jobs, with each job being allocated a power bound and a set of nodes. This motivates the development of techniques for optimizing performance under a strict power constraint for multi-node, multi-core systems. The problem at hand is then to choose a per-processor *configuration* (a number of cores and frequency/voltage [DVFS] state) for a given application executing on a particular number of nodes.

Any technique for choosing configurations under a job-level power constraint suffers from a configuration space that is combinatorially large. A single processor commonly has 16 or more hardware threads and supports a dozen DVFS states. Multiplied by the number of processors in a typical job, the size of the configuration space makes finding the optimal configuration for each processor into a large combinatorial problem, and any analytical solution for the optimal configuration of all processors in the job becomes computationally intractable. Combined with the fact that different configurations can result in vastly different performance [21], we cannot reliably establish the quality of existing approaches, which are potentially missing out on significant performance gains.

In this paper, we develop the first theoretical formulation of the power-constrained performance optimization problem for MPI + OpenMP applications. We present an optimal verified linear programming (LP) formulation, which places a theoretical upper bound on the application performance that can be achieved on power-constrained systems. The LP formulation is optimal under certain conditions. We also present an integer linear programming (ILP) formulation of the same problem, which, while optimal, is not feasible for realistic problem sizes. However, we show that the LP and ILP formulations yield similar results, and that the LP allows us to find theoretical limits for realistic problem sizes.

By taking traces of application execution across multiple configurations and applying either linear or integer programming, we create a schedule of configuration changes that, if applied at runtime, would result in optimal execution time under the given job-level power constraint. We emphasize that we are not proposing this methodology as a production runtime system. Rather, the techniques described in this paper can be used to *evaluate* runtime systems to determine how closely they approach the theoretical bound.

In this paper, we make the following contributions:

- We develop a linear programming formulation to maximize performance under a job-level power constraint through configuration selection and scheduling.
- We validate the LP formulation and the resulting schedules on a cluster system.
- We demonstrate an ILP formulation and show that it closely matches our LP formulation.
- We compare our LP results with those of state-of-the-art heuristics and show there is room for improvement in online power-constrained performance optimization.

As we show in Section 6, our LP-generated schedule yields up to 41.1% improvement in power-constrained performance over state-of-the-art power reallocation systems, and up to 74.9% improvement over the de facto standard of static uniform power caps. This

SC '15, November 15-20, 2015, Austin, TX, USA

DOI: http://dx.doi.org/10.1145/2807591.2807637

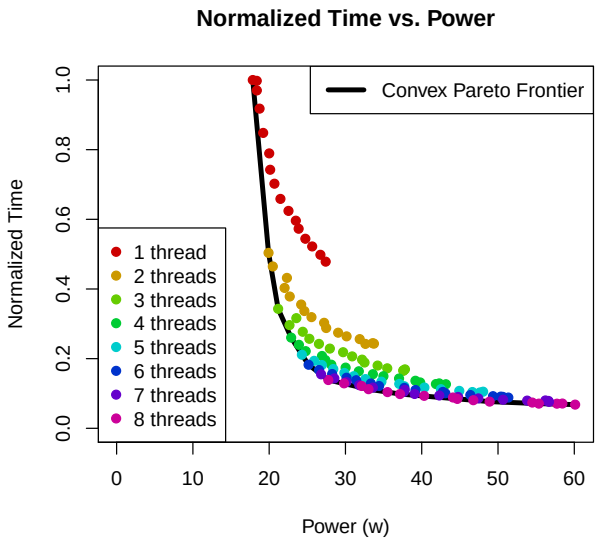

Figure 1: Time vs. processor power for a task from CoMD. For a specific number of threads, power increases and task time decreases with increasing processor frequency.

both demonstrates the shortfalls of current techniques and provides the research community with an approach to evaluate future systems against realistic performance targets.

In the rest of this paper, Section 2 gives a background to the problem of optimizing power-constrained performance. Section 3 provides step-by-step analysis of our solution, Section 4 details contemporary algorithms for running an application under a power constraint, and Section 5 identifies our test system and benchmarks. Next, Section 6 presents analysis and interpretation of our results. Finally, Section 7 explores related work, and Section 8 concludes.

2. OVERVIEW

With power and performance becoming equally important in supercomputer design and operation, every design or operation decision must take both into account, as they are directly related. Past supercomputer designs have largely sidestepped this issue by designing for worst-case power consumption, but future designs will require careful consideration of every system-level power-performance tradeoff [27]. With a multitude of such tradeoffs under consideration, the limits of power-constrained application performance remain an open question.

Numerous factors influence both application power and time to solution in modern parallel, distributed-memory machines. Some of these factors are fixed; in this paper, we focus on dynamically tunable options that can be effected at run time. The simplest runtime choices are node-level configuration options, such as Dynamic Voltage and Frequency Scaling (DVFS)—dynamically adjusting the processor voltage and frequency—and Dynamic Concurrency Throttling (DCT) [8]—dynamically adjusting the number of threads used in a parallel region.

2.1 Job-Level Power-Constrained Optimization

To our knowledge, this paper is the first to present an upper bound on power-constrained performance for MPI + OpenMP applications on a specified number of nodes. Prior work has investigated maximizing power-constrained performance on a *single node* [5, 7]. These works share a common theme, which is to find Pareto-efficient configurations. The set of Pareto-efficient configurations maximizes performance given a power constraint. An example of a time-power Pareto frontier is shown in Figure 1.

Configuration	Freq. (GHz)	Threads
$C_{i,\ 1}$	2.6	8
$C_{i,\ldots}$	...	8
$C_{i,15}$	1.2	8
$C_{i,16}$	1.2	7
$C_{i,17}$	1.2	6
$C_{i,18}$	1.2	5
$C_{i,19}$	1.2	4

Table 1: A sample of time/power Pareto-efficient configurations, C_i, from the task shown in Figure 1.

Previous efforts have targeted distributed-memory applications, which can be represented as a directed acyclic graph (DAG); Figure 2 shows an example graph. In this DAG, vertices correspond to collective operations as well as point-to-point message initiation/reception, and edges correspond to either message transmissions between MPI processes (weighted by a linear function of message size) or computation tasks between two MPI calls on the same process (for the latter we use "edges" and "tasks" interchangeably). Extending this DAG to model hybrid MPI + OpenMP applications means that computation edges can be run in many DVFS and DCT configurations.

Importantly, previous efforts have not targeted maximization of power-constrained performance. Configuration selection has been applied in MPI and MPI + OpenMP applications for the purpose of minimizing energy consumption [19] and to find the extent to which it is possible to save energy without decreasing performance [24]. The goals of minimizing energy and maximizing power-constrained performance are related, yet substantially different. It is important to note that the approaches presented in [19, 24] make no attempt to constrain power, and their requirement that performance be maintained near maximum leaves little room for trading off power and performance in general. Consequently, these approaches require a system with fully provisioned worst-case power, a luxury we won't have in future systems.

The problem that we are solving—optimizing performance subject to a job-level power bound in MPI + OpenMP applications—*cannot* simply borrow existing techniques such as the ones mentioned above. MPI semantics facilitate complex interactions and dependencies between processes, and, at any given time, multiple computation tasks will be running. The presence of multiple tasks overlapping in time adds complexity to the optimization problem; increasing performance (and therefore power) of a single task reduces available power for other simultaneously running tasks on other ranks, potentially increasing overall time to solution.

More importantly, determining all possible sets of co-scheduled tasks is computationally intractable, but any method of optimizing power-constrained performance must determine exactly which tasks are executing at a given point in time. This requirement is complicated by the fact that changing a task's power allocation may shift the task in time, thus changing its set of co-scheduled tasks and potentially changing its optimal power allocation. A valid solution to this problem must determine per-rank power constraints covering the entirety of application execution. These power constraints may change over time, depending on changing application characteristics such as load imbalance. The rank-level power constraints must be selected to maximize application performance un-

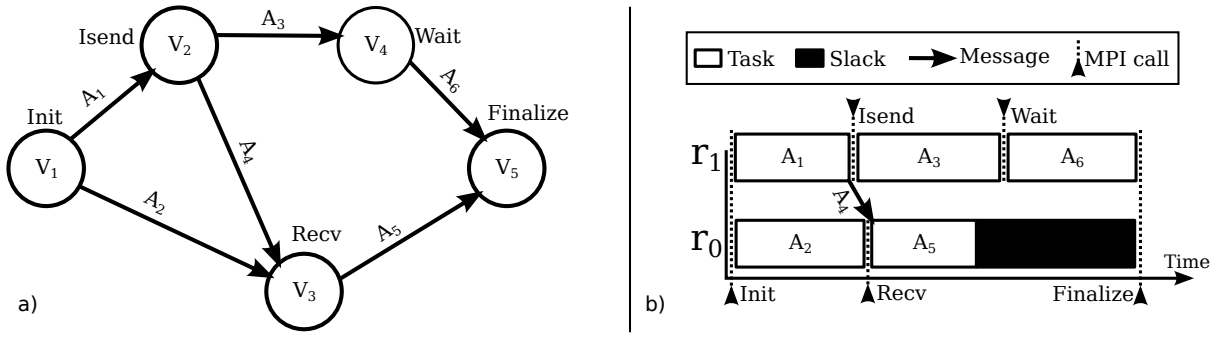

Figure 2: A simple application task graph (a) and timeline (b).

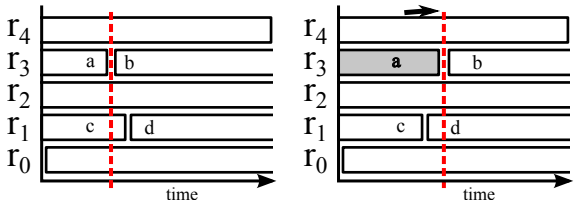

Figure 3: An application's instantaneous power is determined by tasks that overlap in time. A single task may run simultaneously with many tasks, and a change in task's duration may change the set of tasks it overlaps. In this example, task *a* is slowed down, which causes *a* to overlap with *d*, and *b* to cease overlapping with *c*.

der a global job-level constraint, and each rank must select an appropriate configuration for each of its tasks according to varying power availability.

Figure 3 shows the co-scheduling problem. On rank 3, tasks a and b are shown, and task a is slowed. This leads to a different set of tasks executing at the event between tasks a and b demarcated by the dotted line. As practical applications have more than two MPI ranks, it is important to find an efficient solution method for such problems. We introduce our method in Section 3.

2.2 Assumptions

In this paper, we examine applications that meet the following characteristics, which are common in modern HPC applications across all application areas: they (a) are written with MPI and OpenMP, (b) call MPI functions from outside OpenMP parallel regions (e.g., `MPI_THREAD_MULTIPLE`), and (c) run one multithreaded process per socket. The first requirement serves to maximize the range of configurations available for each MPI task, while the second requirement dramatically simplifies the ILP/LP formulations. The third requirement avoids NUMA effects and simplifies power measurement, as RAPL [9] (the mechanism we use for power measurement and per-processor capping) reports data at the socket level, and the processors in our test system only support DVFS at the socket level.

In this paper, we assume that the number of MPI ranks is selected by the user. In a power-constrained environment, this assumption may lead to wasted power and suboptimal performance, but the problem of efficiently allocating resources *between* jobs in a power-constrained cluster is not the focus of this paper, and has been examined previously (e.g. [21]).

2.3 Summary

Previous approaches to power/energy optimization problems have focused either on a single node or on "best-effort" power/energy approaches subject to limited performance loss. It is much more complicated to optimize performance under a power constraint, because changes made to improve performance can result in task-shifting that violates the power constraint. Nontrivial optimization-based approaches—i.e., ones that aggressively attempt to improve performance while respecting the power constraint—require careful treatment. The next section describes two different approaches.

3. LINEAR AND INTEGER/LINEAR PROGRAMMING FORMULATIONS

To address the concerns raised in Section 2, we present an event-based LP formulation of the power-constrained performance optimization problem. While LP formulations are impractical for real-time task scheduling, they provide optimal offline solutions, allowing us, for the first time, to study the optimization potential in power-limited systems and to evaluate online approaches against realistic optimization targets.

Below, Section 3.1 provides the implementation details of our LP formulation, Section 3.2 shows how we handle configurations in the LP formulation, Section 3.3 explains how the LP formulation constrains job-level power using events, and Section 3.4 details our ILP formulation.

3.1 LP Design

Our LP formulation depends on a directed acyclic graph (DAG) representation of the application's computation and communication dependencies, which we obtain from an MPI tracing library. The DAG edges correspond to communication and computation (*tasks*), while the DAG vertices correspond to MPI function calls. Figure 2 shows an example DAG and one potential execution timeline. In general, all applications can be represented in DAG form, but obtaining the information necessary to construct the DAG may require instrumentation of communication or library calls.

Each task can be run in a number of configurations, each associated with a unique power and duration. Table 1 lists sample configurations. The power-constrained performance optimization problem is an instance of a larger class of problems, namely Multi-Mode Resource-Constrained Project Scheduling Problems, or MM-RCPSPs. Drawing on existing treatments of such problems by Koné et al. [18] and Artigues et al. [3] , we adapt previous solution strategies to the problem at hand. Our treatment of the problem differs from previous solutions in that we present a linearization of the event-based formulation that results in realistic, verifiable schedules, as we will show in Section 3.3.

As shown in Figure 4, the our LP formulation is based on one

Symbol	Description
C_i	available configurations for task i
$c_{i,g}$	fraction of task i in configuration g
d_i	duration of task i
$d_{i,g}$	duration of task i to completion in configuration g
p_i	average power of task i
$p_{i,g}$	power of task i in configuration g
s_i	start time of task i
v_y	time of vertex y
$E_{i,j}$	task precedence; 1 if $dest(i) = src(j)$
A	task set; $\|A\| = N$
V	vertex set; $\|V\| = M$
v_M	total duration
PC	job-level power constraint
P_y	power consumption at vertex y
R_y	tasks active at vertex y

Table 2: Symbols used in LP formulation and their definitions.

$$\text{objective}: minimize(v_M) \quad (1)$$
$$v_1 = 0 \quad (2)$$
$$s_j - s_i \geq d_i \; \forall (i,j) \in E \quad (3)$$
$$s_i = v_y \; \forall (i,y) \in \{A \times V\} \mid src(i) = y \quad (4)$$

Figure 4: Objective and constraints for task ordering.

objective (1) and several constraints (2)-(4); Table 2 lists the symbols used and their definitions. Equation (1) shows the objective, which is to minimize application time to solution; we define v_M to be the time of the `MPI_Finalize()` vertex. Equation (2) sets the `MPI_Init()` vertex to time 0. Consequently, all tasks in the application must start at or later than time 0. Of course, most task start times will be determined by dependencies inherent in the application. Equation (3) sets minimum task spacing in time; this enforces task precedence relationships inherent in the application. That is, if task j directly depends on task i, then task j cannot start until i completes. When applied to all combinations in the application task precedence set, E, Equation (3) allows each task to start only after all its predecessors have completed. Equation (4) requires all tasks with a common source vertex to start simultaneously, which can be the case for collective operations or message sources (e.g., `MPI_Send` or `MPI_Isend()`). While our formulation accounts for slack time (the time in the schedule before the locally subsequent task can be started minus the task's execution time), we require a task to precede its slack for simplicity (i.e., a tasks executes and then waits until the next task can be executed). Without Equation (4), slack would be allowed to precede its associated task. Due to our assumption that slack power is equal to the power of the associated task (see Section 3.3), our requirement that tasks precede their slack does not affect the resulting schedule.

In addition to the constraints in Figure 4, our LP formulation relies on representations of per-task configurations. Section 3.2 details their implementation.

3.2 Configurations

Our LP solution procedure handles both discrete and continuous configuration spaces, governed by the equations in Figure 5. In the discrete case, each task is in a single configuration for its entire execution time. In the continuous case, the initial solution places each task in a configuration that may not correspond to an integral

$$c_{i,g} \in \{0,1\} \; \forall i \in A \; \forall g \in C_i \text{ (discrete)} \quad (5)$$
$$0 \leq c_{i,g} \leq 1 \; \forall i \in A \; \forall g \in C_i \text{ (continuous)} \quad (6)$$
$$d_i = \sum_{g \in C_i} d_{i,g} c_{i,g} \; \forall i \in A \quad (7)$$
$$p_i = \sum_{g \in C_i} p_{i,g} c_{i,g} \; \forall i \in A \quad (8)$$
$$\sum_{g \in C_i} c_{i,g} = 1 \; \forall i \in A \quad (9)$$

Figure 5: Equations constraining task configurations. Equation (8) **is an approximation; see Figure 6.**

number of threads or available DVFS states, but results in a shorter time to solution (assuming negligible configuration switching overhead). In either case, the problem is initially formulated with continuous configurations, and the resulting solution is rounded to produce discrete configurations that are realizable in terms of available concurrency levels and DVFS states. The continuous case is implemented by switching the configuration mid-task to emulate the effect of the optimal configurations using multiple physically available discrete configurations, while the discrete case is rounded by selecting the configuration closest to the optimal point on the Pareto frontier. We take this approach because if the problem is initially formulated with discrete configurations, it becomes mixed integer/linear (ILP). This requires a significantly less efficient solution method, which prohibits us from solving realistic problems.

In the discrete case, Equation (5) restricts each task i to a single configuration. We require all tasks to run in a single configuration for their entire duration to avoid excess DVFS overhead, and because changing the concurrency level mid-task is not supported in OpenMP. In the continuous case, Equation (6) restricts each task to a point along a continuum of configurations. In practice, the resulting configuration selection will lie between two neighboring discrete configurations. Equation (7) states that the time required to complete task i is equal to the sum of the time spent in each configuration, and Equation (8) states that the average power for a task is equal to the weighted average power over all configurations used for that task. Equation (9) requires each task to be completed in at least one configuration. Equations (6)-(9) are in common with the LP formulation by Rountree et al. [24].

The LP formulation uses a simplified representation of average power for tasks split between configurations. Technically, the average power of a task split between configurations is not a linear function of the $c_{i,g}$ variables; see Equation (10). However, it is possible to approximate the average power with a linear function; see Equation (11). Equation (11) is derived from Equation (10) using the approximation $d_{i,1} = d_{i,2}$. This approximation is reasonable due to the selection of configurations on the Pareto frontier; at most two $c_{i,g}$ will be nonzero for any task i, and they will always be next to each other on the frontier. This approximation leads to a maximum error of 1.1% in worst-case examples of tested application tasks, but any error caused by the approximation will be in the form of overestimated task power consumption. Therefore, the true average task power is always less than the estimated power, and the resulting schedules are always achievable by a theoretical runtime system.

The introduction of configurations to the problem requires us to find Pareto-efficient, convex (with respect to power and time) sets

$$p_i = \frac{p_{i,1}c_{i,1}d_{i,1} + p_{i,2}c_{i,2}d_{i,2}}{c_{i,1}d_{i,1} + c_{i,2}d_{i,2}} \quad (10)$$

$$p_i = \frac{p_{i,1}c_{i,1} + p_{i,2}c_{i,2}}{c_{i,1} + c_{i,2}} = p_{i,1}c_{i,1} + p_{i,2}c_{i,2} \quad (11)$$

Figure 6: Average power for task i with two candidate configurations. Because neighboring configurations are similar in duration, equation (11) is used as a linear approximation of Equation (10).

of configurations for each task in order to create a purely linear formulation of the problem. Figure 1 depicts one such frontier, along with all possible configurations of the task. Without the convexity requirement, the Pareto frontiers would force the problem to become mixed integer/linear, since a non-convex Pareto frontier cannot be represented as a convex, piecewise-linear function.

Figure 1 also provides the basis for part of our run-time power reallocation algorithm, introduced in Section 4.2; for many application tasks that we tested, increasing the number of threads reliably increased performance, and running with fewer than the maximum number of threads was only Pareto-efficient (on the convex frontier) at the minimum processor frequency.

3.3 Role of Events in the LP Formulation

Our LP formulation constrains power consumption at discrete events. Events correspond to vertices in the application DAG, and the required power for each event is determined by the active tasks at that event. Tasks are considered active at an event if they start at or are running at the time of the event in an initial schedule.

The initial event order is provided by a power-unconstrained schedule for the application DAG that has been modified to reduce slack time. The modification does not change the overall time to solution, but slows tasks off the critical path as much as possible. This initial schedule also provides the task activity set, R. To maximize flexibility in event ordering, slack power is assumed equal to its corresponding task power in this formulation. If a task's slack power were treated as distinct from the active power (as in the Appendix), additional power would be available for use in other simultaneously running tasks, at the expense of introducing additional events at task/slack boundaries. For the event-based formulation, we favor having fewer events over a marginal increase in power sharing to maximize each task's power and performance scaling range.

We consider a restricted version of the event-based formulation in which the time order of events is fixed; this differs from Koné et al. [18] in which the event order is determined by the solver, which requires a mixed integer-linear (ILP) formulation. Our approach results in a strictly linear formulation, which allows for linear solvers and efficient, polynomial-time solutions. This method could be applied to thousands of processes and hundreds of edges per process with little difficulty due the low overhead of our tracing library and the efficiency of LP solvers, whereas existing ILP formulations have only solved instances of tens of edges, which limits analysis to only two MPI processes and a single message exchange.

Figure 7 lists the constraints that make up the fixed-vertex order formulation. Equation (12) provides a lower bound on power at each event as the sum of task power for active tasks at that event; recall that tasks are considered active at an event if they start at or are running at the time of the event in an initial schedule. The vertex/event power is the key variable that enables us to constrain

$$P_y \geq \sum_{i \in R_y} p_i \; \forall y \in V \quad (12)$$

$$P_y \leq PC \; \forall y \in V \quad (13)$$

$$v_y \leq v_z \; \forall (y,z) \in V^2 \mid event(v_y) < event(v_z) \quad (14)$$

$$v_y = v_z \; \forall (y,z) \in V^2 \mid event(v_y) = event(v_z) \quad (15)$$

Figure 7: Power-related constraints in the fixed-vertex order LP.

job-level power consumption, and Equation (12) establishes its definition. Equation (13) restricts each vertex/event power to be under the job-level power constraint, and, in combination with the event and task ordering constraints, ensures our goal of optimizing power-constrained performance. Equations (14) and (15) keep events in their initial order with respect to time. Without Equations (14) and (15), we would be unable to guarantee that the job-level power constraint is respected because arbitrary tasks could overlap in time, and such an overlap may result in higher power consumption than the fixed event order.

3.4 Flow ILP

As detailed in Section 3, our LP/ILP formulations are based on a DAG derived from the application. The flow ILP depends on an additional DAG to account for power consumption. The power DAG closely tracks the application DAG, but also includes edges that depend on a sequencing relationship between tasks (x). In effect, the ILP solver ensures that power flows forward in time from an artificial power source vertex at time zero (vertex 0) to an artificial power sink vertex (vertex $N+1$) at the end of the application (that is, after the last MPI process reaches `MPI_Finalize`). By limiting the power input to the power DAG and carefully constraining how that power is used, we ensure that the resulting application schedule never violates its power constraint. Figure 8 shows an example.

In contrast to our LP formulation, which fixes event order, the flow formulation requires the ILP solver to determine event order. Much of the complexity in solving flow formulation instances arises from the size of the x space; a small two-rank application DAG could result in 2^{100} possible x states. For this reason, we focus on the linear, fixed-vertex order formulation in this paper. The Appendix contains a detailed description of the flow-based, ILP formulation.

While the flow ILP formulation is practically limited to solving small (i.e. fewer than 30 DAG edges) problems, our LP formulation reaches equivalent schedules on a synthetic benchmark over a range of power constraints. The results are shown in Figure 9. For all but three of the 106 power limits tested, the two formulations agree on the application schedule time to within 1.9%. Under the power limits in which the fixed-order and flow formulations disagree, it is clear that providing less than a watt of additional power to the fixed-order formulation would allow it to achieve an equivalent schedule.

3.5 Optimality

For the power-constrained performance optimization problem stated above, our LP formulation guarantees an optimal solution if one exists, under two assumptions: the specific event order used is optimal, and the space of configurations is continuous. The event order arises from the LP solver's need to be aware of which tasks are running at any given time; without this constraint, the problem

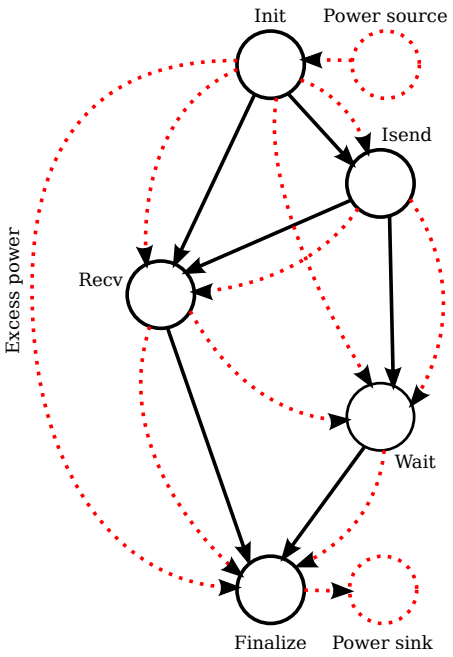

Figure 8: An example overlay of application (black) and power flow (red) DAGs in the flow formulation.

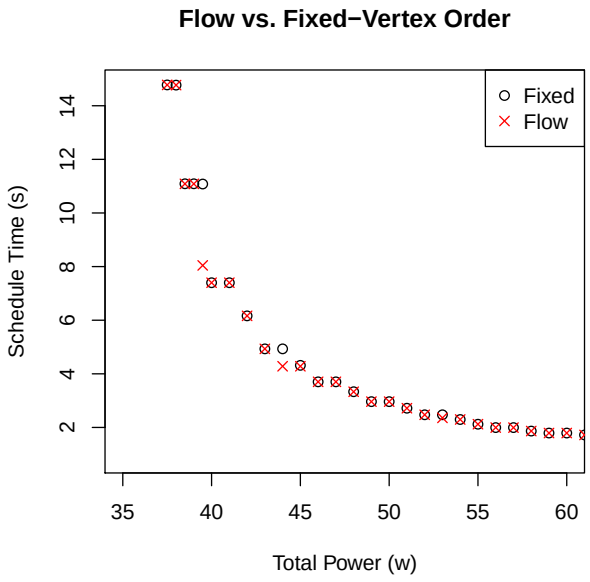

Figure 9: Comparison of flow and fixed-vertex order formulations on a two-process asynchronous message exchange. Results from both formulations in all cases beyond 60w are within 1.9% of each other.

is not linear and becomes an ILP problem. We did not prove that the event order given by our system is optimal, but instead showed the resulting schedules are equivalent to ILP schedules on a small testcase (Figure 9). While the configuration space is not continuous in practice, it can be treated as such by rounding to discrete configurations or simulating the continuous space by splitting tasks into multiple configurations. We chose the former; for an example of the latter, see Adagio [23]. While our paper does not prove the given bounds are optimal, they are the best known bounds for problems of this type.

4. POWER-ALLOCATION ALGORITHMS

We use our LP formulation to evaluate the effectiveness of two contemporary power-allocation algorithms presented in our previous work [20]. The first is a simple static allocation, and the second is a run-time system called *Conductor*. They are described in turn below.

4.1 Static: Fixed, Uniform Power Allocation

The simplest method to allocate per-node power is to distribute application-level power equally between the nodes, which we term *Static*; this method has been used effectively in production clusters within the U.S. Department of Energy. In Intel Xeon processors, power can be capped at the socket level with RAPL by writing the desired power limit to a hardware register [9, 14]. RAPL runs as a control algorithm in firmware that operates asynchronously with respect to the application and selects DVFS states. Because RAPL is implemented in firmware, it is unable to change application concurrency levels or other user-level or OS-level configuration options that may affect power and performance. To maximize performance for most applications, we fix the thread concurrency level at eight per processor, which is the number of hardware cores. *Static* serves as a baseline in our experiments.

4.2 Conductor: Adaptive Power Allocation

To overcome the limitations of RAPL and to reduce load imbalance, we use an adaptive power-allocation algorithm, *Conductor* [20], that periodically changes, based on application behavior and the current power constraint, processor frequency and thread count. *Conductor* consists of two major components: a configuration exploration step, which selects the optimal thread concurrency level for individual computation tasks under the power limit; and, a power reallocation step, which intelligently re-assigns power based on per-process power usage. The following subsections briefly describe these components.

A typical time step may involve several computation operations, each of which may run optimally at a different configuration. Because RAPL can only scale the processor frequency (via DVFS and clock modulation), *Conductor* must select the optimal configuration for each computation operation. In order to select the optimal configuration, *Conductor* records the power and performance profile of each computation operation at all possible configurations. To reduce the run-time overhead of exploring all configurations, *Conductor* runs several configurations in parallel by assigning a unique configuration to each MPI process and collecting the profiled information at the end of the time step. The profiling itself is implemented transparently to the application through the MPI Profiling Interface. Using the profiled information, *Conductor* creates a list of power-efficient configurations (that are Pareto-efficient) for each computation operation. During application execution, a new node-level power limit may be assigned, at which time *Conductor* will select the new optimal configuration under the updated power constraint.

In the second step, *Conductor* monitors power usage per MPI process, estimates the critical path of the application, and reallocates power to speed up the critical path. *Conductor* first deploys Adagio [23] to reduce power consumption in non-critical computation operations by selecting a low-power configuration that finishes computation without perturbing the critical path of the application. After Adagio has slowed non-critical operations to free up power, critical operations run at or near their process-level power constraint, while processes with no (or very few) critical tasks do not use all of their power allocation. Such a situation may be caused by load imbalance inherent to the application, or differences in power efficiency between individual processors. *Conductor* takes advantage of the difference in power usage to speed up the application by reallocating power between processes without violating the constraint. *Conductor* performs the power reallocation step at the end of several time steps demarcated using `MPI_Pcontrol`[1], which we assume has been added by the user.

5. EXPERIMENTAL SETUP

5.1 System and Setup

All experiments were performed on Cab, a 1296-node Xeon E5-2670 cluster at LLNL with an InfiniBand QDR interconnect. Each cab node is composed of two 8-core processors and 32 GB of DRAM.

We use the same number of active cores for OpenMP regions between consecutive MPI calls; that is, we only change the number of active threads at task boundaries, which are demarcated by MPI calls. This mirrors previous work by Li et al. [19]; the reason for making this restriction is that our dynamic selection of OpenMP concurrency (i.e., dynamic concurrency throttling, or DCT) potentially reduces the effectiveness of caching. Changing the number of active cores between two parallel regions temporarily increases the rate of cache misses.

5.2 Benchmarks and Tools

We analyzed and validated our optimizations and runtime power reallocation systems on CoMD, LULESH 2.0, and SP and BT from NAS-MZ. These benchmarks were designed to exhibit performance and scaling behavior typical of applications of interest to the US Department of Energy, and LULESH was used in evaluation of bids for the CORAL series of machines [1]. All benchmarks were tested with 32 MPI processes across 32 8-core processors, using up to 256 cores at a time. We modified all benchmarks to include `MPI_Pcontrol` calls at iteration boundaries to simplify LP data processing and help *Conductor* identify application phases.

CoMD [2] is a molecular dynamics benchmark. CoMD is unique among our tested benchmarks in that all of its MPI communication is in the form of collectives. As a result, the only task that remains for the LP solver or power reallocation algorithm is to minimize load imbalance by reallocating power between ranks at every collective call.

LULESH 2.0 [17] is a shock hydrodynamics benchmark. In terms of MPI communication, LULESH differs from CoMD in that it relies on a multitude of point-to-point messages between collective calls. This behavior complicates analysis of opportunities to reallocate power, but we show in Section 6 that *Conductor* achieves performance that closely tracks optimal, LP-derived performance, trailing by only 5.1% in the average case.

NAS-MZ [28] is an adaptation of the NAS Parallel Benchmarks [4] to the MPI + OpenMP paradigm. The NAS benchmarks were originally designed to be representative of computations and data movement in fluid dynamics applications.

[1]MPI_Pcontrol is part of the MPI profiling interface and allows application instrumentation to be communicated to tools built on top of the MPI Profiling interface.

5.3 Comparisons

To demonstrate that contemporary algorithms have much room for improvement, we compare our LP results with those of *Static* and *Conductor*. Because the flow ILP formulation is limited to solving small problem instances (i.e., fewer than 30 application DAG edges), we compare only the fixed-vertex order LP results to our power reallocation algorithm and uniform static power constraints.

Conductor requires a configuration exploration phase to find configurations on the power-time Pareto frontier for all tasks in a given application. In our comparisons, we discard the first three iterations of every application, which roughly corresponds to the configuration exploration phase; we assume that applications will run for enough iterations to amortize the cost of the exploration phase.

6. EVALUATION AND RESULTS

In this Section, we compare the LP-derived schedules with *Static* and *Conductor*. In summary, we find that current power reallocation systems such as *Conductor* can significantly improve power-constrained performance over *Static*, but in some cases trail optimal, LP-derived schedules by up to 41.1%. *Conductor* closely follows LP-derived schedules in some applications, though it incurs a small performance degradation compared to *Static* (maximum of 2.6%) in some cases. *Conductor* improves performance by an average of 6.7% over *Static*, while the LP indicates an average of 10.8% potential improvement from *Static* to an optimal schedule.

If only the configuration selection is performed (but not power reallocation), there is less overhead than *Conductor*, but also lower performance due to the use of uniform power allocation. In general, the reason *Conductor* trails the LP schedules in performance is not the communication overhead, at least at the scales tested in this paper. The reasons we found to be significant are thrashing in the per-rank power allocation (which induces load imbalance) and configuration switching overhead.

6.1 Validation

To verify that our LP and ILP schedules are realizable and within their power constraints, we *replay* them on their originating benchmarks by selecting a configuration for each task according to the LP/ILP-derived schedule. As the application encounters each MPI call, our replay mechanism changes the configuration appropriately for the next computation task according to the prescribed schedule. All LP and ILP results presented in this Section were validated in the above manner.

Changing configurations between every pair of tasks introduces significant overhead, especially for applications composed primarily of short tasks. To counteract this, we only change configurations if the schedule indicates that the upcoming task will be of sufficient duration to justify the overhead. We use a threshold of 1ms for this purpose.

6.2 Overheads

As we instrument every instance of many MPI calls, our profiler incurs some overhead. The median measurement overhead is 34 microseconds per MPI call, and adds less than 0.05% time to the tested applications. When replaying LP-derived schedules, our runtime makes DVFS transitions between tasks. These incur additional overhead, resulting in a median per-task overhead of 145 microseconds. For the runtime power reallocation algorithms, all power allocation decisions are made synchronously at applica-

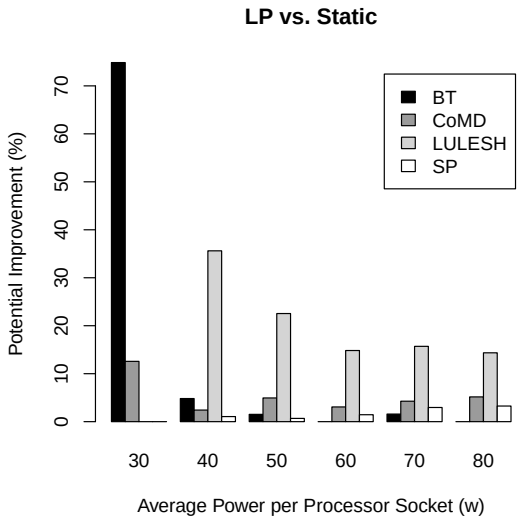

Figure 10: Potential speedup of LP-derived schedules vs. *Static*. Some benchmarks were not able to be scheduled at the lowest average per-socket power constraint.

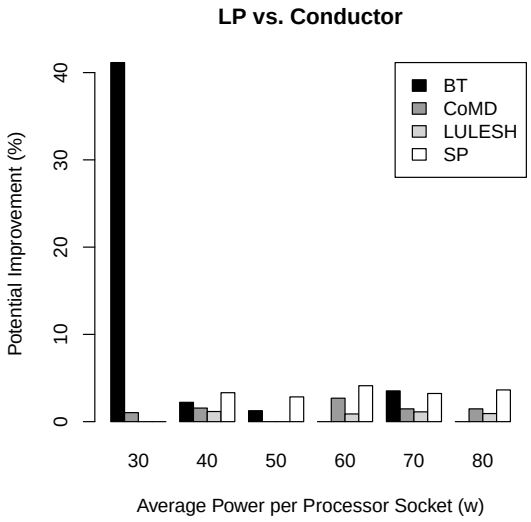

Figure 11: Potential speedup of LP-derived schedules vs. *Conductor*. Some benchmarks were not able to be scheduled at the lowest average per-socket power constraint.

tion `MPI_Pcontrol` calls, with an average overhead of 566 microseconds per invocation. This overhead is amortized across iterations, as power allocation decisions are made after every 5-10 `MPI_Pcontrol` calls.

Our LP and ILP formulations and associated verification tools are not intended as a run-time system. Accordingly, they include steps that make them unsuitable for online performance optimization. First, our system requires offline measurements of the target application in all possible processor configurations (DVFS, OpenMP concurrency), precluding online use. Second, significant processing is required to convert application traces into a format compatible with LP solvers; configurations must be tabulated and power-inefficient points eliminated, and MPI message sources/sinks must be matched. Finally, creating the input to the LP solver and solving the LP formulation takes multiple minutes.

6.3 Results Overview

Across all benchmarks, we find that *Static* lags the optimal LP performance by up to 74.9%. However, this varies across benchmarks and power constraints; in some cases, *Static* is completely sufficient, as the LP-derived schedule shows no benefit. The LP schedule's indication of potential performance improvement is derived primarily from two features: (1) selection of Pareto-efficient configurations for all tasks and (2) frequent adjustment of per-process power allocation to benefit tasks on the critical path. The LP optimizes these two features simultaneously, resulting in higher performance than is achievable by both *Static* and *Conductor* unless perfect knowledge of the system and applications exists. Nonetheless, we conclude that the naive *Static* approach is effective in a surprising number of situations, including some nontrivial applications (e.g., CoMD).

The largest advantages of the LP schedules over *Static* are generally at low per-processor power constraints, which can be observed in Figure 10. Such low power constraints may be encountered when a job scheduler prioritizes other co-scheduled jobs, or when a user runs a completely memory-bound application. The advantage of the LP is due the LP schedules' non-uniform power allocation and optimal configuration selection. In the *Static* schedules, MPI processes are restricted to eight threads at low frequency, whereas the LP schedule selects more power-efficient configurations and allocates enough power to critical tasks to run fewer threads at higher frequency and voltage while minimizing load imbalance. Except for SP, this is also true for individual benchmarks; the greatest gains come at the lowest power constraint. This behavior results in a 35.6% potential speedup over *Static* for LULESH at 40 watts per MPI process and a 74.9% potential speedup over *Static* for BT at 30 watts per MPI process.

Figure 11 shows potential improvement in *Conductor* [20]. Relative to the LP, *Conductor*'s performance is uncorrelated with power constraints. Regardless, the performance of *Conductor* is close to the optimal results derived from LP schedules in many cases; for CoMD, SP, and LULESH, the performance of *Conductor* is within 4.2% of the LP schedules. In one case in CoMD and one case in LULESH, *Conductor* and the LP arrived at equivalent schedules.

6.4 Individual Benchmarks

In the following discussion of benchmark results, we emphasize that the LP improvement vs. *Static* is *potential* improvement, as the LP is not an online runtime system. The *Conductor* improvement, however, is *demonstrated* performance improvement over *Static*.

For applications with minimal load imbalance, our LP-derived schedules show limited benefit over *Static* at power constraints above 30 watts per processor. The LP-derived schedules for CoMD (see Figure 12) show up to 12.6% potential performance improvement for *Static*, with the median 4.6% and the minimum 2.4%. On the other hand, *Conductor* is extremely close to the LP schedules (within 3%). At an average of 30 watts per processor (see Figure 13), the LP allocates power between ranks such that the longest task takes about 1.2s and many tasks use more than 30 watts, yet the job-level power constraint is not violated. In contrast, *Static* is limited to 30 watts on every socket, which triggers RAPL to select lower DVFS states on some sockets, leading task times routinely above 1.3s and as high as 1.47s. Both the LP and *Static* select 8 threads per processor in this case, while *Conductor* selects

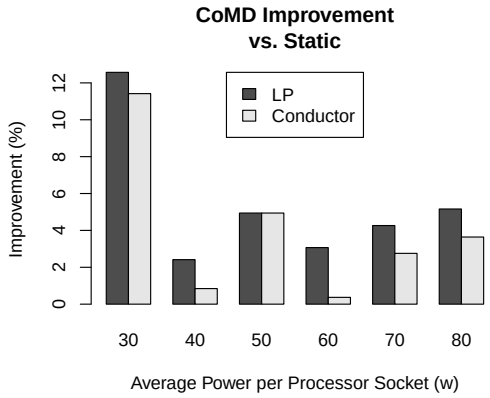

Figure 12: Performance comparison of LP and *Conductor* vs. *Static* for CoMD.

7 threads per processor for less than 1% of long-running (> 0.5s) tasks and allocates up to 32 watts per processor in contrast to the LP's 36 watts. As a result, *Conductor* trails optimal performance by 1% in this case.

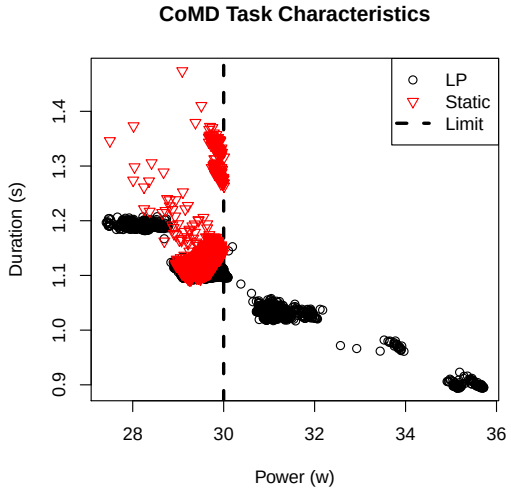

Figure 13: Task duration vs. power for long-running (> 0.5s) tasks in 100 iterations of CoMD at an average per-socket power constraint of 30 watts.

For BT with an average of 30 watts per processor, our LP-derived schedules indicate that *Static* trails optimal performance by 75% and *Conductor* by 24% (Figure 14). These improvements are due to the ability of the LP and *Conductor* to create nonuniform power allocations across ranks, mitigating load imbalance. By allocating power to processes on the critical path, the LP and *Conductor* are able to choose high-performance configurations where *Static* would not. RAPL causes *Static* to run some processors at 22% of their maximum clock frequency while using eight threads per processor, while the LP and *Conductor* are able to run fewer threads at higher frequencies, resulting in higher performance. At higher power constraints, the three methods are within 4.8% of each other.

SP, a scalar pentadiagonal solver from the NAS-MZ benchmark suite, presents a challenge for *Conductor* and shows little room

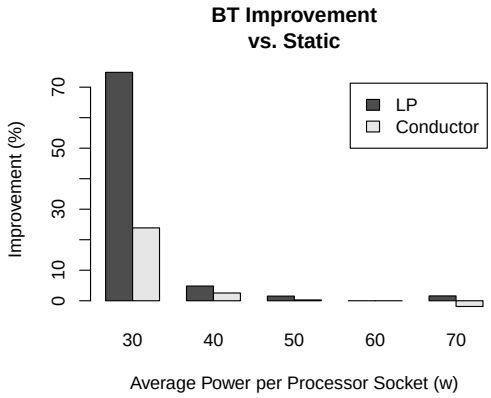

Figure 14: Performance comparison of LP and *Conductor* vs. *Static* for BT.

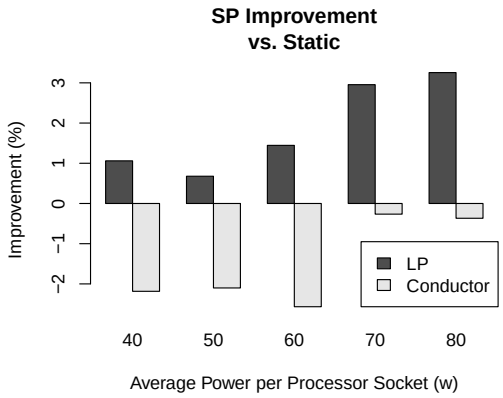

Figure 15: Performance comparison of LP and *Conductor* vs. *Static* for SP.

for improvement via the LP (Figure 15). *Conductor* frequently misidentifies the critical path, inappropriately reducing the power allocation to specific processes. This leads to an average of 1.5% slowdown compared to *Static*. In the worst case, at an average of 60 watts per processor, *Conductor* trails *Static* by 2.6%. The overhead of DVFS transitions and associated logic accounts for 14% of this difference, adding an average of 17 microseconds per task. The overhead of power reallocation, at 566 microseconds per iteration, accounts for another 5.1% of the difference. The remaining difference is entirely attributable to *Conductor* inducing load imbalance between ranks by selecting suboptimal configurations for a subset of tasks in many iterations, thus increasing the time to solution. Overall, it is clear that a runtime system (*Conductor* being no exception) intended to optimize performance under a power constraint must take care to avoid degrading performance of load balanced programs.

Applied to LULESH, the LP indicates significant (>14%) room for improvement for *Static* at all tested job-level power constraints. *Conductor* is able to achieve 99% of the optimal performance indicated by the LP in all cases (Figure 16) and arrives at the LP's schedule under an average per-processor power constraint of 50 watts. Table 3 details a single iteration of this 50-watt case. In this particular iteration, *Conductor* and LP achieve similar time

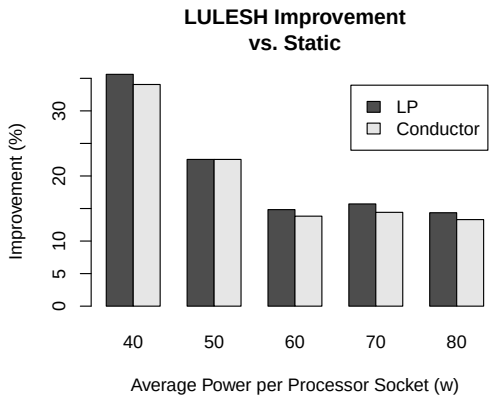

Figure 16: Performance comparison of LP and *Conductor* vs. *Static* for LULESH.

to completion by choosing five OpenMP threads per MPI process, which results in higher performance via lower cache contention and higher average CPU frequencies (because more power can be allocated to each core) for these strategies. *Static* selects eight OpenMP threads, which limits average CPU frequency and decreases performance vs. five threads, perhaps due to cache contention. *Conductor* and the LP also reduce load imbalance by allowing nonuniform power allocation between processes, as indicated by the "standard deviation of power" column in Table 3.

Method	Median time	Std. Dev. power	Threads	Median frequency
Static	4.889	0.009	8	0.8834
Conductor	3.614	0.118	5	0.9942
LP	3.611	0.125	4-5	1.0

Table 3: Task characteristics for a single iteration of LULESH at 1350w (average of 50 watts per processor), considering only long-running (>= 1s) tasks. Median frequency is with respect to the maximum non-boosted clock frequency supported by the processor.

7. RELATED WORK

The most related work to ours is focused on saving energy without increasing execution time in HPC applications using linear programming [24]. The difference between their work and ours is (1) they assume a flat MPI model, whereas our hybrid MPI + OpenMP model makes the problem significantly more complicated, and (2) their work is focused on bounding energy consumption, whereas our work is focused on bounding performance under a power constraint. In general, optimizing for energy consumption may reduce power consumption, but makes no guarantees regarding power constraints.

Several run-time systems focused on minimizing energy-delay product (EDP) or saving energy with modest time increase. These systems included Adagio [23], CPU-Miser [13], AVG [12], and Jitter [16]. Li et al. [19] was the first to consider hybrid MPI + OpenMP programs, where the goal was to minimize the energy-delay product. These systems all leveraged load imbalance or memory (sometimes both) to allow a reduction of CPU frequency with small performance degradation.

Several have worked at power-constrained performance optimization at the run-time system level. A first step is to optimize performance for a single, power-constrained node. The work by Isci et al. [15] optimized performance under a power bound on a single multicore node, and in previous work [5], we did the same for a heterogeneous (CPU+GPU) node. Our work on *Conductor* uses a run-time system to choose configurations as well as reallocate power (described in detail in Section 4) [20].

There is other work in improving (but not optimizing) cluster performance under a power bound, especially on overprovisioned HPC clusters. This includes an empirical study on the effect of different configurations [21], choosing configurations via interpolation [26], and shifting power within a job [6]. Barker et al. demonstrate a system for shifting power in a power-constrained machine to improve performance [6]. This system allocates power to nodes based on load imbalance per iteration. To ensure a global power constraint is enforced, the system iteratively limits the maximum DVFS state on all processors. Our work has a different focus, namely identifying feasible limits of power-constrained optimization, but does not require iterations or global synchronization in the application and doesn't restrain performance of any node in a job unnecessarily. Our system optimizes power allocation at the task level, which implicitly counteracts load imbalance. There has also been work on scheduling algorithms to improve performance under a power bound [10, 11, 25, 22].

8. CONCLUSIONS

By demonstrating upper bounds on power-constrained hybrid MPI + OpenMP application performance, we have shown that current runtimes are leaving some performance on the table and wasting power. In our tests, we found that an optimal power reallocation and configuration selection algorithm can achieve up to 41.1% more performance than existing algorithms, and that requiring configurable node-level parallelism such as OpenMP in addition to MPI goes a long way towards the goal of flexible power and performance management. However, in most cases, our *Conductor* runtime is quite close to the optimal performance as indicated by our LP. In some cases, *Conductor* even achieves optimal schedules as identified by the LP.

The ability to efficiently manage power at runtime will become increasingly important with future generations of supercomputers, which will require new power-aware runtimes. Our work in this paper provides the necessary goalposts and realistic targets for any future runtime to compare against.

9. ACKNOWLEDGEMENTS

This material is based upon work supported by the National Science Foundation under Grant No. 1216829, by the 2013 Exascale Operating and Runtime Systems Program under the Office of Advanced Scientific Computing Research in the DOE Office of Science, and by the U.S. Department of Energy's Lawrence Livermore National Laboratory, Office of Science, under Award number DE-AC52-07NA27344 and by Office of Science, Office of Advanced Scientific Computing Research (LLNL-CONF-675735).

10. APPENDIX: FLOW ILP

The flow-based ILP formulation adds two sets of variables over the common equations in Figure 4: x, the sequence variables, and f, the power flow variables; Figure 17 contains constraints necessary for integrating these two variable sets into the flow formulation, and Table 4 lists the additional symbols. In contrast to the LP

Symbol	Description
E'_{yz}	vertex precedence; 1 if $\exists$ edge from vertex y to z
f_{ij}	power flow from task i to task j
TE	transitive closure of E
TE'	transitive closure of E'
x_{ij}	sequencing; 1 if task i finishes before task j starts
A_0	$A \cup 0$
A_{N+1}	$A \cup \{N+1\}$
A'	$A \cup \{0, N+1\}$

Table 4: Additional symbols used in ILP formulation and their definitions.

formulation, the ILP formulation treats slack separately from computation tasks. Specifically, the boundary between a computation task and its associated slack is generally treated the same as any other vertex in the application DAG, and slack power is no longer assumed equal to its corresponding task power. The ILP formulation assigns a specific power consumption to all slack based on observed slack power on our test system.

The sequence variables x are binary (16), and represent both task precedence inherent in the application (17, 20) and task sequencing determined by the solver for the optimal schedule (18, 19). Specifically, (17) represents all precedence information present in the application DAG, (20) prevents a task from depending on itself, (18) requires that either task i depends on task j, j depends on i, or neither. (19) is a transitivity requirement; in effect, if x_{ij} and x_{ik}, then x_{ik}. (21-24) ensure that slack time is allocated appropriately; (21) and (23) force edges to start immediately after their source vertex's dependencies are completed, while (22) and (24) force edges with destination vertices of multiple in-degree to finish simultaneously with their destination vertex time. (25) uses the sequence variables, x, as indicator variables to enforce a schedule-specific version of (3); in effect, if task j starts after task i finishes in the schedule under evaluation, then (25) becomes (3), ensuring proper task spacing in time. Otherwise, (25) is always true.

Equations (26) and (27) refer to two special edges added to the power DAG. Edge 0 is added prior to the first application vertex (i.e. MPI_Init), while edge $N+1$ is added after the last application vertex (MPI_Finalize). These edges act as a source and sink, respectively, for power, and limit the total instantaneous power consumed by the application to a specific quantity. (28) and (29) establish lower and upper constraints, respectively, on power flow edges in the power DAG. (29) sets flow between tasks i and j to 0 if x_{ij} is 0, and the minimum of p_i and p_j otherwise. Together, (29-31) establish that the power entering and leaving an application DAG edge is equal to the power required to execute that edge in the application under the current schedule.

$$x_{ij} \in \{0,1\}\ \forall (i,j) \in A' \quad (16)$$

$$x_{ij} = 1\ \forall (i,j) \in TE \quad (17)$$

$$x_{ij} + x_{ji} <= 1\ \forall (i,j) \in A'^2 \quad (18)$$

$$x_{ik} >= x_{ij} + x_{jk} - 1\ \forall (i,j,k) \in A'^3 \quad (19)$$

$$x_{ii} = 0\ \forall i \in A' \quad (20)$$

$$x_{ij} = 0\ \forall (i,j) \in A'^2 \mid (src(j), src(i)) \in TE' \quad (21)$$

$$x_{ij} = 0\ \forall (i,j) \in A'^2 \mid (dest(j), dest(i)) \in TE' \quad (22)$$

$$x_{ij} = 0\ \forall (i,j) \in A'^2 \mid src(i) = src(j) \quad (23)$$

$$x_{ij} = 0\ \forall (i,j) \in A'^2 \mid dest(i) = dest(j) \quad (24)$$

$$s_j - s_i >= -M_{ij} + (d_i + M_{ij}) x_{ij}\ \forall (i,j) \in A'^2 \quad (25)$$

$$d_0 = d_{N+1} = 0 \quad (26)$$

$$p_0 = p_{N+1} = PC \quad (27)$$

$$f_{ij} >= 0\ \forall (i,j) \in A'^2 \quad (28)$$

$$f_{ij} <= min(p_i, p_j) x_{ij}\ \forall (i,j) \in A'^2 \quad (29)$$

$$\sum_{j \in A'} f_{ij} = p_i\ \forall i \in A_0 \quad (30)$$

$$\sum_{i \in A'} f_{ij} = p_a\ \forall j \in A_{N+1} \quad (31)$$

Figure 17: Additional constraints in the flow ILP.

11. REFERENCES

[1] Coral benchmark codes. https://asc.llnl.gov/CORAL-benchmarks. Accessed: 2015-01-13.

[2] Comd. https://github.com/exmatex/CoMD, 2013.

[3] C. Artigues, O. Koné, P. Lopez, and M. Mongeau. Mixed-integer linear programming formulations. In C. Schwindt and J. Zimmermann, editors, *Handbook on Project Management and Scheduling Vol.1*, International Handbooks on Information Systems, pages 17–41. Springer International Publishing, 2015.

[4] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, P. Frederickson, T. Lasinski, R. Schreiber, et al. The NAS parallel benchmarks summary and preliminary results. In *Supercomputing*, pages 158–165, 1991.

[5] P. E. Bailey, D. K. Lowenthal, V. Ravi, B. Rountree, M. Schulz, and B. R. de Supinski. Adaptive configuration selection for power-constrained heterogeneous systems. In *International Conference on Parallel Processing*, volume 43, 2014.

[6] K. J. Barker, D. J. Kerbyson, and E. Anger. On the feasibility of dynamic power steering. In *Proceedings of the 2nd International Workshop on Energy Efficient Supercomputing*, pages 60–69. IEEE Press, 2014.

[7] R. Cochran, C. Hankendi, A. K. Coskun, and S. Reda. Pack & cap: adaptive dvfs and thread packing under power caps. In *Proceedings of the 44th annual IEEE/ACM international symposium on microarchitecture*, pages 175–185. ACM, 2011.

[8] M. Curtis-Maury, A. Shah, F. Blagojevic, D. Nikolopoulos, B. de Supinski, and M. Schulz. Prediction models for multi-dimensional power-performance optimization on many cores. In *International Conference on Parallel Architectures and Compilation Techniques*, 2008.

[9] H. David, E. Gorbatov, U. Hanebutte, R. Khanna, and C. Le. RAPL: Memory power estimation and capping. In *ACM/IEEE International Symposium on Low Power Electronics and Design*, pages 189–194. ACM, 2010.

[10] M. Etinski, J. Corbalan, J. Labarta, and M. Valero. Optimizing job performance under a given power constraint in hpc centers. In *Green Computing Conference, 2010 International*, pages 257–267. IEEE, 2010.

[11] M. Etinski, J. Corbalan, J. Labarta, and M. Valero. Linear programming based parallel job scheduling for power constrained systems. In *High Performance Computing and*

Simulation (HPCS), 2011 International Conference on, pages 72–80. IEEE, 2011.

[12] M. Etinski, J. Corbalan, J. Labarta, M. Valero, and A. Veidenbaum. Power-aware load balancing of large scale mpi applications. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–8. IEEE, 2009.

[13] R. Ge, X. Feng, W. Feng, and K. W. Cameron. CPU Miser: A performance-directed, run-time system for power-aware clusters. In *ICPP*, 2007.

[14] Intel. Intel-64 and IA-32 Architectures Software Developer's Manual, Volumes 3A and 3B: System Programming Guide, 2011.

[15] C. Isci, A. Buyuktosunoglu, C. Cher, P. Bose, and M. Martonosi. An analysis of efficient multi-core global power management policies: Maximizing performance for a given power budget. In *IEEE/ACM International Symposium on Microarchitecture*, pages 347–358, 2006.

[16] 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 *Supercomputing*, Nov. 2005.

[17] I. Karlin, J. Keasler, and R. Neely. Lulesh 2.0 updates and changes. Technical Report LLNL-TR-641973, Lawrence Livermore National Laboratory, August 2013.

[18] O. Koné, C. Artigues, P. Lopez, and M. Mongeau. Event-based milp models for resource-constrained project scheduling problems. *Computers & Operations Research*, 38(1):3–13, 2011.

[19] D. Li, B. de Supinski, M. Schulz, K. Cameron, and D. Nikolopoulos. Hybrid MPI/OpenMP power-aware computing. In *IEEE International Parallel and Distributed Processing Symposium*, pages 1–12, 2010.

[20] 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 *International Supercomputing Conference*, 2015.

[21] T. Patki, D. K. Lowenthal, B. Rountree, M. Schulz, and B. R. de Supinski. Exploring hardware overprovisioning in power-constrained, high performance computing. In *Proceedings of the 27th international ACM conference on International conference on supercomputing*, pages 173–182. ACM, 2013.

[22] T. Patki, A. Sasidharan, M. Melarth, D. K. Lowenthal, B. Rountree, M. Schulz, and B. de Supinski. Practical resource management in power-constrained, high performance computing. In *High-Performance Distributed Computing*, June 2015.

[23] B. Rountree, D. K. Lowenthal, B. de Supinski, M. Schulz, and V. W. Freeh. Adagio: Making DVS practical for complex HPC applications. In *International Conference on Supercomputing*, Yorktown Heights, N.Y., USA, June 2009.

[24] B. Rountree, D. K. Lowenthal, S. Funk, V. W. Freeh, B. R. de Supinski, and M. Schulz. Bounding energy consumption in large-scale MPI programs. In *Supercomputing, 2007. SC'07. Proceedings of the 2007 ACM/IEEE Conference on*, pages 1–9. IEEE, 2007.

[25] O. Sarood, A. Langer, A. Gupta, and L. Kale. Maximizing throughput of overprovisioned hpc data centers under a strict power budget. In *Supercomputing*, 2014.

[26] O. Sarood, A. Langer, L. Kalé, B. Rountree, and B. De Supinski. Optimizing power allocation to cpu and memory subsystems in overprovisioned hpc systems. In *CLUSTER*, 2013.

[27] J. Shalf, S. Dosanjh, and J. Morrison. Exascale computing technology challenges. In *High Performance Computing for Computational Science–VECPAR 2010*, pages 1–25. Springer, 2011.

[28] R. F. vanderWijngaart and J. Haopiang. Nas parallel benchmarks, multi-zone versions. 2003.