# Maximizing Throughput of Overprovisioned HPC Data Centers Under a Strict Power Budget

## ABSTRACT

Building future generation supercomputers while constraining their power consumption is one of the biggest challenges faced by the HPC community. For example, US Department of Energy has set a goal of 20 MW for an exascale ($10^{18}$ flops) supercomputer. To realize this goal, a lot of research is being done to revolutionize hardware design to build power efficient computers and network interconnects. In this work, we propose a software-based online resource management system that leverages hardware facilitated capability to constrain the power consumption of each node in order to judicially allocate power and nodes to a job. Our scheme uses this hardware capability in conjunction with an adaptive runtime system that can dynamically change the resource configuration of a running job allowing our resource manager to re-optimize allocation decisions to running jobs as new jobs arrive or a running job terminates.

We also propose a performance modeling scheme that estimates the essential power characteristics of a job at any scale. The proposed online resource manager uses these performance characteristics for making scheduling and resource allocation decisions that maximize the job throughput of the supercomputer under a given power budget. We demonstrate the benefits of our approach by using a mix of jobs with different power-response characteristics. We show that with a power budget of 4.75 MW, we can obtain up to 5.2X improvement in job throughput when compared with the SLURM baseline scheduling policy. We corroborate our results with real experiments on a relatively small scale in which we obtain a 1.7X improvement.

## 1. INTRODUCTION

US Department of Energy has identified four key areas that require significant research breakthroughs in order to achieve exascale computing over the next decade [1]. This paper explores a global power monitoring strategy for high performance computing (HPC) data centers and addresses one of these four key areas - machine operation under power envelope of 20MW. Our work focuses on a hardware-assisted software-based resource management scheme that intelligently allocates nodes and power to jobs with the aim of maximizing the job throughput, under a given power budget. We make a reasonable assumption that it is economical to add any number of nodes to the supercomputer to efficiently utilize the power budgeted for the data center.

Computer subsystems such as CPU and memory have a vendor-specified Thermal Design Power (TDP) that corresponds to the maximal power draw by the subsystem. Currently, maximum power consumption of a HPC data center is determined by the sum of the TDP of its subsystems. This is an overkill, as these subsystems seldom run at their TDP limit. Nonetheless, near TDP amount of power has to be set aside for the subsystems so that the circuit breakers do not trip in that rare case when the power consumption reaches TDP. Recent advances in processor and memory hardware designs have made it possible for the user to control the power consumption of the CPU and memory through software, e.g., the power consumption of Intel's Sandy Bridge family of processors can be user-controlled through the Running Average Power Limit (RAPL) library [2]. Some other examples of such architectures are IBM Power6, Power7, and AMD Bulldozer. This ability to constrain the maximum power consumption of the subsystems below the vendor-specified TDP value allows us to add more machines while ensuring that the total power consumption of the data center does not exceed its power budget. Such a system is called an *overprovisioned* system [3].

Earlier work [3, 4] shows that an increase in the power allowed to the processor (and/or memory) does not yield a proportionate increase in the application's performance. As a result, for a given power budget, it can be better to run an application on larger number of nodes with each node capped at lower power than fewer nodes each running at its TDP. The optimal resource configuration for an application can be determined by profiling an application's performance for varying number of nodes, CPU power and memory power and then selecting the best performing configuration for the given power budget [4]. However, in the case of a data center, there is an additional decision to be made: how to distribute available nodes and power amongst the pending and running jobs. Additionally, new job requests arrive with time and currently running jobs terminate, which requires re-optimization of scheduling and resource allocation decisions. With this context, we believe the major contributions of this paper are:

- An online resource manager (PARM) that uses overprovisioning, power capping and job malleability along with power-response characteristics of each job for scheduling and resource allocation decisions that significantly improves the job throughput of the data center (Section 4).
- A performance model that accurately estimates an applications performance for a given number of nodes and CPU power cap (Section 5). We demonstrate the use of our model by estimating characteristics of five applications having different power-response characteristics (Section 6.3).
- An evaluation of our online resource manager on a 38 node

cluster with two different job data sets. A speedup of 1.7 was obtained when compared with SLURM (Section 6).

- An extensive simulated evaluation of our scheduling policy for larger machines and its comparison with the SLURM baseline scheduling policy. We achieve up to 5.2X speedup operating under a power budget of 4.75 MW (Section 7).

## 2. RELATED WORK

To the best of our knowledge, our study is the first to employ CPU power capping and job malleability for improving throughput of an overprovisioned HPC data center. Patki et al [3] proposed the idea of overprovisioning the compute nodes in power-constrained HPC data centers. They profile an application at different scales and different CPU power caps. They then select the best operating configuration for the application given a power budget. In earlier work, Sarood et. al. [4] extended this idea to include memory power caps and proposed a curve fitting scheme to get an exhaustive profile of an application at various scales, CPU and memory power caps. This profile is then used to obtain the optimal operating configuration of the application under a strict power budget. The novelty of our work in this paper is that it proposes a scheduling scheme to maximize throughput under a strict power budget for a data center scheduling *multiple* jobs simultaneously.

Performance modeling using DVFS has been studied before [5]. Most of the existing research estimates execution time based on CPU frequency. These models can not be used directly in the context of CPU power capping because applications having different memory/CPU characteristics can have processors working at different frequencies while operating under the same CPU power cap. The strong-scaling power aware model proposed in this paper differs from previous work as it estimates execution time of a job for a given package power cap (that includes the power consumption of cores, caches and memory controller present on the chip). There has also been work on developing performance models that capture the energy efficiency of an application [6]. However, our approach is different as it treats power as a *constraint* which is a much harder problem than optimizing energy efficiency.

## 3. DATA CENTER AND JOB CAPABILITIES

In this section, we describe some of the capabilities or features which, according to our understanding, ought to be present in future HPC data centers. In the following sections, we highlight the role these capabilities play for a scheduler, whose goal is to increase the throughput of a data center while ensuring fairness.

**Power capping**: This feature allows the scheduler to constrain the individual power draw of each node. Intel's Sandy Bridge processor family supports on-board power measurement and capping through the RAPL interface [2]. RAPL is implemented using a series of Machine Specific Registers (MSRs) which can be accessed to read power usage for each power plane. RAPL supports power capping Package and DRAM power planes by writing into the relevant MSRs. Here, 'Package' corresponds to the processor chip that hosts processing cores, caches and memory controller. In this paper, we use package power interchangeably with CPU power, for ease of understanding. RAPL can cap power at a granularity of milliseconds which is adequate given that the capacitance on the motherboard and/or power supply smoothes out the power draw at a granularity of seconds.

**Overprovisioning**: By using RAPL to cap the CPU (same as package) power below TDP value, it is possible to add more nodes to the data center while staying within the power budget. An overprovisioned system is thus defined as a system that has more nodes than a conventional system operating under the same power budget. Due to the additional nodes, such a system can not enable all of its nodes to function at their maximum TDP power levels simultaneously.

**Moldable jobs**: In these jobs, user specifies the range of nodes (the minimum and the maximum number of nodes) on which the job can run. The job scheduler decides the number of nodes within the specified range to be allocated to the job. Once decided, the number of nodes cannot be changed during job execution.

**Malleable jobs**: Such jobs can shrink to a smaller number of nodes or expand to a larger number of nodes upon instruction from an external command. Typically, the range of the nodes in which the job can run is dictated by its memory usage and strong scaling characteristics. To enable malleable jobs, two components are critical – *a smart job scheduler*, which decides when and which jobs to shrink or expand, and a *parallel runtime system* which provides dynamic shrink and expand capability to the job. We rely on existing runtime support for malleable jobs in Charm++ [7]. In Charm++, malleability is achieved by dynamically exchanging compute objects between processors at runtime. Applications built on top of such an adaptive system have been shown to shrink and expand with small costs [8]. Charm++ researchers are currently working on further improving the support for malleable jobs. Malleability support in MPI applications has been demonstrated in [9].

## 4. THE RESOURCE MANAGER

Figure 1 shows the block diagram of our online Power Aware Resource Manager, or PARM. It has two major modules: the scheduler and the execution framework. The scheduler is responsible for identifying which jobs should be scheduled and exactly what resources should be devoted to each job. We refer to the resource allocation for each job by the *resource combination* tuple, $(n, p)$, where $n$ is the number of nodes and $p$ is the CPU power cap for each of the $n$ nodes. The scheduling decision is made based on the Integer Linear Program (ILP), and the job profiles generated by our strong scaling power aware model described in Section 5. The scheduler's decisions are fed as input to the execution framework which implements/enforces them by launching new jobs, shrinking/expanding running jobs, and/or setting the power caps on the nodes.

The scheduler is triggered whenever a new job arrives or when a running job ends or abruptly terminates due to an error or any other reason ('Triggers' box in Figure 1). At each *trigger*, the scheduler tries to re-optimize resource allocation to the set of pending as well as currently running jobs with the objective of maximizing overall throughput. Our scheduler uses both CPU power capping and moldability/malleability features for throughput maximization. We formulate this resource optimization problem as an Integer Linear Program (ILP). The relevant terminology is described in Table 1. Our scheduling scheme can be summarized as:

**Input:** A set of jobs that are currently executing or are ready to be executed ($\mathcal{J}$) with their expected execution time

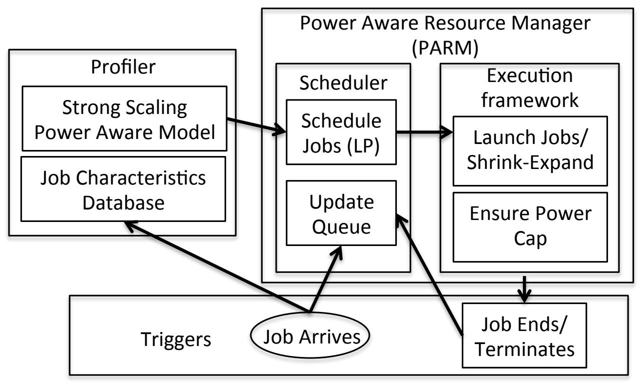

Power Aware Resource Manager (PARM)
Profiler
Strong Scaling Power Aware Model
Job Characteristics Database
Scheduler
Schedule Jobs (LP)
Update Queue
Execution framework
Launch Jobs/ Shrink-Expand
Ensure Power Cap
Triggers
Job Arrives
Job Ends/ Terminates

**Figure 1: A high level overview of PARM**

**Table 1: Integer Linear Program Terminology**

| **Variable** | **Description** |
|---|---|
| $\mathbf{N}$ | total number of nodes in the data center |
| $J$ | set of all jobs |
| $\mathcal{I}$ | set of jobs that are currently running |
| $I$ | set of jobs in the pending queue |
| $\mathcal{J}$ | set of jobs which have already arrived and have not yet been completed i.e they are either pending or currently running, $\mathcal{J} = \mathcal{I} \cup I$ |
| $N_j$ | set of node counts on which job $j$ can be run |
| $P_j$ | set of power levels at which job $j$ should be run or in other words, the power levels at which job $j$'s performance is known |
| $n_j$ | number of nodes at which job $j$ is currently running |
| $x_{j,n,p}$ | binary variable, 1 if job $j$ should run on $n$ nodes at power $p$, otherwise 0 |
| $t_{now}$ | current time |
| $t^a_j$ | arrival time of job $j$ |
| $W_{base}$ | base machine power that includes everything other than CPU and memory |
| $t_{j,n,p}$ | execution time for job $j$ running on $n$ nodes with power cap of $p$ |
| $s_{j,n,p}$ | strong scaling power aware speedup of application $j$ running on $n$ nodes with power cap of $p$ |

corresponding to a set of resource combinations $(n, p)$, where $n \in N_j$ and $p \in P_j$.
**Objective:** Maximize data center throughput.
**Output:** Allocation of resources to jobs at each trigger event, i.e., identifying the jobs that should be executed along with their resource combination *(n,p)*.

## 4.1 Integer Linear Program Formulation

We make the following assumptions and simplifications in the formulation:

- All the nodes of a given job are allocated the same power.
- We do not include cooling power of the data center in our calculations.
- Job characteristics do not change significantly during the course of its execution.
- Expected wall clock time and the actual execution time are equal for the purpose of decision making by the scheduler.
- $W_{base}$, that includes power for all the components of a node other than the CPU and memory subsystems, is assumed to be constant.

*Objective Function*

$$\sum_{j\in\mathcal{J}} \sum_{n\in N_j} \sum_{p\in P_j} w_j * s_{j,n,p} * x_{j,n,p} \qquad (1)$$

*Select One Resource Combination Per Job*

$$\sum_{n\in N_j} \sum_{p\in P_j} x_{j,n,p} \leq 1 \qquad \forall j \in I \qquad (2)$$

$$\sum_{n\in N_j} \sum_{p\in P_j} x_{j,n,p} = 1 \qquad \forall j \in \mathcal{I} \qquad (3)$$

*Bounding total nodes*

$$\sum_{j\in\mathcal{J}} \sum_{p\in P_j} \sum_{n\in N_j} n x_{j,n,p} \leq \mathbf{N} \qquad (4)$$

*Bounding power consumption*

$$\sum_{j\in\mathcal{J}} \sum_{n\in N_j} \sum_{p\in P_j} (n * (p + W_{base}))x_{j,n,p} \leq W_{max} \qquad (5)$$

*Disable Malleability (Optional)*

$$\sum_{n\in N_j} \sum_{p\in P_j} n x_{j,n,p} = n_j \qquad \forall j \in \mathcal{I} \qquad (6)$$

**Figure 2: Integer Linear Program formulation of PARM scheduler**

- A job once selected for execution is not stopped until its completion, although the resources assigned to it can change during its execution.
- All jobs are from a single user (or have the same priority). This condition can be relaxed by introducing appropriate priority factors in the objective function of the ILP.

Scheduling problems are framed as ILPs and ILPs are NP-hard problems. Maximizing throughput in the objective function requires introducing variables for the start and end time of jobs. These variables make the ILP computationally very intensive and thus impractical for online scheduling in many cases. In this work, we propose to drop the job start and end time variables and take a greedy approach by selecting jobs and resource allocations that maximizes the sum of the power-aware speedup (described later) of selected jobs. This objective function improves the job throughput while keeping the ILP optimization computationally tractable for online scheduling.

We define the strong scaling power aware speedup of a job $j$ as follows:

$$s_{j,n,p} = \frac{t_{j,min(N_j),min(P_j)}}{t_{j,n,p}} \qquad (7)$$

where $s_{j,n,p}$ is the speedup of job $j$ executing using resource combination $(n, p)$ with respect to its execution with resource combination $(min(N_j), min(P_j))$. Objective function (Eq. 1) of the ILP maximizes the sum of the power aware speedups of the jobs selected for execution at every trigger event. This leads to improvement in FLOPS/Watt (or power efficiency, as we define it). Improved power efficiency implies better job throughput (results discussed in Section 6, 7). Oblivious maximization of power efficiency may lead to starvation for jobs with low strong scaling power aware speedup. Therefore, to ensure fairness, we introduced a weighing factor $(w_j)$ in the objective function, which is defined as follows:

$$w_j = (t^{rem}_{j,min(N_j),min(P_j)} + (t_{now} - t^a_j))^\alpha \quad (8)$$

$w_j$ artificially boosts the strong scaling power aware speedup of a job by multiplying it to the job's completion time, where completion time is the sum of the time elapsed since job's arrival and the job's remaining execution time with resource combination $(min(N_j), min(P_j))$ i.e. $(t^{rem}_{j,min(N_j),min(P_j)})$ . The percentage of a running job completed between two successive triggers is determined by the ratio of the time interval between the two triggers and the total time required to complete the job using its current resource combination. Percentage of the job that has been completed so far can then be used to compute $t^{rem}_{j,min(N_j),min(P_j)}$. The constant $\alpha$ ($\alpha \geq 0$) in Eq. 8 determines the priority given to job fairness against its strong scaling power aware speedup i.e. a smaller value of $\alpha$ favors job throughput maximization while a larger value favors job fairness.

We now explain the constraints of our ILP (Figure 2):

- Select one resource combination per job (Eq. 2,3): $x_{j,n,p}$ is a binary variable indicating if job $j$ should run using resource combination $(n,p)$. This constraint ensures that at most one of the variables $x_{j,n,p}$ is set to 1 for any job $j$. The jobs which are already running (set $\mathcal{I}$) continue to run although they can be assigned a different resource combination (Eq. 3). The jobs in the pending queue ($I$), for which at least one of the variables $x_{j,n,p}$ is equal to 1 (Eq. 2), are selected for execution and moved to the set of jobs currently running ($\mathcal{I}$).
- Bounding total nodes (Eq. 4): This constraint ensures that the number of active nodes do not exceed the maximum number of nodes available in the overprovisioned data center.
- Bounding power consumption (Eq. 5): This constraint ensures that power consumption of all the nodes does not exceed the power budget of the data center.
- Disable Malleability (Eq. 6): To quantify the benefits of malleable jobs, we consider two versions of our scheduler. The first version supports only moldable jobs and is called as noSE (i.e. no Shrink/Expand), The second version allows both moldable and malleable jobs and is called as wSE (i.e. with Shrink/Expand). Malleability can be disabled by using Eq. 6. This constraint ensures that number of nodes assigned to running jobs does not change during the optimization process. However, it allows changing the power allocated to running jobs. In real-world situations, the jobs submitted to a data center will be a mixture of malleable and non-malleable jobs. The scheduler can apply Eq. 6 to disable malleability for non-malleable jobs.

## 5. STRONG SCALING POWER AWARE MODEL

Recent processors allow power capping, which gives a new dimension to performance modeling. In this section, we propose a strong scaling power aware model, by extending Downey's [10] strong scaling model and making it power aware. The goal is to develop a model that can estimate the execution time of an application for any given resource combination $(n,p)$.

### 5.1 Strong Scaling Model

We used Downey's [10] strong scaling model after modifying the boundary conditions. An application can be characterized by an average parallelism of $A$. The application's parallelism remains equal to $A$, except for a fraction $\sigma$ of total execution time . The variance of parallelism, represented as $V = \sigma(A-1)^2$, depends on $\sigma$, where $0 \leq \sigma \leq 1$. The execution time, $t(n)$, of an application can then be defined as follows:

$$t(n) = \begin{cases} \dfrac{T_1 - \frac{T_1\sigma}{2A}}{n} + \dfrac{T_1\sigma}{2A}, & 1 \leq n \leq A \quad (9) \\ \dfrac{\sigma(T_1 - \frac{T_1}{2A})}{n} + \dfrac{T_1}{A} - \dfrac{T_1\sigma}{2A} & A < n \leq 2A-1 \quad (10) \\ \dfrac{T_1}{A}, & n > 2A-1 \quad (11) \end{cases}$$

where $n$ is the number of nodes and $T_1$ is the execution time on a single node. Since we have to estimate execution time corresponding to different number of nodes $n$ given $t(1) = T_1$, we had to modify Downey's model according to boundary condition $t(1) = T_1$. The first equation in this group represents the range of $n$ where the application is most scalable. This is the part where the number of nodes is less than $A$, i.e., the average amount of parallelism. The application's scalability declines significantly once $n$ becomes larger than $A$ because some nodes are unable to do work in parallel owing to lack of parallelism. Finally, for $n \geq 2A$, the execution time $t(n)$ equals $T_1/A$ and does not decrease. Given application characteristics $\sigma$, $A$, and $T_1$, this model can be used to estimate execution time for any number of nodes $n$.

### 5.2 Adding Power Awareness to Strong Scaling Model

The effect of increasing frequency on the execution time $t$ varies from application to application [11] . In this section, we first describe a basic framework that models $t$ as a function of CPU frequency $f$. Since, $f$ can be expressed as a function of CPU power $p$, we can finally express $t$ in terms of $p$.

#### *5.2.1 Execution Time as a Function of Frequency*

Existing work [11] indicates that increase in CPU frequency beyond a certain threshold frequency, $f_h$, does not reduce the execution time $t$. The value of $f_h$ depends on the memory bandwidth being used by the application. Since $t \propto \frac{1}{f}$, we can express $t$ as [5]:

$$t(f) = \begin{cases} \dfrac{W}{f} + T, & \text{for } f < f_h \quad (12) \\ T_h, & \text{for } f \geq f_h \quad (13) \end{cases}$$

where $W$ and $T$ are constants that roughly correspond to the CPU and memory bounded work respectively. $T_h$ is the execution time at frequency $f_h$. Given that $f_l$ is the lowest possible frequency a CPU can operate at, Eq. 12 should obey the boundary conditions $t(f_l) = T_l$ and $t(f_h) = T_h$.

Parameter $\beta$ characterizes the frequency-sensitivity of an application and can be expressed as:

$$\beta = \frac{T_l - T_h}{T_l} \quad (14)$$

The range of $\beta$ depends on the CPU's DVFS range. Given the DVFS range of $(f_l, f_{max})$, $\beta \leq 1 - \frac{f_l}{f_{max}}$. Typically, CPU-bound applications have higher values for $\beta$ whereas memory-intensive applications have smaller $\beta$ values.

Using Eq. 14 and subjecting Eq. 12 to boundary conditions, $t(f_l) = T_l$ and $t(f_h) = T_h$, gives us:

$$W = \frac{T_h \beta f_l f_h}{(1-\beta)(f_h - f_l)} \tag{15}$$

$$T = T_h - \frac{T_h \beta f_l}{(1-\beta)(f_h - f_l)} \tag{16}$$

### 5.2.2 *Execution Time as a Function of CPU Power*

Although Intel has not released complete details of how the CPU power cap is ensured, it has been hinted that this is achieved using a combination of DVFS and CPU throttling. Core input voltage and frequency can be set within manufacturer defined ranges. This, in turn, defines a range over CPU power capping that can be achieved using DVFS. Let $p_l$ denote the CPU power corresponding to $f_l$, where $f_l$ is the minimum frequency the CPU can operate at using DVFS. Beyond $p_l$, power is reduced by mechanisms other than DVFS, e.g., CPU throttling. The threshold $p_l$, where CPU throttling takes over from DVFS, can be determined by looking at the processor's operating frequency. CPU or package power includes the power consumption by its various components such as cores, caches, memory controller, etc. The value of $p_l$ varies depending on an application's usage of these components. In a CPU bound application, a processor might be able to cap power to lower values using DVFS, since only the cores are consuming power. In contrast, for a memory intensive application, $p_l$ might be higher, since the caches and memory controller are also consuming power in addition to the cores. Since core input voltage is proportional to $f$, power consumption of the cores, $p_{core}$, can be modeled as:

$$p_{core} = Cf^3 + Df \tag{17}$$

where $C$ and $D$ are constants. Since the number of cache and memory accesses are proportional to frequency, DVFS can change cache and memory controller power consumption as well. The CPU power can then be expressed as [12]:

$$p = p_{core} + \sum_{i=1}^{3} g_i L_i + g_m M + p_{base} \tag{18}$$

where $L_i$ is accesses per second to level $i$ cache, $g_i$ is the cost of a level $i$ cache access, $M$ is the number of memory accesses per second, $g_m$ is the cost per memory access, and $p_{base}$ is the base package power consumption. Eq. 18 can also be written as:

$$p = F(f) = af^3 + bf + c \tag{19}$$

where $a$, $b$, and $c$ are constants. In Eq. 19, the term $bf$ corresponds to the cores' leakage power and the power consumption of caches and memory controller. The term $af^3$ represents the dynamic power of the cores, whereas $c = p_{base}$ is the base CPU power. The constants $a$ and $b$ are application dependent since the cache and memory behavior can be different across applications. Eq. 19 can be rewritten as a depressed cubic equation and solved using Fermat's Last Theorem to get $F^{-1}$:

$$\begin{aligned} f = F^{-1}(p) \;&=\; \sqrt[3]{\frac{p-c}{2a} + \sqrt{\frac{(p-c)^2}{4a^2} + \frac{b^3}{27a^3}}} \\ &+\; \sqrt[3]{\frac{c-p}{2a} + \sqrt{\frac{(p-c)^2}{4a^2} + \frac{b^3}{27a^3}}} \end{aligned} \tag{20}$$

To express $t$ in terms of $p$, we use Eq. 20 to replace $f$, $f_l$, and $f_h$ in Eqs. 12, 15, 16. To combine our power aware model with the strong scaling model described in Section 5.1, we replace $T_h$ in Eqs. 12, 15, 16 with $t(n)$ from Eqs. 9, 10, 11. **Summary:** We present a comprehensive model to estimate $t$ for any resource combination $(n, p)$, given application parameters $\sigma, T_1, A, p_l, p_h, \beta, a$ and $b$. We substitute parameters $\sigma, T_1$ and $A$ into Eqs. 9, 10, 11 to determine $t = t(n)$, i.e., execution time using $n$ nodes working at maximum power. We then use $T_h = t(n)$ and parameters $p_l, p_h, \beta, a$ and $b$ to determine $t(n, p)$ using Eqs. 12, 15, 16, and 20, i.e., execution time using $n$ nodes operating at $p$ Watts each.

## 6. EXPERIMENTAL RESULTS

In this section, we describe our experimental setup that includes applications, testbed, and job datasets. We obtain the application characteristics using the power-aware strong-scaling performance model and finally compare the performance of the noSE and wSE versions of PARM with SLURM.

### 6.1 Applications

We used five applications, namely, Wave2D, Jacobi2D, LeanMD, Lulesh [13], and Adaptive Mesh Refinement or AMR [14]. These applications have different CPU and memory usage:

- Wave2D and Jacobi2D are 5-point stencil applications that are memory-bound. Wave2D has higher FLOPS than Jacobi2D.
- LeanMD is a computationally intensive molecular dynamics application.
- CPU and memory usage of Lulesh and AMR lies in between the stencil applications and LeanMD.

### 6.2 Testbed

We conducted our experiments on a 38-node Dell PowerEdge R620 cluster (which we call the Power Cluster). Each node containing an Intel Xeon E5-2620 Sandy Bridge with 6 physical cores at 2GHz, 2-way SMT with 16 GB of RAM. These machines support on-board power measurement and capping through the RAPL interface [15]. The CPU power for our testbed can be capped in the range $[25-95]W$, while the capping range for memory power is $[8-35]W$.

### 6.3 Obtaining Model Parameters of Applications

Application characteristics depend on the input type, e.g., gird size. We fix the respective input types for each application. Each application needs to be profiled for some (n,p) combinations to obtain data for curve fitting. 1 step/iteration is sufficient to get the performance for a given data point. Since, a step/iteration is usually of the order of milliseconds, the cost of profiling the application at several data points is negligible compared to the overall execution time of the application.

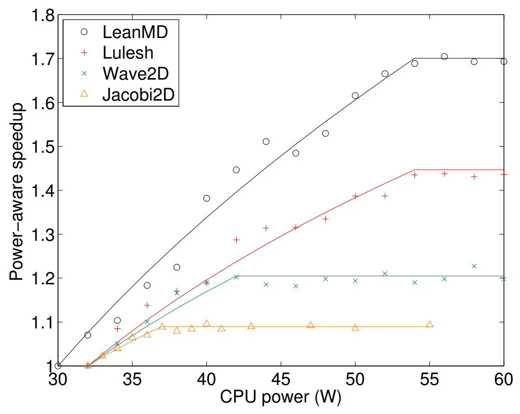


**Figure 3: Modeled (lines) and observed (markers) power aware speedups for 4 applications**

**Table 2: Obtained model parameters**

| **Application** | **a** | **b** | $p_l$ | $p_h$ | $\beta$ |
|---|---|---|---|---|---|
| **LeanMD** | 1.65 | 7.74 | 30 | 52 | 0.40 |
| **Wave2D** | 3.00 | 10.23 | 32 | 40 | 0.16 |
| **Lulesh** | 2.63 | 8.36 | 32 | 54 | 0.30 |
| **AMR** | 2.45 | 6.57 | 32 | 54 | 0.33 |
| **Jacobi2D** | 1.54 | 10.13 | 32 | 37 | 0.08 |

We use linear and non-linear regression tools provided by MATLAB to determine the application parameters by fitting the sampled application performance data to the performance model proposed in Section 5. The obtained parameter values for all the applications are listed in Table 2 and are discussed here:

- The parameter $c$ (CPU base power) lies in the range $[13-14]$W for all applications
- $p_l$ was 30W for LeanMD and $32W$ for rest of the applications. For LeanMD, it is possible to cap the CPU power to a lower value just by decreasing the frequency using DVFS. This is because LeanMD is a computationally intensive application and therefore most of the power is consumed by the cores rather than caches and memory controller. On the contrary, for other applications, CPU throttling kicks in at a higher power level because of their higher cache/memory usage.
- value of $p_h$ lies in the range of $[37-54]W$ for the applications under consideration.
- value of $\beta$ lies in the range $[0.08-0.40]$. Higher value of $\beta$ means higher sensitivity to CPU power.
- Wave2D and Jacobi2D have the largest memory footprint that results in high CPU-cache-memory traffic. Therefore the value of $b$ is high for these two applications.

Figure 3 shows the modeled (lines) as well as the observed (markers) power-aware speedups for 4 applications under consideration. Since *AMR*'s characteristics are very similar to *Lulesh*, we leave *AMR* out to improve the clarity of the figure. Each application's speedup was calculated with respect to the execution time when that application was executing at $p = p_l$. LeanMD has the highest power-aware speedup whereas Jacobi2D has the lowest.

## 6.4 Power Budget

We assume a power budget of 3300W to carry out experiments using our Power Cluster. Although the vendor-specified TDP of CPU and memory of the Dell nodes was 95W and 35W, respectively, the actual power consumption of CPU and memory never went beyond 60W and 18W, respectively, when running any of the applications. Therefore, instead of the vendor-specified TDP, we consider 60W and 18W as the maximum CPU and memory power consumption and use them to calculate the number of nodes that can be installed in a traditional data center. The maximum power consumption of a node, thus, adds up to $60W + 18W + 38W = 116W$, where $38W$ is the base power of a node. Therefore, the total number of nodes that can be installed in a traditional data center with a power budget of 3300W will be $\lfloor \frac{3300}{116} \rfloor = 28$ nodes. By capping the CPU power below 60W, the overprovisioned data center will be able to power more than 28 nodes.

## 6.5 Job Datasets

We constructed two job datasets by choosing a mix of applications from the set described in Section 6.1. All these applications are written using the Charm++ parallel programming model and hence support job malleability. Application's power-response characteristics can influence the benefits of PARM. Therefore, in order to better characterize the benefits of PARM, these two job datasets were constructed such that they have very different average values of $\beta$. We name these datasets as SetL and SetH, with average $\beta$ value of 0.1 and 0.27, respectively. For instance, SetH has 3 LeanMD, 3 Wave2D, 2 Lulesh, 1 Jacobi, and 1 AMR job, that gives us an average $\beta$ value of 0.27. A mix of short, medium and long jobs were constructed by randomly generating wall clock times with a mean value of 1 hour. Similarly, the job arrival times were generated randomly. Each dataset spans over 5 hours of cluster time and approximately 20 scheduling decisions were taken (a scheduling decision is taken whenever a new job arrives or a running job terminates). The minimum and the maximum number of nodes on which a job can run was determined by the job's memory requirements. We used 8 node levels (i.e. $|N_j| = 8$) that are uniformly distributed between the minimum and maximum number of nodes on which the job can run. The memory power is capped at the fixed value of 18W whereas we used 6 CPU power levels - $[30, 32, 34, 39, 45, 55]$W.

## 6.6 Performance Metric

We compare our scheduler with SLURM [16]: an open-source resource manager that allocates compute nodes to jobs and provides a framework for starting, executing and monitoring jobs on a set of nodes. SLURM provides resource management on many of the most powerful supercomputers of the world including Tianhe-1A, Tera 100, Dawn, and Stampede. We setup both PARM and SLURM on the testbed. For comparison purpose, we use SLURM's baseline scheme in which the user specifies the exact number of nodes requested for the job and SLURM uses FIFO + backfilling for making scheduling decisions. We call this as the SLURM baseline scheme or just the baseline scheme. The number of nodes requested for a job submitted to SLURM is the minimum number of nodes on which PARM can run that job.

We use response time and completion time as the metric for comparing PARM and SLURM. A job's response time,

$t_{res}$, is the time interval between its arrival and the beginning of its execution. Execution time, $t_{exe}$, is the time from start to finish of a job's execution. Completion time, $t_{comp}$, is the time between job's arrival and the time it finished execution, i.e., $t_{comp} = t_{res} + t_{exe}$. Job throughput is the inverse of the average completion time of jobs. In this study, we emphasize on completion time as the performance comparison metric, even though typically response time is the preferred metric. This is because unlike conventional data centers, where resources allocated to a job and hence the jobs execution time are fixed, our scheduler dynamically changes job configuration during execution which can vary job execution time significantly. Hence, response time is not a very appropriate metric for comparison in this study. Completion time includes both the response time and the execution time and is therefore the preferred metric of comparison.

### 6.7 Results

Figure 4(a) shows the average completion times of the two datasets with SLURM and the noSE and wSE versions of PARM. The completion times for *wSE* and *noSE* include all overhead costs including the ILP optimization time and the costs of constriction and expansion of jobs. As noted in the figure, PARM significantly reduces the average completion time for both the data sets. This improvement can mainly be attributed to the reduced average response times shown in Figure 4(b). Our scheduler intelligently selects the best power levels for each job which allows it to add more nodes to benefit from strong scaling and/or scheduling more jobs simultaneously. The completion times of wSE scheme are better than noSE. Job malleability allows wSE scheme to shrink and expand jobs at runtime. This gives flexibility to the ILP to re-optimize the allocation of nodes to the running and pending jobs. On the other hand, noSE reduces the solution space of ILP by not allowing running jobs to change the number of nodes allocated to them. Additionally, wSE increases the machine utilization towards the tail of the job dataset execution when there are very few jobs left running. The wSE version can expand the running jobs to run on the unutilized machines. These factors reduce both the average completion and the average response time in wSE (Figure 4(a), 4(b)). As shown by Figure 4(c), wSE scheme utilizes 36 nodes on an average compared to an average of 33 nodes used in the case of noSE for SetL.

A smaller value of $\beta$ means that effect of decreasing the CPU power on application performance is small. When $\beta$ is small, the scheduler will prefer to allocate less CPU power and use more nodes. On the other hand, when $\beta$ is large, the benefits of adding more nodes at the cost of decreasing the CPU power are smaller. The flexibility to increase the number of nodes gives PARM higher benefit over SLURM when $\beta$ is small as compared to the case when $\beta$ is large. Therefore, lower the sensitivity of applications to the allocated CPU power (i.e. smaller value of $\beta$), higher will be the benefit of using PARM. This is corroborated with the observation (Figure 4) that the benefits of using PARM as compared to SLURM are much higher with dataset SetL ($\beta = 0.1$) as compared to dataset SetH ($\beta = 0.27$).

## 7. LARGE SCALE PROJECTIONS

After experimentally showing the benefits of PARM on a real cluster, we now analyze its benefit on very large machines. Since it was practically infeasible for us to do actual job scheduling on very large machine, we use the SLURM simulator [17] which is a wrapper around SLURM. This simulator gives us information about SLURM's scheduling decisions without actually executing the jobs. In the following subsections, we describe our shrink/expand cost model, give the experimental setup and then present a comparison of PARM scheduling with SLURM baseline scheduling policy.

### 7.1 Modeling Cost of Shrinking and Expanding Jobs

Constriction and expansion of jobs has an overhead associated with it. These overheads come from data communication done to balance the load across the new set of processors assigned to the job and from the boot time of nodes. A scheduler typically makes two decisions: 1) how many nodes to assign to each job, and 2) which nodes to assign to each job. We address the first decision in this paper and defer the second for future work. Let us say that job $j$ with a total memory of $m_j$ MB, has to expand from $n_f$ nodes to $n_t$ nodes. For simplification of analysis, we assume that each job is initially allocated a cuboid of nodes (with dimensions- $\sqrt[3]{n_f} \times \sqrt[3]{n_f} \times \sqrt[3]{n_f}$) interconnected through a 3D torus. After the expand operation, size of the cuboid becomes $\sqrt[3]{n_f} \times \sqrt[3]{n_f} \times \frac{n_t}{\sqrt[3]{n_f}^{2}}$. For load balance, the data in memory ($m_j$ MB) will be distributed equally amongst the $n_t$ nodes. Hence, the communication cost for the data transfer can be expressed as (secs):

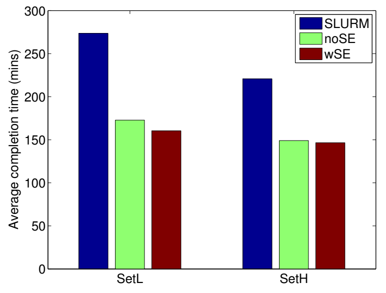
$$t_c = \frac{(\frac{m_j}{n_f} - \frac{m_j}{n_t}) * n_f}{2 * b * n_f^{\frac{2}{3}}} \tag{21}$$

where $b$ is the per link bandwidth in MB/sec. The numerator in Eq. 21 represents the total data to be transferred whereas the denominator represents the bisection bandwidth of the cuboid. Similarly, the cost of shrinking a job is determined by computing the cost of distributing the data of $n_f - n_t$ nodes equally across the final $n_t$ nodes.

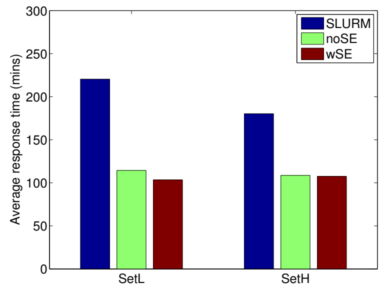
Boot times can be significant for some supercomputers. Since many supercomputers in Top500 [18] belong to the Blue Gene family, we include their boot time when evaluating our scheme. We adopt a simple linear model to calculate the boot time ($t_b$) for expand operation. The following linear relationship is obtained by using the Intrepid boot time data [19]:

$$t_b(\text{in seconds}) = (n_t - n_f) * 0.01904 + 72.73 \tag{22}$$

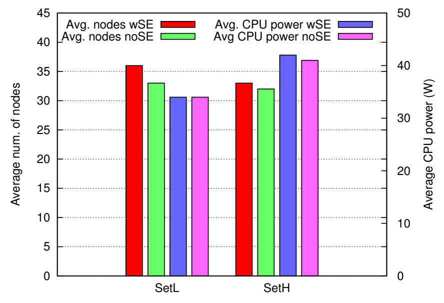
In an expand operation, communication phase can start only after additional nodes become available. These additional nodes might have to be booted. Therefore the total cost of a shrink or expand operation is sum of the boot time and the data transfer time, i.e., $t_{se} = t_c + t_b$. A job set for expansion might receive additional nodes from a job undergoing constriction in the same scheduling decision. Therefore, an expanding job has to wait until the shrinking job has released the additional resources. To simplify this analysis, we determine the maximum $t_{se}$ from amongst the shrinking/expanding jobs ($t_{se}^{max}$) and add $2t_{se}^{max}$ to the execution times of all the jobs shrinking or expanding during the current scheduling decision. To control the frequency of constriction or expansion of a job, and consequently its cost, we define a parameter $f_{se}$ (in secs). $f_{se}$ is the time after which a job can shrink or expand. i.e. if a job was shrunk or expanded at $t$ secs, then it can be shrunk or expanded only after $t + f_{se}$ secs. This condition is enforced using Eq. 6.



(a) Average completion time



(b) Average response time



(c) Average nodes and average power per node

**Figure 4: (a) Average completion times, and (b) average response times for SetL and SetH with SLURM and noSE, wSE versions of PARM. (c) Average number of nodes and average CPU power in the wSE and noSE versions of PARM.**

## 7.2 Experimental Setup

The results presented in this section are based on the job logs [20] of Intrepid. Intrepid is a IBM BG/P supercomputer with a total of $40,960$ nodes, installed at Argonne National Lab. The job trace spans over 8 months and has 68936 jobs. We extracted 3 subsets of 1000 successive jobs each from the trace file to conduct our experiments. These subsets will be referred to as Set1, Set2, and Set3 and the starting job id for these subsets are 1, 10500, and 27000, respectively. To measure the performance of PARM in the wake of diverse job arrival rates, we generated several other datasets from each of these sets by multiplying the arrival times of each job by $\gamma$, where $\gamma \in [0.2 - 0.8]$. Multiplication of the arrival times with $\gamma$ increases the job arrival rate without changing the distribution of job arrival times.

As we do not know application characteristics ($\sigma, T_1, A, f_l, f_h, \beta, a$ and $b$) of the jobs in the Intrepid trace file, we take the parameter values from Section 6.3 and randomly assign values from these ranges to jobs in the Intrepid trace file. Since Intrepid does not allow moldable/malleable jobs, jobs request a fixed number of nodes instead of a range of nodes. For jobs submitted to the PARM scheduler, we consider this number as the maximum nodes that the job should be run on, i.e., $max(N_j)$ and set $min(N_j) = \theta * max(N_j)$, where $\theta$ is randomly selected in the range $[0.2 - 0.6]$. The power consumption of Intrepid nodes is not publicly available, therefore we use the power values from our testbed cluster (described in Section 6.2). Hence, the maximum power consumption per node is taken to be 116W. The maximum power consumption of $40,960$ nodes thus equals $116 \times 40960 = 4751360$W. The SLURM scheduler will schedule on 40960 nodes with each node running at maximum power level. As in Section 6.5, PARM uses 6 CPU power levels, $P_j = \{30, 33, 36, 44, 50, 60\}W$.

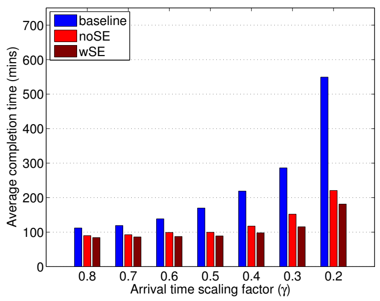
## 7.3 Performance Results

Both noSE and wSE significantly reduce average completion times compared to SLURM's baseline scheduling policy (Figure 5). As $\gamma$ decreases from 0.8 to 0.2, the average completion time increases in all the schemes because the jobs arrive at a much faster rate and therefore have to wait in the queue for longer time before they get scheduled. However, this increase in the average completion times with both our schemes is not as significant as it is with the baseline scheme.

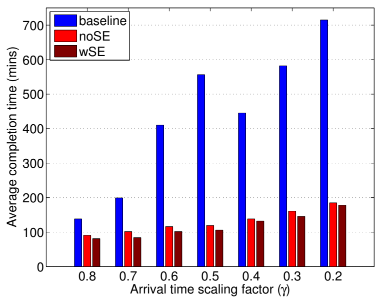
Both noSE and wSE have significantly improved average response times, with wSE outperforming noSE (Table 3). Execution time in wSE includes the costs of shrinking or expanding the job (Section 7.1). Despite this overhead, wSE outperforms noSE in average execution time of jobs. wSE version consistently outperforms noSE in all data sets. The average completion times reported in Figure 5) includes cost of all overheads. In all the job datasets, the average overhead for shrinking and expanding the jobs was less than 1% of the time taken to execute the dataset. We controlled these costs by setting $f_{se} = 500$ secs, i.e., the scheduler waited for at least 500 secs between two successive shrink and expand operations for a job. We found the cost for solving the ILP to be very small. The largest ILP that we solved took 15 secs which is negligible given the frequency at which the scheduler was invoked is much smaller. We use data from Table 3 to explain two observations related to the speedups of our schemes. The speedup in average completion time for both our policies is calculated relative to the baseline (shown in Table 3).

- Higher speedup in average completion time for $\gamma = 0.5$ compared to $\gamma = 0.7$ in Set2: The baseline case has a 4 fold increase in response time compared to just a 2 fold increase in response time for wSE when $\gamma$ changes from 0.7 to 0.5. Average execution time in wSE version increases only slightly after $\gamma$ changes from 0.7 to 0.5 (Table 3), while it remains constant in the baseline scenario. Since, completion time is the sum of response and execution time, we observe higher speedups for smaller values of $\gamma$ in Set2.
- Smaller speedups in Set3 as compared to Set2: Upon job trace inspection, we discovered that there are not enough jobs to keep the machine fully utilized during the first half of Set3. Therefore, even after reducing $\gamma$ to small values, the response time is not significantly affected in Set3. In contrast, Set2 has a very high machine utilization which leads to a significant increase in the response time of the baseline scheduler as $\gamma$ is reduced (from 0.5 to 0.7), resulting in higher speedups for both our scheduling policies.

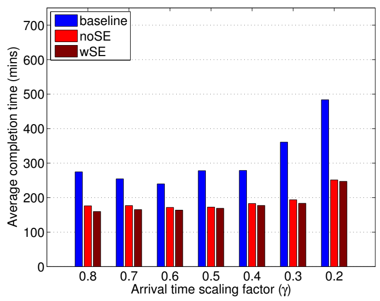
To further see the effectiveness of our scheme, we compared it with a baseline scheduler if it were to schedule jobs on an overprovisioned system. All the CPUs in this system are power capped at the same value ($< 60$W). This allows the baseline scheduler to add more nodes while remaining within the same power budget, e.g., setting the power cap for each CPU to 30W allows baseline (SLURM) to use up to $\lfloor \frac{4751360}{30+18+38} \rfloor = 55248$ nodes. In Table 4, we present the speedup of wSE relative to a baseline scheduler operating on an overprovisioned system with different CPU power caps



(a) Set1



(b) Set2



(c) Set3

**Figure 5: Average completion times of baseline, *noSE* and *wSE*. Job arrival times in all the sets (Set1, Set2, Set3) were scaled down by factor $\gamma$ to get diversity in job arrival rate**

**Table 3: Average response times, average execution times, average number of nodes and speedup in average completion times for the baseline scheduling policy, wSE, and noSE for different data sets.**

| Set | Avg Resp. Time (mins) | | | Avg Exe. Time (mins) | | | Avg. Num. of Nodes | | | Speedup | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | baseline | wSE | noSE | baseline | wSE | noSE | baseline | wSE | noSE | wSE | noSE |
| **1** ($\gamma = 0.5$) | 90 | 3 | 6 | 80 | 84 | 95 | 453 | 610 | 601 | 1.91 | 1.70 |
| **2** ($\gamma = 0.5$) | 500 | 34 | 57 | 57 | 69 | 89 | 632 | 714 | 721 | 5.25 | 4.66 |
| **3** ($\gamma = 0.5$) | 217 | 99 | 88 | 60 | 73 | 90 | 520 | 662 | 665 | 1.65 | 1.61 |
| **2** ($\gamma = 0.7$) | 142 | 12 | 20 | 57 | 66 | 83 | 596 | 656 | 660 | 2.36 | 1.96 |
| **3** ($\gamma = 0.7$) | 194 | 95 | 86 | 60 | 73 | 90 | 488 | 596 | 599 | 1.54 | 1.43 |

**Table 4: Comparison of $wSE$ with the baseline scheduler running on an overprovisioned system (at different CPU power caps) using Set 2 ($\gamma = 0.5$)**

| CPU power cap (W) | 30 | 40 | 50 | 60 |
|---|---|---|---|---|
| **Speedup** | 4.32 | 1.86 | 2.33 | 5.25 |
| **Avg number of nodes** | 50332 | 42486 | 39700 | 37956 |

as the reference. Significant speedups in Table 4 emphasize the benefits of solving the ILP for determining optimal job configurations rather than using the baseline scheduler which runs all jobs at the same node power. This shows that even after scheduling on an overprovisioned system and using greater than 40960 nodes, the baseline scheduler still under-performs our scheduler by a significant margin.

## 7.4 Analyzing Tradeoff Between Fairness and Throughput

We introduced the term $w_j$ in the objective function of the scheduler's ILP (Eq. 1) to prevent starvation of jobs with low power-aware speedup. In this section, we analyze the trade-off between maximum completion time of any job (fairness) and the average completion time of the jobs (equivalent to throughput). The parameter $\alpha$ can be be tuned by the data center administrator to control fairness and throughput. We performed several experiments by varying $\alpha$ and measured its impact on average and maximum completion times. Notice in Figure 6, as the value of $\alpha$ increases, the maximum completion time decreases at the cost of increase in average completion time.

## 7.5 How Much Profile Data is Sufficient?

Our scheduler's ILP takes a job's execution profile at different resource combinations as input. Larger number of power levels $|P_j|$, will lead to better scheduling decisions compared to the case when fewer number of power levels are provided. However, the number of binary variables in the ILP ($x_{j,n,p}$) are proportional to the number of power levels of the jobs. The larger the number of variables, the longer it takes to solve the ILP. As mentioned earlier, the maximum cost of solving the ILP in all of our experiments was 15 secs where we used $|P_j|= 6$ and the queue $\mathcal{J}$ had 200 jobs. In this subsection, we see the impact of varying the number of power levels used per job. We did several experiments where we scheduled Set1 with $\gamma = 0.5$ using up to 8 different power levels, $|P_j|= 8$. For example, the case with 2 power levels means that all jobs can execute either at 30W or 60W. The average and maximum completion times with the baseline scheduler for Set1 ($\gamma = 0.5$) were 170 mins and $1,419$ mins, respectively. As we keep on increasing the number of power levels from 1 to 6, the solution space of the ILP increases and therefore we get better solutions, i.e., the average and maximum completion times decrease as the number of power levels increase (Figure 7). However, the improvement in both the average and maximum completion times becomes negligible after 6 power levels.

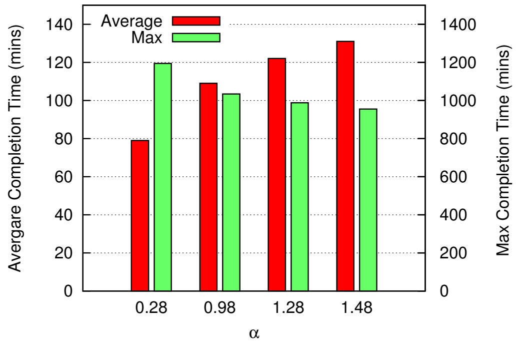


**Figure 6: Average (left axis) and maximum (right axis) completion times for Set 1 for different values of ($\alpha$)**

## 8. CONCLUSIONS AND FUTURE WORK

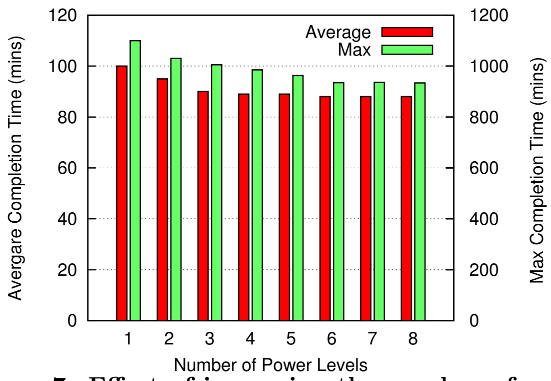


**Figure 7: Effect of increasing the number of power levels ($|P_j|$) on the average and maximum completion time of Set 1 ($\gamma = 0.5$). There is negligible improvement in performance after 6 power levels**

To the best of our knowledge, this is the first study that proposes an online power-aware resource manager that uses CPU power capping and job malleability to maximize job throughput of an overprovisioned HPC data center under a strict power budget. We formulate our scheduling problem as an ILP that is computationally tractable for online scheduling. We propose a strong scaling power aware model that generates the execution time profile of an application by obtaining its power and strong scaling characteristics. We first evaluate our scheme on real hardware and then project its benefits on larger machines. Job trace data of a large supercomputer was used to obtain the distribution of job arrival times and wall clock times for a real workload. We demonstrate the benefits of PARM by comparing it with SLURM that is used in conventional HPC data centers. Our results show that both our schemes i.e, with and without malleable jobs, can achieve a speedup of up to 4.5X in average completion time over SLURM's baseline strategy.

Thermal behavior of CPUs can significantly affect the reliability of the machine as well as the cooling costs of the data center. Two compute nodes operating under the same CPU power cap might end up working at different *speeds* due to different thermal conditions. In future, we plan to address this *heterogeneity* by using dynamic load balancing capabilities of a system that supports dynamic object migration [21]. We also plan to investigate the possibility of incorporating thermal constraints along with a strict power constraint in our scheduling scheme.

## 9. REFERENCES

[1] "2013 Exascale Operating and Runtime Systems," http://science.doe.gov/grants/pdf/LAB_13-02.pdf, Office of Advanced Science Computing Research (ASCR), Tech. Rep.

[2] B. Rountree, D. H. Ahn, B. R. de Supinski, D. K. Lowenthal, and M. Schulz, "Beyond DVFS: A First Look at Performance Under a Hardware-enforced Power Bound," in *IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW)*, 2012.

[3] 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*, ser. ICS '13, New York, NY, USA, 2013.

[4] O. Sarood, A. Langer, L. V. Kale, B. Rountree, and B. d. Supinski, "Optimizing Power Allocation to CPU and Memory Subsystems in Overprovisioned HPC Systems," in *Proceedings of IEEE Cluster 2013*, Indianapolis, IN, USA, September 2013.

[5] C.-H. Hsu and W.-C. Feng, "Effective Dynamic Voltage Scaling through CPU-Boundedness Detection," in *Proceedings of the 4th International Conference on Power-Aware Computer Systems*, ser. PACS'04, 2005.

[6] R. Ge, X. Feng, and K. Cameron, "Modeling and Evaluating Energy-performance Efficiency of Parallel Processing on Multicore Based Power Aware Systems," in *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, 2009, pp. 1–8.

[7] L. Kalé, "The Chare Kernel Parallel Programming Language and System," in *Proceedings of the International Conference on Parallel Processing*, vol. II, Aug. 1990, pp. 17–25.

[8] L. V. Kalé, S. Kumar, and J. DeSouza, "A Malleable-Job System for Timeshared Parallel Machines," in *2nd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid 2002)*, May 2002.

[9] M. C. Cera, Y. Georgiou, O. Richard, N. Maillard, and P. O. A. Navaux, "Supporting Malleability in parallel Architectures with Dynamic CPUSETs Mapping and Dynamic MPI," in *Proceedings of the 11th International Conference on Distributed Computing and Networking*, ser. ICDCN'10, 2010.

[10] A. B. Downey, "A Model for Speedup of Parallel Programs," Tech. Rep., 1997.

[11] O. Sarood and L. Kale, "Efficient Cool Down of Parallel Applications," *Workshop on Power-Aware Systems and Architectures in conjunction with International Conference on Parallel Processing*, 2012.

[12] X. Chen, C. Xu, and R. Dick, "Memory Access Aware on-line Voltage Control for Performance and Energy Optimization," in *Computer-Aided Design (ICCAD), 2010 IEEE/ACM International Conference on*, 2010, pp. 365–372.

[13] I. Karlin, A. Bhatele, J. Keasler, B. Chamberlain, J. Cohen, Z. DeVito, R. Haque, D. Laney, E. Luke, F. Wang *et al.*, "Exploring Traditional and Emerging Parallel Programming Models using a Proxy Application," in *International Parallel and Distributed Processing Symposium*, 2013.

[14] A. Langer, J. Lifflander, P. Miller, K.-C. Pan, , L. V. Kale, and P. Ricker, "Scalable Algorithms for Distributed-Memory Adaptive Mesh Refinement," in *Proceedings of the 24th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD 2012). To Appear*, New York, USA, October 2012.

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

[16] M. A. Jette, A. B. Yoo, and M. Grondona, "SLURM: Simple Linux Utility for Resource Management," in *In Lecture Notes in Computer Science: Proceedings of Job Scheduling Strategies for Parallel Processing (JSSPP) 2003*. Springer-Verlag, 2002, pp. 44–60.

[17] A. Lucero, "Slurm Simulator," http://www.bsc.es/marenostrum-support-services/services/slurm-simulator, Tech. Rep.

[18] "Top500 supercomputing sites," http://top500.org, 2013.

[19] ANL, "Running Jobs on BG/P systems," https://www.alcf.anl.gov/user-guides/bgp-running-jobs#boot-time.

[20] "Parallel Workloads Archive," http://www.cs.huji.ac.il/labs/parallel/workload/, Tech. Rep.

[21] L. Kalé and S. Krishnan, "Charm++ : A Portable Concurrent Object Oriented System Based on C++," in *Proceedings of the Conference on Object Oriented Programmi ng Systems, Languages and Applications*, September 1993.