Power-Constrained Performance Scheduling of Data Parallel Tasks

Eric Anger
School of Electrical and
Computer Engineering
Georgia Institute of Technology
Atlanta, Georgia
eanger@gatech.edu

Jeremiah Wilke
Sandia National Laboratories
Livermore, California
jjwilke@sandia.gov

Sudhakar Yalamanchili
School of Electrical and
Computer Engineering
Georgia Institute of Technology
Atlanta, Georgia
sudha@gatech.edu

***Abstract*—This paper explores the potential benefits to asynchronous task-based execution to achieve high performance under a power cap. Task-graph schedulers can flexibly reorder tasks and assign compute resources to data-parallel (elastic) tasks to minimize execution time, compared to executing step-by-step (bulk-synchronously). The efficient utilization of the available cores becomes a challenging task when a power cap is imposed. This work characterizes the trade-offs between power and performance as a Pareto frontier, identifying the set of configurations that achieve the best performance for a given amount of power. We present a set of scheduling heuristics that leverage this information dynamically during execution to ensure that the processing cores are used efficiently when running under a power cap. This work examines the behavior of three HPC applications on a 57 core Intel Xeon Phi device, demonstrating a significant performance increase over the baseline.**

I. Introduction

The scientific and enterprise computing domains are seeing the rise of extreme scale systems to handle the explosive growth in problem sizes [1]. Typically, these systems are constructed from high core count data parallel processors such as graphics processing units (GPUs) as well as general purpose data parallel processors like the Xeon Phi. The bulk synchronous parallel (BSP) model has been the dominant programming and execution time abstraction employed for programming each processor [2]. In spite of significant successes, modern applications with irregular and unstructured control and data flows exhibit significant inefficiencies in the BSP model inspiring the emergence of the asynchronous task graph model for programming extreme scale applications to achieve new performance peaks [3], [4]. However, this performance must be achieved subject to increasingly stringent power caps, as these systems will need to be designed to fit within the tightening constraints of power and scalability.

The runtime systems used to execute the asynchronous task graphs have so far ignored the imposed power caps, focusing solely on performance. While existing works [5], [6] have examined the problem of scheduling tasks with limited compute resources, power constraints create a new challenge for performance optimization in the High Performance Computing (HPC) domain. Ensuring that the execution of these task graphs is power-efficient will take on greater importance as system scales increase.

This work presents a set of scheduling strategies for task graphs that improve performance when running under a power cap. In this work we compare a static core-limiting system, analogous to the way systems are provisioned for the worst-case power behavior, with dynamic, power-aware schedulers that can flexibly decide when resources can be used so long as the power cap is not exceeded. We propose the offline construction of Pareto-optimal power–performance profiles of data-parallel tasks to guide the dynamic schedulers, providing a set of trade-offs between power consumption and the number of cores. We also examine how leveraging program slack, an existing technique from the BSP model [7], performs in asynchronous models, showing how it is *ineffective* at reducing average power.

The following sections describe the methodology for the construction of Pareto frontiers followed by the formulation of the scheduling heuristics. We evaluate their performance on a set of real HPC applications. Our approach demonstrates that the power-aware schedulers can perform on average 18.8%, 15.4%, and 38.4% better than the statically constrained schedulers when limited by the power cap for the CG, LU, and Cholesky applications and perform as well as the state-of-the-art when power is no longer restrictive. This work shows the initial examination of task graph application performance under a power cap and discusses challenges and opportunities for refinement.

II. Related Work

The problem of effectively scheduling task graphs has been shown to be difficult, particularly when additional constraints, such as limited power or number of cores, need to be considered. The work by Buttari et al. [8] describes the formulation of linear algebra algorithms including Cholesky and LU decompositions into tasks and describing their performance improvements over more traditional algorithms. The DAGuE runtime [3] provides a programming model and framework for the execution of task graphs for the HPC domain, focusing on scale and speed of execution for large problem sizes, and has been used to implement the DPLASMA library of

E2SC2016; Salt Lake City, Utah, USA; November 2016
978-1-5090-3856-5/16/$31.00 ©2016 IEEE

dense linear algebra functions for distributed systems [9]. This formulation of asynchronous DAGs has been expanded by Wu [10] to add an additional layer of hierarchy, distinguishing between the coarse-grained task parallelism and the fine-grained data parallelism within a task - which has been term elastic or parallel tasks [11], [12]. To address the scheduling of graphs when constrained by the number of cores available, Barbosa el al. [5] present a static method and Vydyanathan et al. [6] present a dynamic method for assigning cores to tasks to minimize execution time. These works use analytical performance models based on the task workload. In contrast, our paper focuses on the empirical behavior of tasks to classify their execution.

Tackling the problem of power limits for HPC systems has presented many techniques for increased energy efficiency. One common technique is to leverage *slack* to slow down cores that do not affect the execution time of the overall application, hence saving energy [13]. This has been expanded to distributed HPC workloads [14]. Understanding when imbalance occurs in an application in order to reduce energy when executing some applications under a power cap has been demonstrated [7]. These works apply to bulk-synchronous programming models, whereas the work proposed by this paper applies to the more complex problem of scheduling asynchronous task graphs.

Zhang [15] provides a survey of methods for enforcing a power limit, both by the hardware and software. However, the surveyed techniques are viewed from the perspective of administration, ensuring that limits are met within a timely manner. Methods for increasing energy efficiency by adjusting the configuration either of the number of threads given to each task [16] or the configuration of the hardware, tweaking either the frequency states or number of available cores for use [17] have been demonstrated. Additionally, work by Patki et al. [18] shows the effect of power-aware job scheduling depending on user behavior. While not focused on executing task graphs, that work provides a foundation for applying power limitations to entire clusters, ensuring fast task turnaround time based upon user requirements.

Research has been shown to estimate the performance, power, or energy behavior of application tasks as a function of frequency state [19]. Work such as that proposed by Anger et al. [20] presents a technique to model the time and energy behaviors of tasks as a function of program input, shown to provide low overhead. The models in this paper leverage Pareto frontiers to describe the power and performance for each of the tasks. Pareto-optimal configurations have been used to characterize system designs across multiple cores [21]. The work in this paper differs by examining the task-specific power–performance tradeoffs for an individual system.

III. Scheduling Task Graphs

This work examines applications built using the asynchronous task-graph programming model, where individual operations, called *tasks*, form the nodes in a directed acyclic graph (DAG) with the edges formally specifying the data

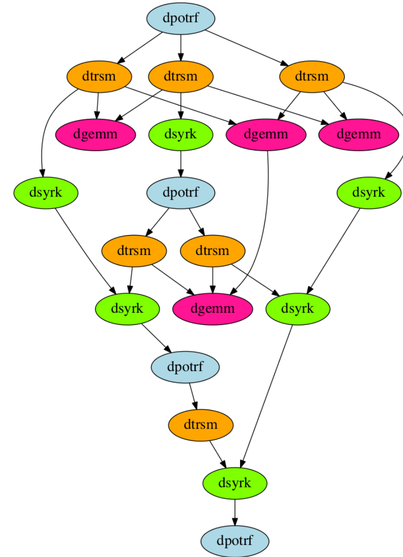

Fig. 1: Task graph showing task dependencies for a 4x4 tile-base right-looking Cholesky decomposition.

dependencies between tasks. Figure 1 shows the task graph for Cholesky decomposition; a task on the graph is only allowed to execute once all of its predecessors have completed. A *runtime* monitors the progress of tasks as well as the utilization of system resources, such as the number of cores already assigned to tasks.

The design of the runtime must take into consideration the complicating issue of hierarchical levels of parallelism provided by the task graph. There is coarse-grained parallelism for tasks that may execute concurrently provided there is no data-dependence between them. Tasks may have fine-grained data parallelism within a task so that a node within the DAG may be executed on more than one processor core. Such tasks have been termed *elastic* [11]. This trade-off between degrees of parallelism makes the scheduling decision to be made by the runtime more complicated, as it may chose between running multiple tasks or a single task on multiple processor cores.

Making a scheduling decisions depends on the relationship between task configuration—in this case, number of processor cores—and the resulting performance. Existing theoretical models, such as Amdahl's Law, demonstrate a monotonically increasing relationship between number of cores and performance. However this may not always be the case due to the effects such as communication overhead. As a result, a parallel task may have higher performance with a smaller number of processors than larger. The number of cores a task uses may also change the amount of power consumed. That is, each task can be characterized by a set of points corresponding to the execution time and power consumed by the task for a given configuration. Figure 2 shows this for a single task; each point represents running the task with a specific number of processor cores.

From these figures it is possible to find configurations that achieve the same performance for different amounts of power,

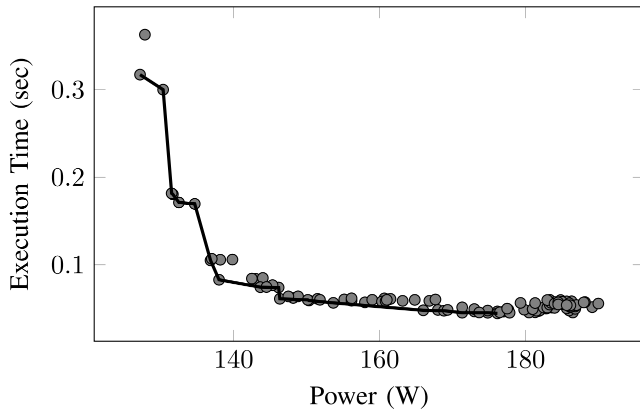

Fig. 2: Task configuration points and resulting Pareto frontier for the `DPOTRF` task from Cholesky factorization.

or conversely different performance for the same amount of power. The set of configurations that achieve the highest performance for a given power consumption are called *Pareto-optimal* and lie on the *Pareto frontier*. In other words, a configuration is Pareto-optimal if it is not strictly dominated by any other. The line in Figure 2 is drawn through the Pareto frontier. By selecting a configuration on the Pareto frontier, the scheduler knows that it is choosing the best performing configuration for the given amount of power. This work proposes the offline construction of these frontiers for use during task scheduling, allowing the selection of the most power-efficient configurations.

IV. Power-Efficient Scheduling

During execution, the runtime decides which tasks to run as well as the number of processor cores they may use. The system is provisioned with a certain number of cores that may be used globally across all tasks at any moment in time; the scheduler makes the decision as to how these cores get assigned, as well as which tasks to assign them to. As soon as a task is capable to execute, a call to the scheduling heuristic decides which tasks in the set of ready tasks to run, as well as the configuration to use for each of these tasks. This work presents four different scheduling heuristics to demonstrate the ability to increase performance while operating under a power cap, with each heuristic providing a successively more comprehensive approach to minimize execution time of the task graphs while remaining below the power limit, including how leveraging the Pareto frontiers, in addition to a power-aware heuristic, results in high performance.

A. Machine Model

This work examines the issue of scheduling under a power cap from the perspective of a single, homogeneous system, but may be adapted to distributed or heterogeneous systems. The task graphs used in this work exhibit an extremely high degree of parallelism both coarse-grained (task-level) and fine-grained (data-level). Due to this degree of expressed parallelism, this work leverages an Intel Xeon Phi coprocessor [22] to provide a homogeneous computing system with a large core count. The Intel Xeon Phi architecture allows for up to 72 in-order x86 cores connected by a ring interconnect. Each core can support up to four hardware threads. The instruction scheduler issues from each hardware thread in a round-robin fashion, but with the limitation that it is unable to issue from the same thread back-to-back. As a simplification, this work assumes each core is fungible. Rather than modeling the interactions between threads when they must compete for resources on a core, we limit each core to two simultaneous threads, which can each provide a continuous instruction stream to the core.

Each core has its own L1 and L2 cache, with device-local memory and a large vector processing unit. Importantly, this architecture retains a traditional programming model similar to existing CPU workloads, allowing for compiler and runtime control over the degree of parallelism per task and across tasks. As such, existing workloads need little modification in order to run on these devices.

V. Scheduling Heuristics

This work examines four different scheduling heuristics, described below. The input to each heuristic is the set of tasks that are ready to run, as well as the set of available resources, which includes the amount of power left in the budget, as well as the current number of unassigned processor cores.

A. Fair Scheduler (FS)

Algorithm 1 Fair Scheduler

1: $numCores$ = getNumIdleCores()
2: $numTasks$ = size($pendingTasks$)
3: $coresPerTask = numCores/numTasks$
4: Assign $coresPerTask$ to each Task in $pendingTasks$

The Fair Scheduler (Algorithm 1) is the most simple heuristic which assigns cores in a round-robin manner to all of the ready-to-execute tasks. For this heuristic, the goal is to maximize utilization of the cores with no regard given to the power consumed during execution. Rather, the power utilization will be enforced statically by selecting the highest number of total cores that results in a peak power consumption that does not exceed the power cap. This is analogous to provisioning the scheduler for the worst case, preventing any cores from being used if they will, at any time, go over the power cap. This scheduler forms the baseline for this work.

B. Core Constrained Scheduler (CCS)

The second scheduler is called the Core Constrained Scheduler (2). This heuristic, adapted from [5], attempts to minimize execution time of the task graph by assigning cores to the tasks that lie on the critical path. To achieve this, an estimate must be made of the total time to complete a specific task plus the time to complete the longest chain of tasks to the bottom of the graph, called the *makespan*. To calculate the makespan, the scheduler assumes that all subsequent tasks

Algorithm 2 Core Constrained Scheduler

1: Set each task to highest performance configuration
2: **while** $\sum_{i=0}^{numTasks} cores_i > numIdleCores$ **do**
3: **for each** Pending task T **do**
4: Determine makespan with $cores_T = cores_T - 1$
5: **if** Makespan is largest seen yet **then**
6: Set T as losing task
7: **end if**
8: **end for**
9: Reduce threads for losing task
10: **end while**

after the current one to be scheduled will use the highest performance configuration possible.

As with the Fair Scheduler, this scheduler does not consider power consumption to make its scheduling decisions. Instead, a similar static limitation will be placed on the number of idle cores to ensure that at no time it goes over the power cap. This represents the current state of the art in achieving high performance, but it is not cognizant of the power cap. The goal is to examine how well these heuristics manage when power limitations are put into place.

C. Power Aware Scheduler (PAS)

Algorithm 3 Power Aware Scheduler

1: Set each task to highest performance Pareto point
2: **while** $\sum cores_i > numIdleCores$ or $\sum power(cores_i) + currentPower > powerCap$ **do**
3: **for each** Pending task T **do**
4: Move to next less powerful Pareto-optimal point
5: Determine the new makespan
6: **if** Makespan is shortest seen yet **then**
7: Set T as losing task
8: **end if**
9: **end for**
10: Walk down Pareto frontier for losing task
11: **end while**

The Power Aware Scheduler (3) is the first scheduler that leverages the Pareto frontiers to model the power behaviors of the executing tasks, providing a mechanism for dynamically tuning the power behavior of the system. Instead of statically limiting the number of idle cores available to the scheduler, this heuristic is aware of the power cap, as well as the amount of power used by each task. The insight is that, informed by the Pareto frontiers, this heuristic is able to exceed the number of cores in the prior two schedulers so long as it guarantees that it remains under the power cap. In this way, it is able to accelerate some tasks by giving them more cores, reducing the makespan of the graph, so long as there is power available.

The key insight for this scheduler is that power can be attributed dynamically, allowing tasks to leverage cores so long as, for the current scheduling tick, the increase in power on top of the current power draw does not exceed the power cap. In practice, this results in a higher average core utilization than the static schedulers, as the scheduler can assign more cores to tasks than the static limit, so long as there is power headroom.

TABLE I: Machine configuration used for experimental evaluation.

Component	Parameter	Value
host	CPU	Intel i7-975
host	CPU cores	4
host	CPU clock rate	3.33 GHz
host	Memory size	6GB
coprocessor	CPU	Intel Xeon Phi 3120A
coprocessor	CPU clock rate	1.10 GHz
coprocessor	CPU cores	57
coprocessor	Memory size	6GB

D. Slack Aware Scheduler (SAS)

Algorithm 4 Slack Aware Scheduler

1: Call PAS
2: **for each** Pending task T **do**
3: **if** T is not on critical path **then**
4: **while** Slack $\neq 0$ **do**
5: Move to next less powerful Pareto-optimal point
6: **end while**
7: **end if**
8: **end for**

The Slack Aware Scheduler (4) is an improvement over PAS. The key insight for this heuristic is that a task may be slowed down so long as it does not affect the critical path of the graph. The additional amount of time a task can run before affecting the schedule is called *slack* [13]. This scheduler operates like the PAS, but with the additional step that allows tasks to move into lower power configurations so long as there is slack. The scheduler determines if there is slack by slowing down the thread and recalculating the graph's makespan; there is slack if the makespan does not change. The goal for this scheduler is to achieve the same performance as the PAS, but with a lower average power. This builds off of existing techniques for achieving energy efficiency from slack [7], but expands their use to asynchronous task graphs.

VI. Performance Evaluation

This section describes the experimental evaluation of the Pareto frontiers, as well as the behavior of the four presented schedulers. It describes the execution environment upon which the different heuristics were run, as well as the structure of each task's Pareto frontiers used as the power and performance models to guide the power-aware scheduling decisions.

A. Experiment Setup

Table I shows the hardware configuration for the host and coprocessor card. To enable launching tasks asynchronously

on the device, this work leverages the Intel Compiler's Heterogeneous Offload programming model. The main scheduling thread executes on the host, launching tasks to the device and checking on the execution progress. When a task needs to be launched, a new worker thread on the device is spun up, which then uses the `offload` compiler pragma to issue task execution to the coprocessor. The main scheduler then waits for this worker thread to complete. The offload framework takes an entire coprocessor to choreograph offload requests, so the total number of hardware threads a scheduler can assign tasks to at any time is 112.

This work leverages the `libmicmgmt` library provided by Intel which accesses Xeon Phi parameters [23] to poll the onboard power sensors. These measure the current and voltage for the device at a rate of $50Hz$. To collect the profiling data to construct the Pareto frontiers, each task graph was executed by a profiling scheduler that would repeatedly launch a task for a set period and read the power of the coprocessor, recording the highest observed power and the average task execution time. This power number includes the idle power, measured as 106.9 Watts, for the coprocessor, which must be subtracted in order to determine the incremental power cost for launching a task.

For the validation experiments, the applications are executed by each scheduler and the execution times compared to the Fair Scheduler. Due to the coarse power measurement granularity, this work uses the values from the Pareto frontiers to represent the task power behavior. As such, this work does not examine the enforcement of the power limit during experimental execution.

B. Applications

The schedulers in this work are fed by a DAG of tasks. Instead of constructing synthetic tasks this work uses three different task graphs taken from current HPC workloads. Each application can be parameterized to the size of the input problem as well as the degree of tiling. The tasks from these applications are linear algebra kernels provided by the Intel MKL library [24]. Each task can be configured to use a different number of OpenMP threads to control their degree of data-level parallelism.

Conjugate Gradient (CG) The conjugate gradient example is adapted from the HPCG benchmark for solving a matrix equation of the form $Ax = b$ for a given sparse matrix A and vector b for unknown x. The task-based version is derived from the HPCG benchmark [25], using a compressed sparse row (CSR) representation of A. The sparse structure of the matrix A is derived from a structured 3D stencil with X-Y-Z linearized to form the column and row indices. We assume a +1/-1 connectivity in each dimension, resulting in a a total of 27 nonzeroes for each row (fewer for boundaries). The current example does not use a preconditioner. The individual kernels required for conjugate gradient are `DDOT` and `DAXPY` and a sparse matrix-vector multiply (`SpMV`).

Cholesky Decomposition The Cholesky decomposition calculates the decomposition of a positive-definite matrix A into the form $A = LL^*$ where the matrix L is a real lower-triangular matrix with positive elements in the diagonal. We use a tiled-based right-looking algorithm, as described in [26]. Cholesky is done in double precision, requiring the BLAS kernels `DPOTRF`, `DSYRK`, `DTRSM`, and `DGEMM`. The task graph for a 4×4 tile-based Cholesky is shown in Figure 1.

LU Decomposition The Lower-Upper (LU) decomposition factorizes a matrix A into the form $A = LU$ where L is a lower triangular matrix and U is an upper triangular matrix. As with Cholesky, the LU Decomposition uses a tiled, right-looking algorithm [26]. The tasks for this application are `DTRSM`, `DGETRF`, and `DGEMM`.

C. Results

Figure 3 shows the different Pareto frontiers for each task in the three task graphs examined in this work. The curves pass through the Pareto-optimal points that express the highest relative speedup over single-threaded execution versus power consumed over idle. Some tasks have a strong correlation between power and speedup, such as `SpMV` in CG and `DGEMM` in Cholesky and LU, as seen by the relatively straight shapes of the Pareto frontiers. On the other hand, tasks such as `DAXPY` in CG and `DPOTRF` in Cholesky have few Pareto-optimal points, which indicates poor performance scaling.

Figure 4 shows the performance of each scheduler relative to the performance of the Fair Scheduler, as a function of an imposed power cap. The power cap starts at 35 Watts, which is guaranteed to contain at least one Pareto-optimal point for each task. The CCS performs similarly to the baseline. For a fixed number of cores, CCS has a higher performance, but also a higher power consumption. When the power cap is imposed, the CCS must use fewer total cores than the FS, even though it uses them more efficiently.

However, the two power-aware schedulers, due to their dynamic nature, are more power-efficient than the baseline and the CCS. For Cholesky and LU, the PAS and SAS schedulers perform better than FS and CCS across all power caps. This becomes less pronounced as the power cap increases for CG in particular; at this level of power cap, the FS and CCS are not statically restricted by the number of cores they can use. That is, they can use all of the cores available and still remain underneath the cap. Compared to Cholesky and LU, the tasks in CG scale poorly (except for `SpMV`), so that even though there is more headroom under the power cap, the tasks would not exhibit a speedup.

An important observation is of the parabolic shape to the performance improvement of PAS and SAS over both the baseline and CCS, seen most clearly with LU: when the power cap is low, none of the schedulers perform well since very few cores can be used. There is a middle region where the dynamic, power-aware schedulers achieve the greatest speedup attributed to their higher average core utilization. Then finally, the power cap gets high enough to allow all cores to be used, even for the static schedulers, so performance normalizes. This underscores the need to understand what power caps ought to be feasibly set at from the perspective of applications, possibly

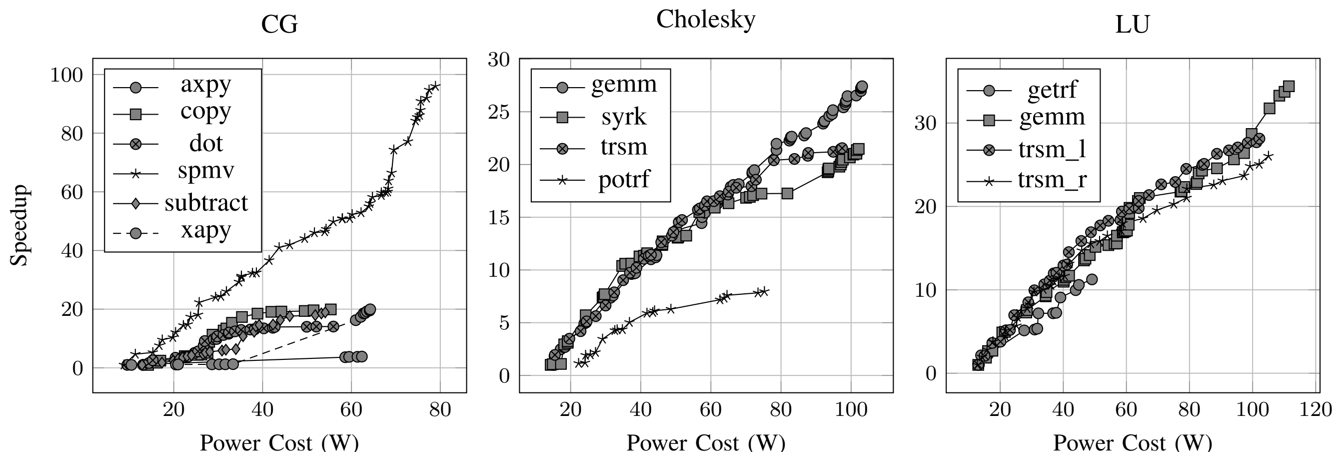

Fig. 3: Pareto frontiers for each task in CG, Cholesky, and LU task graphs.

across phases. Certain power cap–application relationships can elicit the greatest improvements in efficiency and others elicit little. Characterizing this in a general way remains a problem to be addressed.

The slack-aware scheduler performs comparably to the power-aware scheduler. However, SAS has an average power of only 0.43% less than PAS for Cholesky and has an increase in power of 0.15% and 1.2% for CG and LU, respectively. This goes against the hypothesis that taking advantage of slack can lower average power while retaining performance. The reason for this behavior can be attributed to the difference between the asynchronous programming model and the bulk-synchronous model (that can leverage slack). Tasks that slow themselves down free up cores for use in subsequent scheduling ticks. The schedulers only look at the current set of tasks to be scheduled. This may not include the tasks that lie on the critical path of the overall graph; still, the scheduler minimizes the distance between the current tasks to be scheduled and the end of the graph, using up the power headroom as necessary. In the end, the limiting task is run at a higher power, but does not affect the execution time of the graph. This results in a similar or higher average power. A more intelligent scheduler that can determine the true execution-limiting tasks may be better informed to leverage any slack during scheduling.

For this work, the Pareto frontiers are constructed from profiling runs, using the average execution time and peak power. However, tasks such as `SpMV` are dependent upon the input data. As well, the interactions between the tasks is not modeled, such as the behavior of scheduling more than two hardware threads per core. While this work does not attempt to model these behaviors, they would provide greater accuracy for the schedulers, potentially increasing the power-efficiency and performance of the graphs.

VII. Conclusion

This work presents a methodology for scheduling asynchronous task graphs while remaining under an imposed power cap. Key to this methodology is the use of Pareto frontiers to represent the power–performance configurations to ensure the greatest task performance for a given amount of power. In this work, the configuration space is the number of cores assigned to a data-parallel task. This work shows four scheduling heuristics used by the runtime to determine the configuration state for each task: a Fair Scheduler that assigns cores in a round-robin fashion, a Core-Constrained Scheduler that attempts to minimize the distance from the current tasks to the end of the graph, a Power-Aware Scheduler that leverages the Pareto frontiers to pick the tasks to best utilize the available power budget, and a Slack-Aware Scheduler that attempts to slow down those tasks that are not on the critical path, reducing average power consumption. An experimental validation of Conjugate Gradient, Cholesky Decomposition, and LU Decomposition shows how the dynamic, power-aware schedulers are able to increase performance while remaining underneath a power cap.

References

[1] S. Ashby, P. Beckman, J. Chen, P. Colella, B. Collins, D. Crawford, J. Dongarra, D. Kothe, R. Lusk, P. Messina, T. Mezzacappa, P. Moin, M. Norman, R. Rosner, V. Sarkar, A. Siegel, F. Streitz, A. White, and M. Wright, "The Opportunities and Challenges of Exascale Computing—Summary Report of the Advanced Scientific Computing Advisory Committee (ASCAC) Subcommittee," *US Department of Energy Office of Science*, Fall 2010.

[2] A. V. Gerbessiotis and L. G. Valiant, "Direct Bulk-Synchronous Parallel Algorithms," *Journal of Parallel and Distributed Computing*, vol. 22, no. 2, pp. 251–267, Aug. 1994.

[3] G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, P. Lemarinier, and J. Dongarra, "DAGuE: A Generic Distributed DAG Engine for High Performance Computing," in *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops (IPDPSW)*, May 2011, pp. 1151–1158.

[4] N. Fauzia, V. Elango, M. Ravishankar, J. Ramanujam, F. Rastello, A. Rountev, L. Pouchet, and P. Sadayappan, "Beyond reuse distance analysis: Dynamic analysis for characterization of data locality potential," *ACM Trans. Archit. Code Optim.*, vol. 10, pp. 1–29, 2013.

[5] J. Barbosa, C. Morais, R. Nobrega, and A. P. Monteiro, "Static scheduling of dependent parallel tasks on heterogeneous clusters," in *Cluster Computing, 2005. IEEE International*, Sep. 2005, pp. 1–8.

[6] N. Vydyanathan, S. Krishnamoorthy, G. Sabin, U. Catalyurek, T. Kurc, P. Sadayappan, and J. Saltz, "An Integrated Approach to Locality-Conscious Processor Allocation and Scheduling of Mixed-Parallel Ap-

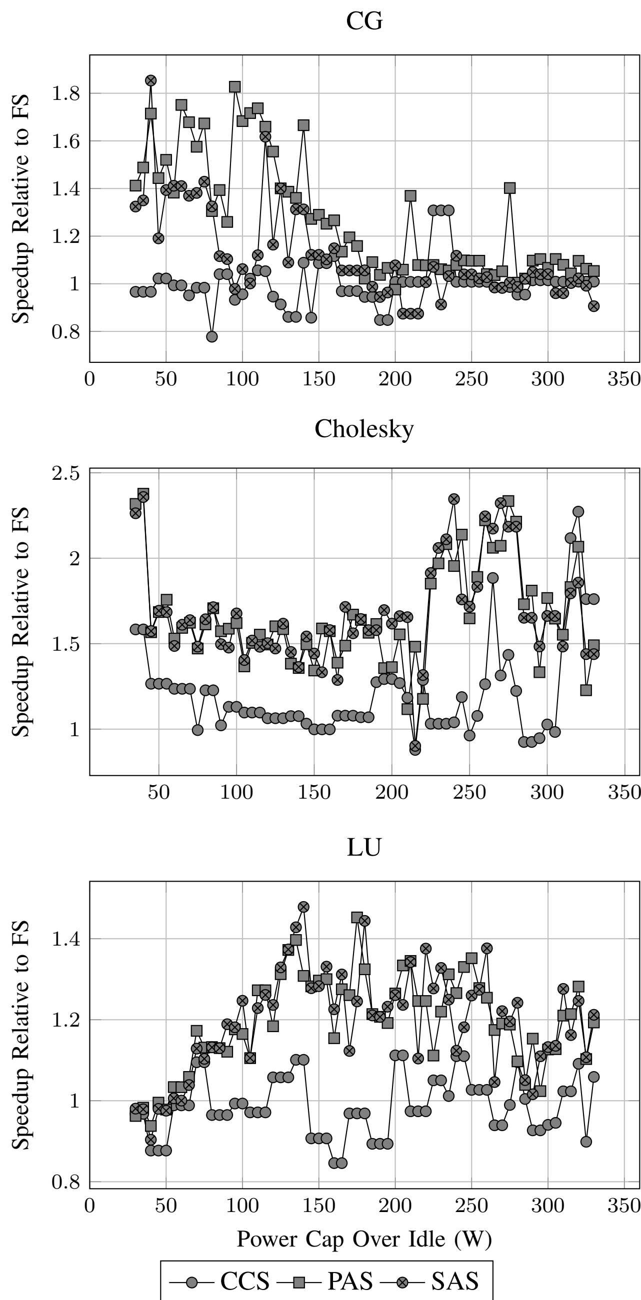

Fig. 4: Relative performance of the CCS, PAS, and SAS schedulers over the Fair Scheduler for the three task graphs as a function of the power cap.

plications," *IEEE Transactions on Parallel and Distributed Systems*, vol. 20, no. 8, pp. 1158–1172, Aug. 2009.

[7] K. J. Barker, D. J. Kerbyson, and E. Anger, "On the Feasibility of Dynamic Power Steering," in *Energy Efficient Supercomputing Workshop (E2SC), 2014*, Nov. 2014, pp. 60–69.

[8] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra, "A class of parallel tiled linear algebra algorithms for multicore architectures," *Parallel Computing*, vol. 35, no. 1, pp. 38–53, Jan. 2009.

[9] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, A. Haidar, T. Herault, J. Kurzak, J. Langou, P. Lemarinier, H. Ltaief, P. Luszczek, A. YarKhan, and J. Dongarra, "Flexible Development of Dense Linear Algebra Algorithms on Massively Parallel Architectures with DPLASMA," in *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops (IPDPSW)*, May 2011, pp. 1432–1441.

[10] W. Wu, A. Bouteiller, G. Bosilca, M. Faverge, and J. Dongarra, "Hierarchical DAG Scheduling for Hybrid Distributed Systems," in *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*, May 2015, pp. 156–165.

[11] A. Sbirlea, K. Agrawal, and V. Sarkar, "Elastic Tasks: Unifying Task Parallelism and SPMD Parallelism with an Adaptive Runtime," in *Euro-Par 2015: Intl. Conference on Parallel and Distrib. Computing*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015.

[12] N. Vydyanathan, S. Krishnamoorthy, G. M. Sabin, U. V. Catalyurek, T. Kurc, P. Sadayappan, and J. H. Saltz, "An Integrated Approach to Locality-Conscious Processor Allocation and Scheduling of Mixed-Parallel Applications," *IEEE Transactions on Parallel Distrib. Syst.*, vol. 20, pp. 1158–1172, 2009.

[13] N. Kappiah, V. W. Freeh, and D. Lowenthal, "Just In Time Dynamic Voltage Scaling: Exploiting Inter-Node Slack to Save Energy in MPI Programs," in *Supercomputing, 2005. Proceedings of the ACM/IEEE SC 2005 Conference*, 2005.

[14] D. Li, B. de Supinski, M. Schulz, K. Cameron, and D. Nikolopoulos, "Hybrid MPI/OpenMP power-aware computing," in *2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, Apr. 2010, pp. 1–12.

[15] H. Zhang and H. Hoffmann, "Maximizing Performance Under a Power Cap: A Comparison of Hardware, Software, and Hybrid Techniques," in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '16. New York, NY, USA: ACM, 2016, pp. 545–559.

[16] S. Sridharan, G. Gupta, and G. S. Sohi, "Holistic Run-time Parallelism Management for Time and Energy Efficiency," in *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ser. ICS '13. New York, NY, USA: ACM, 2013, pp. 337–348.

[17] P. Bailey, D. Lowenthal, V. Ravi, B. Rountree, M. Schulz, and B. de Supinski, "Adaptive Configuration Selection for Power-Constrained Heterogeneous Systems," in *2014 43rd International Conference on Parallel Processing (ICPP)*, Sep. 2014, pp. 371–380.

[18] T. Patki, D. K. Lowenthal, A. Sasidharan, M. Maiterth, B. L. Rountree, M. Schulz, and B. R. de Supinski, "Practical Resource Management in Power-Constrained, High Performance Computing," in *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '15. New York, NY, USA: ACM, 2015, pp. 121–132.

[19] M. Curtis-Maury, A. Shah, F. Blagojevic, D. S. Nikolopoulos, B. R. de Supinski, and M. Schulz, "Prediction Models for Multi-dimensional Power-performance Optimization on Many Cores," in *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '08. New York, NY, USA: ACM, 2008, pp. 250–259.

[20] E. Anger, D. Dechev, G. Hendry, J. Wilke, and S. Yalamanchili, "Application Modeling for Scalable Simulation of Massively Parallel Systems," in *2015 IEEE International Conference on High Performance Computing and Communications (HPCC)*, Aug. 2015.

[21] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger, "Dark Silicon and the End of Multicore Scaling," in *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ser. ISCA '11. New York, NY, USA: ACM, 2011, pp. 365–376.

[22] R. Rahman, *Intel Xeon Phi Coprocessor Architecture and Tools: The Guide for Application Developers*, 1st ed. Berkely, CA, USA: Apress, 2013.

[23] *Intel Xeon Phi Coprocessor System Software Developers Guide*, Mar. 2014, http://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/xeon-phi-coprocessor-system-software-developers-guide.pdf.

[24] *Intel Math Kernel Library Developer Reference*, Aug. 2015, https://software.intel.com/en-us/articles/mkl-reference-manual.

[25] J. Dongarra and M. A. Heroux, "Toward a new metric for ranking high performance computing systems," in *Toward a New Metric for Ranking High Performance Computing Systems*, 2013.

[26] A. Haidar, H. Ltaief, A. YarKhan, and J. Dongarra, "Analysis of dynamically scheduled tile algorithms for dense linear algebra on multicore architectures," *Concurr. Comput. : Pract. Exp.*, vol. 24, pp. 305–321, 2011.