TX: Algorithmic Energy Saving for Distributed Dense Matrix Factorizations

Li Tan and Zizhong Chen
University of California, Riverside
{ltan003, chen}@cs.ucr.edu

***Abstract*—The pressing demands of improving energy efficiency for high performance scientific computing have motivated a large body of solutions using Dynamic Voltage and Frequency Scaling (DVFS) that strategically switch processors to low-power states, if the peak processor performance is unnecessary. Although OS level solutions have demonstrated the effectiveness of saving energy in a black-box fashion, for applications with variable execution patterns, the optimal energy efficiency can be blundered away due to defective prediction mechanism and untapped load imbalance. In this paper, we propose TX, a library level *race-to-halt* DVFS scheduling approach that analyzes Task Dependency Set of each task in distributed Cholesky/LU/QR factorizations to achieve substantial energy savings OS level solutions cannot fulfill. Partially giving up the generality of OS level solutions per requiring library level source modification, TX leverages algorithmic characteristics of the applications to gain greater energy savings. Experimental results on two clusters indicate that TX can save up to 17.8% more energy than state-of-the-art OS level solutions with negligible 3.5% on average performance loss.**

I. Introduction

A. Motivation

With the growing prevalence of distributed-memory architectures, high performance scientific computing has been widely employed on supercomputers around the world ranked by the TOP500 list [1]. Considering a crucial fact that the costs of powering a supercomputer are rapidly increasing nowadays due to expansion of its size and duration in use, improving energy efficiency for high performance scientific applications has been regarded as a pressing issue to solve. The Green500 list [2], ranks the top 500 supercomputers worldwide by performance-power ratio in six-month cycles. Root causes of high energy consumption while achieving performance efficiency have been widely studied. With different focuses of studies, holistic hardware and software approaches for reducing energy costs of running high performance scientific applications have been extensively proposed. Software-controlled hardware solutions such as DVFS-directed (Dynamic Voltage and Frequency Scaling) energy efficient scheduling are deemed to be effective and lightweight [3] [4] [5] [6] [7] [8] [9]. Performance and memory constraints have been considered as trade-offs for energy savings [10] [11] [12] [13] [14].

DVFS is a runtime technique that is able to switch operating voltage and working frequency of a hardware component (CPU, GPU, memory, etc.) to different *scales* (also known as *gears* [4]) per workload characteristics of applications to gain energy savings dynamically. CPU and GPU are the most widely applied hardware components for energy efficiency via DVFS due to two reasons: (a) Compared to other components such as memory, CPU/GPU DVFS is easier to implement [15], and various handy DVFS APIs have been industrialized for CPU/GPU DVFS such as CPUFreq kernel infrastructure [16] incorporated into the Linux kernel and NVIDIA System Management Interface (nvidia-smi) [17] for NVIDIA GPUs; (b) CPU energy costs dominate the total system energy consumption [18] (CPU and GPU energy costs dominate if heterogeneous architectures are considered), and thus saving CPU and GPU energy greatly improves energy efficiency of the whole system. In this work, we focus on distributed-memory systems without GPU. Energy saving opportunities can be exploited by reducing CPU frequency and voltage for non-CPU-bound operations such as large-message MPI communication, since generally execution time of such operations barely increases at a low-power state of CPU. Given the fact that energy consumption equals product of average power consumption and execution time, i.e., $E = \overline{P} \times T$, and the assumption that dynamic power consumption by a CMOS-based processor is proportional to product of working frequency and square of supply voltage, i.e., $P \propto fV^2$ [19] [20], energy savings can be effectively achieved using DVFS-directed strategical scheduling approaches with little performance loss.

High performance applications can be scheduled in the unit of task, a set of operations that are functionally executed as a whole. Different tasks within one process or across processes may depend on each other due to *intra-process* and *inter-process* data dependencies. Parallelism of task-parallel applications can be characterized by graph representations like Directed Acyclic Graph (DAG), where data dependencies among parallel tasks are appropriately denoted by directed edges. Effectiveness for analyzing parallelism using DAG is greatly beneficial to achieving energy efficiency for high performance applications. As typical task-parallel algorithms for scientific computing, dense matrix factorizations in numerical linear algebra have been widely adopted to solve systems of linear equations. Empirically, as standard functionality, routines of dense matrix factorizations are provided by various software libraries of numerical linear algebra for distributed-memory multicore architectures, such as ScaLAPACK [21] and DPLASMA [22]. Therefore, saving energy for distributed dense matrix factorizations contributes significantly to the greenness of high performance scientific computing nowadays.

B. Limitations of Existing Solutions

Most existing energy saving solutions for high performance applications are (combination of) variants of two classic approaches: (a) A Scheduled Communication (*SC*) approach [3] [23] [8] that keeps low CPU performance during communication and high CPU performance during computation, as large-message communication is not CPU-bound while computation is, and (b) a Critical Path (*CP*) approach [8] [9] [24] that guarantees that tasks on the CP run at the highest CPU frequency while reduces frequency *appropriately* (i.e., without further delay to incur performance loss) for tasks off the CP to minimize slack. Per the operating layer, existing solutions can be categorized into two types: OS level and application level. In general, OS level solutions feature two properties: (a) Working aside running applications and thus requiring no application-specific knowledge and source modification, and (b) making online energy efficient scheduling decisions via dynamic monitoring and analysis. However, application level solutions statically utilize application-specific knowledge to perform specialized scheduling for saving energy, generally with source modification and recompilation (i.e., *generality*) trade-offs. Although with high generality, OS level solutions may suffer from critical disadvantages below, and consequently are far from a sound and complete solution, for applications such as distributed dense matrix factorizations in particular:

EFFECTIVENESS. Although intended to be effective for general applications, OS level approaches rely greatly on underlying workload prediction mechanism, due to lack of knowledge of application characteristics. A prediction algorithm can work well for a specific type of applications sharing similar characteristics, but can be error-prone for other applications, in particular, applications with variable (or even random) execution patterns where the prediction mechanism performs poorly. Algorithms presented in [23] [8] [9] predict execution characteristics of upcoming intervals (i.e., a fixed time slice) according to recent intervals. This prediction mechanism is based on a simple assumption that task behavior is identical every time a task is executed [9]. However, it can be defective for applications with variable execution patterns, like matrix factorizations, where the remaining unfinished matrices become smaller as the factorizations proceed. In other words, length variation of iterations of matrix factorizations can make the prediction inaccurate, which invalidates potential energy savings. Further, the OS level prediction can be costly and thus energy savings are diminished. Given that OS level solutions must predict execution details in the next interval using prior execution information, execution history in some cases may not necessarily be a reliable source for workload prediction, e.g., for applications with fluctuating runtime patterns at the beginning of the execution. As such, it can be time-consuming to obtain an accurate prediction. Since during prediction no energy savings can be fulfilled, considerable potential energy savings can be wasted for a qualified but lengthy prediction.

COMPLETENESS. OS level solutions only work when tasks like computation/communication are running, but energy saving opportunities are untapped during the time otherwise. Empirically, even though load balancing techniques are leveraged, due to data dependencies among tasks and load imbalance that is not completely eliminated, not all tasks in different processes across nodes can start to work and finish at the same time. More energy can be saved for tasks waiting at the beginning of an execution and the last task of one process finishing earlier than that of other processes across nodes. Restricted by the daemon-based nature of working aside real running tasks, OS level solutions cannot attain energy savings for such durations.

C. Our Contributions

In this paper, we propose a library level *race-to-halt* DVFS scheduling approach via Task Dependency Set (TDS) analysis based on algorithmic characteristics, namely TX, to save energy for task-parallel distributed-memory applications, taking distributed dense matrix factorizations for example. The idea of library level *race-to-halt* scheduling is intended for any task-parallel programming models where data flow analysis can be applied. In summary, the contributions are as follows:

- Compared to application level solutions, for widely used software libraries such as numerical linear algebra libraries, TX restricts source modification and recompilation at library level, and replacement of the energy efficient version of the libraries is allowed at link time (i.e., with *partial loss of generality*). No further source modification and recompilation are needed for applications where the libraries are called;
- Compared to OS level solutions, TX is able to achieve substantial energy savings for distributed dense matrix factorizations (i.e., with *higher energy efficiency*), since via algorithmic TDS analysis, TX circumvents the defective prediction mechanism at OS level, and manages to save more energy from the load imbalance;
- With negligible 3.5% on average performance loss, on two clusters, TX is evaluated to achieve up to 33.8% energy savings compared to original runs, and up to 17.8% and 15.9% more energy savings than state-of-the-art OS level *SC* and *CP* approaches, respectively.

The rest content is organized below. Section 2 introduces distributed dense matrix factorizations. We present TDS and CP in section 3, and our TX approach in section 4. Implementation details and experimental results are provided in section 5. Section 6 discusses related work and section 7 concludes.

II. RELATED WORK

Numerous other types of energy efficient DVFS scheduling algorithms exist, but only a few of them were designed for high performance scientific computing. We detail them in the categories *OS-level*, *Application-level*, *Simulation-based*.

OS-LEVEL. There exist a large body of OS level energy saving approaches for high performance scientific applications. Lim *et al.* [23] developed a runtime system that transparently reduces CPU power for communication phases to minimize energy-delay product. Ge *et al.* [11] proposed a runtime system and an integrated performance model for achieving energy efficiency and constraining performance loss through performance modeling and prediction. Rountree *et al.* [8] developed a *SC* approach that employs a linear programming solver collecting communication trace and power characteristics for generating a energy saving scheduling. Subsequent work [9] presented another runtime system by improving and extending previous classic scheduling algorithms and achieved significant energy savings with extremely limited performance loss.

APPLICATION-LEVEL. Kappiah *et al.* [6] introduced a scheduled iteration method that computes the total slack per processor per timestep, then schedules CPU frequency for upcoming timesteps. Liu *et al.* [25] presented a technique that tracks the idle durations for one processor to wait for others to reach the same program point, and utilizes this information to reduce the idle time via DVFS without performance loss. Tan *et al.* [13] proposed an adaptively aggressive scheduling strategy for data intensive applications with moderated performance trade-off using speculation. Subsequent efforts [14] proposed an adaptive memory-aware strategy for distributed matrix multiplication that trades grouped computation/communication with memory costs for less overhead on employing DVFS.

$$
\begin{pmatrix} A_{11} & A_{21}^T & A_{31}^T & A_{41}^T \\ A_{21} & A_{22} & A_{32}^T & A_{42}^T \\ A_{31} & A_{32} & A_{33} & A_{43}^T \\ A_{41} & A_{42} & A_{43} & A_{44} \end{pmatrix} = \begin{pmatrix} L_{11} & 0 & 0 & 0 \\ L_{21} & L_{22} & 0 & 0 \\ L_{31} & L_{32} & L_{33} & 0 \\ L_{41} & L_{42} & L_{43} & L_{44} \end{pmatrix} \times \begin{pmatrix} L_{11}^T & L_{21}^T & L_{31}^T & L_{41}^T \\ 0 & L_{22}^T & L_{32}^T & L_{42}^T \\ 0 & 0 & L_{33}^T & L_{43}^T \\ 0 & 0 & 0 & L_{44}^T \end{pmatrix}
$$

$$
= \begin{pmatrix} L_{11}L_{11}^T & L_{11}L_{21}^T & L_{11}L_{31}^T & L_{11}L_{41}^T \\ L_{21}L_{11}^T & L_{21}L_{21}^T + L_{22}L_{22}^T & L_{21}L_{31}^T + L_{22}L_{32}^T & L_{21}L_{41}^T + L_{22}L_{42}^T \\ L_{31}L_{11}^T & L_{31}L_{21}^T + L_{32}L_{22}^T & L_{31}L_{31}^T + L_{32}L_{32}^T + L_{33}L_{33}^T & L_{31}L_{41}^T + L_{32}L_{42}^T + L_{33}L_{43}^T \\ L_{41}L_{11}^T & L_{41}L_{21}^T + L_{42}L_{22}^T & L_{41}L_{31}^T + L_{42}L_{32}^T + L_{43}L_{33}^T & L_{41}L_{41}^T + L_{42}L_{42}^T + L_{43}L_{43}^T + L_{44}L_{44}^T \end{pmatrix}
$$

Fig. 1. Matrix Representation of a 4 $\times$ 4 Blocked Cholesky Factorization (We henceforth take Cholesky factorization for example due to algorithmic similarity).

SIMULATION-BASED. There exist some efforts on improving energy efficiency for numerical linear algebra algorithms (Cholesky/LU/QR), but most of them are based on simulation or only work for single multicore machine. Few studies have been conducted on energy efficient matrix factorizations running on distributed-memory architectures. Slack reclamation methods such as Slack Reduction and Race-to-Idle algorithms [24] [26] have been proposed to save energy for dense linear algebra operations on shared-memory multicore processors. Instead of running benchmarks on real machines, a power-aware simulator, in charge of runtime scheduling to achieve task level parallelism, was employed to evaluate the proposed power-control policies for linear algebra algorithms. DVFS techniques used in their approaches were also simulated.

III. Distributed Dense Matrix Factorizations

As classic dense numerical linear algebra algorithms for solving systems of linear equations, such as $Ax = b$ where A is a given coefficient matrix and b is a given vector, Cholesky factorization applies to the case that A is a symmetric positive definite matrix, while LU and QR factorizations apply to any general $M \times N$ matrices. The goal of these algorithms is to factorize A into the form LL^T where L is lower triangular and L^T is the transpose of L, the form LU where L is unit lower triangular and U is upper triangular, and the form QR where Q is orthogonal and R is upper triangular, respectively. Thus from $LL^Tx = b$, $LUx = b$, $QRx = b$, x can be easily solved via forward substitution and back substitution. In practice, distributed matrix factorizations are widely employed in extensive areas of high performance scientific computing. Next we introduce an effective graph representation for demonstrating parallelism in runs of distributed dense matrix factorizations.

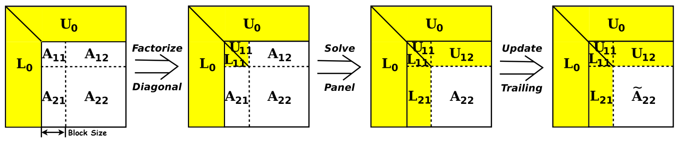

Fig. 2. Stepwise Illustration of LU Factorization without Pivoting.

For performance efficiency, distributed dense matrix factorizations can be implemented as follows: (a) Partition a global matrix into a cluster using load balancing techniques such as 2-D block cyclic data distribution; (b) perform local *diagonal matrix* factorizations individually and communicate factorized matrices among local nodes for *panel matrix* solving and *trailing matrix* updating, as shown in Figure 2, a stepwise LU factorization. As typical task-parallel algorithms where data dependencies frequently arise, parallel runs of distributed Cholesky/LU/QR factorizations can be characterized via Directed Acyclic Graph (DAG), where dependencies among parallel tasks are appropriately represented. DAG for distributed dense matrix factorizations is formally defined below:

DEFINITION 1. Data dependencies among parallel tasks of distributed Cholesky/LU/QR factorizations running on a distributed-memory computing system are modeled by a Directed Acyclic Graph (DAG) $G = (V, E)$, where each node $v \in V$ denotes a task of distributed Cholesky/LU/QR factorizations, and each directed edge $e \in E$ represents a dynamic data dependency from task t_j to task t_i that both tasks manipulate on either different *intra-process* or *inter-process* local matrices (i.e., an *explicit* dependency) or the same *intra-process* local matrix (i.e., an *implicit* dependency), denoted by $t_i \to t_j$.

EXAMPLE. Due to similarity among the three matrix factorizations and space limitation, we henceforth take Cholesky factorization for example. Consider a 4 $\times$ 4 blocked Cholesky factorization in Figure 1. The outcome of the task factorizing A_{11}, i.e., L_{11}, is employed in the tasks solving local matrices L_{21}, L_{31}, and L_{41} in the same column as L_{11}, i.e., the tasks calculating the *panel matrix*. In other words, there exist three data dependencies from the tasks solving L_{21}, L_{31}, and L_{41} to the task factorizing A_{11}, denoted by three *solid* directed edges from the task Factorize(1,1) to the tasks Solve(2,1), Solve(3,1), and Solve(4,1) individually as given in Figure 3.

IV. Task Dependency Set and Critical Path

Next we present Task Dependency Set (TDS) and Critical Path (CP) of application runs, where TDS contains dependency information of parallel tasks at runtime and CP pinpoints potential energy savings in terms of slack among the tasks.

A. Task Dependency Set

For determining the appropriate timing to switch frequency for energy savings, we leverage TDS in our TX approach. Next we first formally define TDS, and then show how to generate two types of TDS for each task in Cholesky factorization using an example. Producing TDS for LU and QR factorizations is similar with minor changes per algorithmic characteristics.

DEFINITION 2. Given a task t of a distributed matrix factorization, data dependencies related to a data block manipulated by the task t are classified as elements of two types of TDS: $\mathsf{TDS}_{in}(t)$ and $\mathsf{TDS}_{out}(t)$, where dependencies from the data block to other tasks t_i are categorized into $\mathsf{TDS}_{out}(t)$ and denoted as t_i, and dependencies from other tasks t_j to the data block are categorized into $\mathsf{TDS}_{in}(t)$ and denoted as t_j.

EXAMPLE. Consider the same Cholesky factorization in Figure 1. Two TDS of each task can be generated statically per algorithmic characteristics of Cholesky factorization: Since the resulting local matrices of factorization tasks (e.g., L_{11}) are employed in column-wise *panel matrix* solving (e.g., solving L_{21}, L_{31}, and L_{41}), data dependencies from *panel matrices* to

Algorithm 1 *DVFS Scheduling Algorithm Using CP*

```
DVFS_CP($CritPath$, $task$, $FreqSet$)
 1: if ($task \in CritPath$ || TDS$_{out}$($task$) != $\varnothing$) then
 2:     SetFreq($f_h$)
 3: else
 4:     $slack \leftarrow$ GetSlack($task$)
 5:     if ($slack > 0$) then
 6:         $f_{opt} \leftarrow$ GetOptFreq($task$, $slack$)
 7:     if ($f_l \leq f_{opt} \leq f_h$) then
 8:         if ($f_{opt} \notin FreqSet$) then
 9:             SetFreq($\lfloor f_{opt} \rfloor$, $\lceil f_{opt} \rceil$, $ratio$)
10:         else SetFreq($f_{opt}$)
11:     else if ($f_{opt} < f_l$) then
12:         SetFreq($f_l$)
13: end if
```

Algorithm 2 *DVFS Scheduling Algorithm Using TX*

```
DVFS_TX($task$, $CurFreq$)
 1: while (TDS$_{in}$($task$) != $\varnothing$) do
 2:     if ($CurFreq$ != $f_l$) then
 3:         SetFreq($f_l$)
 4:     if (Recv($DoneFlag$, $t_1$)) then
 5:         delete(TDS$_{in}$($task$), $t_1$)
 6: end while
 7: SetFreq($f_h$)
 8: if (IsFinished($task$)) then
 9:     foreach $t_2 \in$ TDS$_{out}$($task$) do
10:         Send($DoneFlag$, $t_2$)
11:     SetFreq($f_l$)
12: end if
```

TABLE I. NOTATION IN ALGORITHMS 1 AND 2.

$task, t_1, t_2$	One task of matrix factorizations, out of Factorize, Update1, Update2, and Solve
f_l	The lowest CPU frequency set by DVFS
f_h	The highest CPU frequency set by DVFS
f_{opt}	Optimal ideal freq. to finish a task w/o slack
$ratio$	Ratio between durations of split frequencies
TDS$_{in}$($task$)	Task Dependency Set consisting of tasks that are depended by $task$ as the input
TDS$_{out}$($task$)	Task Dependency Set consisting of tasks that depend on $task$ as the input
$CritPath$	One task trace consisting of tasks to finish matrix factorizations with zero total slack
$slack$	Amount of time that a task can be delayed by without performance loss overall
$CurFreq$	Current CPU frequency in use
$DoneFlag$	Indicator of the finish of a task

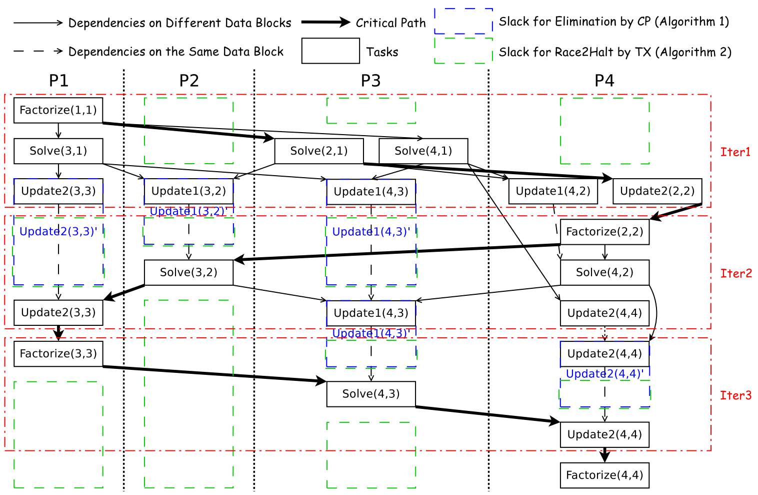

Fig. 3. DAG Representation of Task and Slack Scheduling of CP and TX Approaches for the 4 $\times$ 4 Blocked Cholesky Factorization in Figure 1 on a 2 $\times$ 2 Process Grid Using 2-D Block Cyclic Data Distribution.

factorized *diagonal matrices* are included in TDS$_{in}$ of tasks solving *panel matrices* (e.g., TDS$_{in}$(S(2,1)), TDS$_{in}$(S(3,1)), and TDS$_{in}$(S(4,1))), and TDS$_{out}$ of tasks factorizing *diagonal matrices* (e.g., TDS$_{out}$(F(1,1))). Likewise TDS$_{in}$ and TDS$_{out}$ of other tasks holding different dependencies can be produced.

B. Critical Path

Although load balancing techniques are leveraged for distributing workloads as evenly as possible, assuming that all nodes have the same hardware configuration, slack can still be incurred since different processes can be utilized unfairly due to three reasons: (a) Inbalanced computation delay due to data dependencies among tasks, (b) imbalanced task partitioning, and (c) imbalanced communication delay. Difference in CPU utilization results in different amount of computation slack. For instance, constrained by data dependencies, the start time of processes running on different nodes differs from each other, as shown in Figure 3 where P1 starts earlier than the other three processes. Moreover, since the location of local matrices in the global matrix determines what types of computation are performed locally, load imbalancing from difference in task types and task amount allocated to different processes cannot be eliminated completely by the 2-D block cyclic data distribution, as shown in Figure 3 where P2 has lighter workloads compared to the other three processes. Imbalanced communication time due to different task amount among the processes further extends the difference in slack length.

Critical Path (CP) is one particular task trace from the beginning task of one execution of a task-parallel application to the ending one with the total slack of zero. Any delay on tasks on the CP increases the total execution time of the application, while dilating tasks off the CP into their slack individually without further delay does not cause performance loss as a whole. Energy savings can be achieved by appropriately reducing frequency to dilate tasks off the CP into their slack as much as possible, which is referred to as the *CP* approach. Numerous existing OS level solutions effectively save energy via *CP-aware* analysis [3] [23] [8] [9] [24]. Figure 3 highlights one CP for Cholesky factorization with bold edges. We next present a feasible algorithm to generate a CP via TDS analysis.

V. TX: ENERGY EFFICIENT RACE-TO-HALT DVFS SCHEDULING VIA ALGORITHMIC TDS ANALYSIS

Next we present in detail the three energy efficient DVFS scheduling approaches for distributed dense matrix factorizations individually: The *SC* approach, the *CP* approach, and our TX approach. We further demonstrate that TX manages to save substantial energy for distributed dense matrix factorizations, since via *TDS-based race-to-halt*, it circumvents the defective prediction mechanism employed by the *CP* approach at OS level, and further saves energy from potential load imbalance. Table I lists the notation used henceforth in this section.

A. Custom Functions

In Algorithms 1 and 2, six custom functions are introduced for readability: delete(TDS(t_1), t_2), SetFreq(), IsLastInstance(), Send(), Recv(), and IsFinished(). The implementation of delete() is straightforward: Remove t_2 from the TDS of t_1 (its counterpart insert(TDS(t_1), t_2) is also implemented to add task t_2 into the TDS of task t_1, not shown in the algorithms). SetFreq() is a wrapper of DVFS APIs that set specific CPU frequencies, and Send() and Recv() are wrappers of MPI communication routines that send and receive flag messages among tasks respectively. IsLastInstance() is employed to determine if the current task is the last instance of the same type of tasks operating the same data block, and IsFinished() is employed to determine if the current task is finished: Both are easy to implement at library level.

B. Scheduled Communication Approach

One effective and straightforward solution to save energy for task-parallel applications is to set CPU frequency to high during computation, while set it to low during communication, given the fact that large-message communication is not bound by CPU performance while computation is, so the peak CPU performance is not necessary during communication. Although substantial energy savings can be achieved from the Scheduled Communication (SC) approach [23] [8], it leaves potential energy saving opportunities from other types of slack (e.g., see slack shown in Figure 3) untapped. More energy savings can be fulfilled via fine-grained analysis of execution characteristics of the applications, in particular during non-communication. Next we present two well-designed approaches that take advantage of computation slack to further energy savings. Note since the *SC* approach does not conflict with solutions exploiting slack from non-communication, it can be incorporated with the next two solutions seamlessly to maximize energy savings.

C. Critical Path Approach vs. TX Approach

Given a detected CP (e.g., via static analysis [3] or local information analysis [9]) for task-parallel applications, the Critical Path (CP) approach saves energy as shown in Algorithm 1 and Figure 3: For all tasks on the CP, the working CPU frequency is set to the highest for attaining the peak CPU performance, while for tasks not on the CP whose total slack is larger than zero (e.g., tasks with no outgoing *explicit* data dependencies in Figure 3), lowering frequency appropriately is performed to dilate the tasks into their slack as much as possible, without incurring performance loss of the applications. Due to the discrete domain of available CPU frequencies defined for DVFS, if the calculated optimal frequency that can eliminate slack lies in between two available neighboring frequencies, the two frequencies can be leveraged to approximate it by calculating a ratio of durations operating at the two frequencies. The two frequencies are then assigned to the durations separately based on the ratio. Lines 7-9 in Algorithm 1 sketch the frequency approximation method [9]. The ratio of split frequencies is calculated via prior knowledge of the mapping between frequency and execution time of different types of tasks. Note that we denote the two neighboring available frequencies of f_{opt} as $\lfloor f_{opt} \rfloor$ and $\lceil f_{opt} \rceil$.

Different from the *CP* approach that reduces CPU frequency for tasks off the CP to eliminate slack without performance loss, TX employs a *race-to-halt* mechanism that leverages TDS$_{in}$ and TDS$_{out}$ of each task to determine the timing of *race* and *halt*, as shown in Algorithm 2 and Figure 3. Respecting data dependencies, one *dependent* task cannot start until the finish of its *depended* task. TX keeps the *dependent* task staying at the lowest frequency until all its *depended* tasks have finished when it may start, and then allows the *dependent* task to work at the highest frequency to complete as soon as possible, before being switched back to the low-power state. A task sends a $DoneFlag$ to all its *dependent* tasks to notify them that data needed has been processed and ready for use. A *dependent* task is retained at the lowest frequency while waiting for $DoneFlags$ from all its *depended* tasks, and removes the dependency to a *depended* task from its TDS$_{in}$, once a $DoneFlag$ from the *depended* task is received.

VI. Implementation and Evaluation

We have implemented TX, and for comparison purposes, the library level *SC* approach to evaluate the effectiveness of TX to save energy during non-communication slack. For comparing with the OS level *SC* and *CP* approaches, we communicated with the authors of Adagio [9] and Fermata [8] and received the latest version of both implementations. We also compare with another OS level solution CPUSpeed [27], an interval-based DVFS scheduler that scales CPU performance according to runtime CPU utilization during the past interval. Regarding workload prediction, Adagio and Fermata leverage the PAST algorithm [28], and CPUSpeed uses a prediction algorithm similar to the RELAX algorithm employed in CPU MISER [11]. With application-specific knowledge known, library level solutions do not need the workload prediction mechanism. We denote the above approaches as follows:

- Orig: Original runs of different-scale distributed dense matrix factorizations without any energy saving approaches;
- SC_lib: A library level implementation of the *SC* approach;
- Fermata: An OS level implementation of the *SC* approach based on the PAST workload prediction algorithm;
- Adagio: An OS level implementation of the *CP* approach based on the PAST algorithm, where Fermata is incorporated;
- CPUSpeed: An OS level implementation of the *SC* approach based on a workload prediction algorithm similar to RELAX;
- TX: A library level implementation of the *race-to-halt* approach based on TDS analysis, where SC_lib is incorporated.

TABLE II. Hardware Configuration for Experiments.

Cluster	HPCL	ARC
System Size (# of Nodes)	8	108
Processor	2×Quad-core AMD Opteron 2380	2×8-core AMD Opteron 6128
CPU Freq.	0.8, 1.3, 1.8, 2.5 GHz	0.8, 1.0, 1.2, 1.5, 2.0 GHz
Memory	8 GB RAM	32 GB RAM
Cache	128 KB L1, 512 KB L2, 6 MB L3	128 KB L1, 512 KB L2, 12 MB L3
Network	1 GB/s Ethernet	40 GB/s InfiniBand
OS	CentOS 6.2, 64-bit Linux kernel 2.6.32	CentOS 5.7, 64-bit Linux kernel 2.6.32
Power Meter	PowerPack	Watts up? PRO

A. Experimental Setup

We applied all five energy saving approaches to distributed Cholesky/LU/QR factorizations with five different global matrix sizes each. Experiments were performed on two power-aware clusters: HPCL and ARC. Table II lists the hardware configuration of the two clusters. Note that we measured the total dynamic and leakage energy consumption using PowerPack [18], a comprehensive software and hardware framework for energy profiling and analysis of high performance systems and applications; the total of static and dynamic power consumption was measured using Watts up? PRO [29]. Both energy and power consumption refer to total energy and power costs respectively on all involved components of one node such as CPU, memory, disk, motherboard, etc. Due to shared power meter for three nodes of the ARC cluster, power consumption measured is for the total power consumption of three nodes, while energy consumption measured is for all energy costs collected from all eight nodes of the HPCL cluster. CPU frequency switching was implemented via CPUFreq [16] and directly modifying CPU frequency system configuration files.

B. Results

In this section, we present experimental results on power, energy, and performance efficiency and trade-off by comparing TX with the other energy efficient approaches, respectively.

POWER SAVINGS. First we evaluate the capability of power savings from the five energy saving approaches for distributed dense matrix factorizations on the ARC cluster (due to the similarity of results, data for distributed LU and QR factorizations is not shown), where power consumption is measured by sampling at a constant rate through the execution of the applications. Figure 4 depicts the total system power consumption of three nodes (out of sixteen nodes in use) running distributed Cholesky factorization with different solutions using a 160000 $\times$ 160000 global matrix, where we select time durations of the first few iterations. Among the six executions, there exist four power patterns (including the theoretical one CP_theo that calculates computation slack effectively, and lowers power to eliminate the slack): (a) Orig and CPUSpeed – employed the same highest CPU frequency for both computation and communication, resulting almost constant power consumption around 950 Watts; (b) SC_lib, Fermata, and Adagio – lowered down CPU frequency during the communication, i.e., the five low-power durations around 700 Watts, and resumed the peak CPU performance during the computation; (c) CP_theo – not only scheduled low power states for communication, but also slowed down computation to eliminate computation slack – this is a theoretical value curve instead of real measurement, which is how OS level approaches such as Adagio is supposed to save more power as a *CP-aware* approach; and (d) TX – employed the *race-to-halt* strategy to lower down CPU performance for all durations other than computation.

Specifically, upon a workload prediction algorithm that inspects dynamic prior CPU utilization periodically, CPUSpeed failed to produce accurate prediction and scale CPU power states accordingly: It kept the peak CPU power all the time. Either relying on known application characteristics (SC_lib) or detecting MPI communication calls (Fermata and Adagio), all three approaches can identify communication durations and apply DVFS accordingly. As discussed previously, solutions only slow down communication are semi-optimal. Adagio and TX are expected to utilize computation slack for achieving additional energy savings. Due to the defective OS level prediction mechanism, Adagio failed to predict behavior of future tasks and calculate computation slack accurately. Consequently no low-power states were switched to during computation

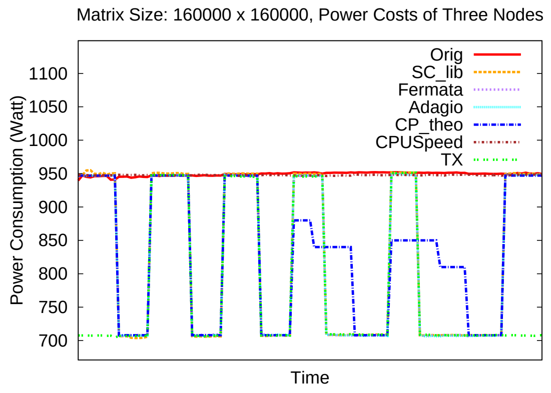

Fig. 4. Power Consumption of Distributed Cholesky Factorization with Different Energy Saving Approaches on a 16 $\times$ 16 Process Grid ARC Cluster.

for Adagio. Different from solutions saving energy via slack reclamation, TX relies on the *race-to-halt* mechanism where computation is conducted at the peak CPU performance and the lowest CPU frequency is employed immediately otherwise. The nature of *race-to-halt* also guarantees no high-power states are employed during waiting durations resulting from load imbalance and data dependency, i.e., the two low-power durations in green where the application starts and ends, additional energy saving opportunities exploited by TX only.

ENERGY SAVINGS. Next we compare energy savings achieved by all five approaches on the HPCL cluster, as shown in Figure 5, where energy consumption is measured by recording on/off collection of power and time costs when an application starts/ends. For eliminating errors from scalability, we collected energy and time data of five matrix factorizations with different global matrix sizes ranging from 5120 to 25600, respectively. Considerable energy savings are achieved by all approaches except for CPUSpeed, with similar energy saving trends among all approaches: TX prevails over all other approaches with higher energy efficiency; SC_lib, Fermata, and Adagio gain similar energy savings. Overall, for Cholesky, TX can save energy 30.2% on average and up to 33.8%; for LU and QR, TX can achieve 16.0% and 20.0% on average and up to 20.4% and 23.4% energy efficiency, respectively. Due to the reasons discussed for power savings, Adagio only achieves similar energy efficiency to SC_lib and Fermata, without fulfilling additional energy savings from slack reclamation of computation. With application-specific knowledge instead of workload prediction, TX manages to achieve energy savings during both computation and communication slack. Moreover, TX benefits from the advantage of saving more energy from load imbalance while other approaches cannot exploit. Next we further evaluate energy savings by increasing load imbalance.

EFFECTS OF BLOCK SIZE. As depicted in Figure 3 and discussed above, the additional energy savings can be achieved from potential load imbalance, i.e., the area only covered by green dashed boxes. Empirically, regardless of the workload partition techniques, load imbalance can grow due to larger tasks, longer communication, etc. For manifesting the strength of TX in achieving additional energy savings for *completeness*, we deliberately imbalance the workload through expanding tasks by using greater block sizes for Cholesky, while keeping the default block size for LU and QR. As shown in Figure 5, the average energy savings fulfilled by TX for Cholesky (30.2%) are consequently greater than LU and QR (16.0%

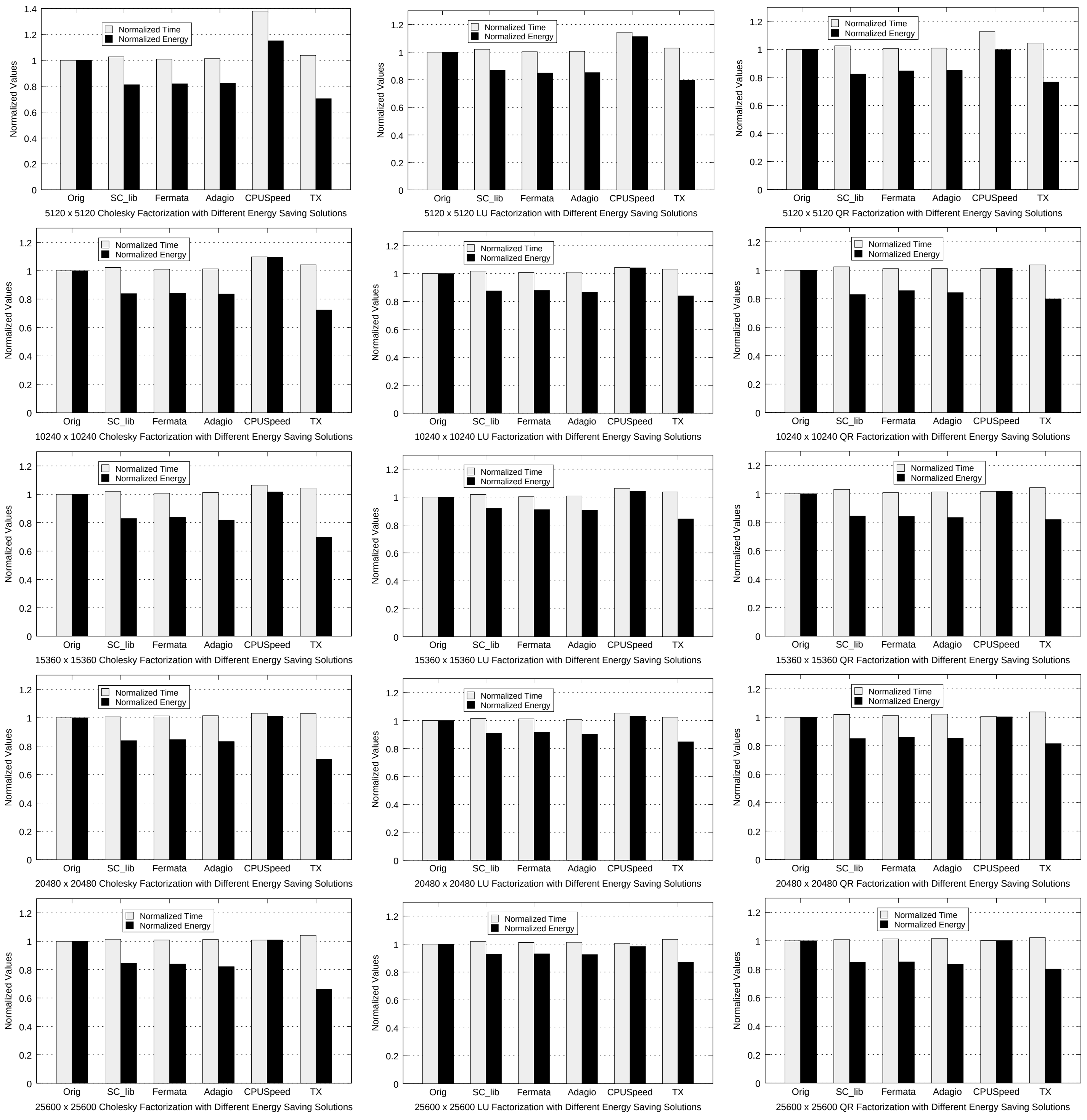

Fig. 5. Energy and Performance Efficiency of Distributed Cholesky/LU/QR Factorizations with Different Energy Saving Approaches on the HPCL Cluster.

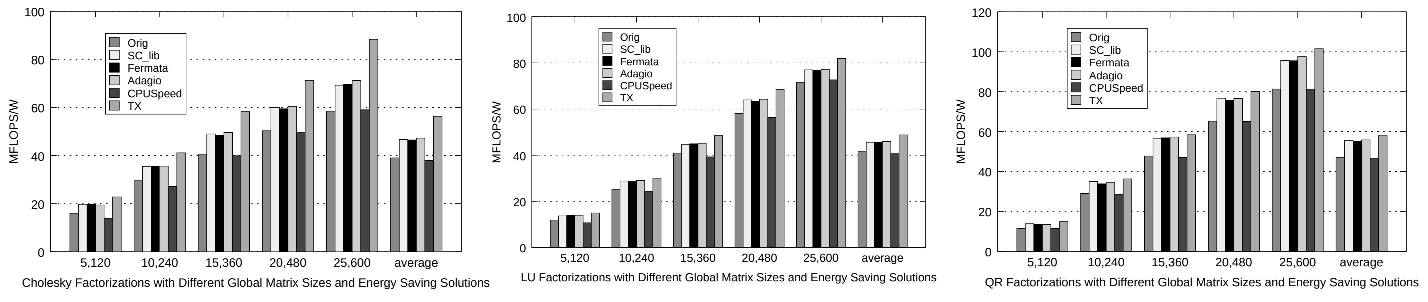

Fig. 6. Energy and Performance Trade-off of Distributed Cholesky/LU/QR Factorizations with Different Energy Saving Approaches on the HPCL Cluster.

and 20.0%). Compared to the second most effective approach Adagio, TX can save Cholesky 12.8% more energy on average.

PERFORMANCE LOSS. Figure 5 also illustrates performance loss from different energy saving approaches against the original executions. We can see TX only incurs negligible time overhead: 3.8%, 3.1%, 3.7% on average for Cholesky, LU, and QR individually, similar to the time overhead of all other approaches except for CPUSpeed. The minor performance loss on employing these techniques is primarily originated from three facts: (a) Although large-message communication is not CPU-bound, pre-computation required for starting up a communication link before any data transmission is necessary and is affected by CPU performance, so the low-power state during communication can slightly degrade performance; (b) switching CPU frequency via DVFS is essentially implemented by modifying CPU frequency system configuration files, and thus minor overhead is incurred from such in-memory file read/write operations [14]; and (c) CPU frequency transition latency is required for the newly-set frequency to take effect. Further, TX suffers from minor performance loss from TDS analysis, including TDS and CP generation, and maintaining TDS for each task. The high time overhead of CPUSpeed is another reason for its little and even negative energy savings besides the defective prediction mechanism at OS level.

ENERGY/PERFORMANCE TRADE-OFF. A optimal energy saving approach requires to achieve the maximal energy savings with the minimal performance loss. Per this requirement, energy-performance integrated metrics are widely employed to quantify if the energy efficiency achieved and the performance loss incurred are well-balanced. We adopt Energy-Delay Product (EDP) to evaluate the overall energy and performance trade-off of the five approaches, in terms of MFLOPS/W, which evaluates the amount of floating-point operations per second within the unit of one Watt (i.e., the greater value, the better efficiency). As shown in Figure 6, compared to other approaches, TX is able to fulfill the most balanced trade-off between energy and performance, with similar trends in power/energy savings discussed above. Specifically, TX has higher MFLOPS/W values for Cholesky compared to LU and QR, due to the higher energy savings achieved from the more imbalanced load without additional performance loss.

VII. CONCLUSIONS

The looming overloaded energy consumption of high performance scientific computing brings significant challenges to green computing in this era of ever-growing power costs for large-scale computing systems. DVFS techniques have been widely employed to improve energy efficiency for task-parallel applications. With high generality, OS level solutions are regarded as feasible energy saving approaches for such applications. We observe for applications with variable execution patterns, OS level solutions suffer from defective prediction mechanism and untapped potential energy savings from load imbalance, and thus cannot optimize energy efficiency. Giving up partial generality, the proposed library level approach TX is evaluated to save more energy with little performance loss for distributed Cholesky/LU/QR factorizations on two power-aware clusters compared to classic OS level solutions.

REFERENCES

[1] *TOP500 Supercomputer Lists*. http://www.top500.org/.

[2] *Green500 Supercomputer Lists*. http://www.green500.org/.

[3] G. Chen, K. Malkowski, M. Kandemir, and P. Raghavan, "Reducing power with performance constraints for parallel sparse applications," in *IPDPS*, 2005, pp. 1–8.

[4] V. W. Freeh and D. K. Lowenthal, "Using multiple energy gears in MPI programs on a power-scalable cluster," in *PPoPP*, 2005, pp. 164–173.

[5] R. Ge, X. Feng, and K. W. Cameron, "Performance-constrained distributed DVS scheduling for scientific applications on power-aware clusters," in *SC*, 2005, p. 34.

[6] N. Kappiah, V. W. Freeh, and D. K. Lowenthal, "Just in time dynamic voltage scaling: Exploiting inter-node slack to save energy in MPI programs," in *SC*, 2005, p. 33.

[7] R. Springer, D. K. Lowenthal, B. Rountree, and V. W. Freeh, "Minimizing execution time in MPI programs on an energy-constrained, power-scalable cluster," in *PPoPP*, 2006, pp. 230–238.

[8] B. Rountree, D. K. Lowenthal, S. Funk, V. W. Freeh, B. R. de Supinski, and M. Schulz, "Bounding energy consumption in large-scale MPI programs," in *SC*, 2007, pp. 1–9.

[9] B. Rountree, D. K. Lowenthal, B. R. de Supinski, M. Schulz, V. W. Freeh, and T. Bletsch, "Adagio: Making DVS practical for complex HPC applications," in *ICS*, 2009, pp. 460–469.

[10] K. H. Kim, R. Buyya, and J. Kim, "Power aware scheduling of bag-of-tasks applications with deadline constraints on DVS-enabled clusters," in *CCGrid*, 2007, pp. 541–548.

[11] R. Ge, X. Feng, W.-C. Feng, and K. W. Cameron, "CPU MISER: A performance-directed, run-time system for power-aware clusters," in *ICPP*, 2007, p. 18.

[12] X. Wang, X. Fu, X. Liu, and Z. Gu, "Power-aware CPU utilization control for distributed real-time systems," in *RTAS*, 2009, pp. 233–242.

[13] L. Tan, Z. Chen, Z. Zong, R. Ge, and D. Li, "A2E: Adaptively aggressive energy efficient DVFS scheduling for data intensive applications," in *IPCCC*, 2013, pp. 1–10.

[14] L. Tan, L. Chen, Z. Chen, Z. Zong, R. Ge, and D. Li, "HP-DAEMON: High performance distributed adaptive energy-efficient matrix-multiplication," in *ICCS*, 2014, pp. 599–613.

[15] H. David, C. Fallin, E. Gorbatov, U. R. Hanebutte, and O. Mutlu, "Memory power management via dynamic voltage/frequency scaling," in *ICAC*, 2011, pp. 31–40.

[16] *CPUFreq - CPU Frequency Scaling*. https://wiki.archlinux.org/index.php/CPU_Frequency_Scaling.

[17] *NVIDIA System Management Interface (nvidia-smi)*. https://developer.nvidia.com/nvidia-system-management-interface/.

[18] R. Ge, X. Feng, S. Song, H.-C. Chang, D. Li, and K. W. Cameron, "PowerPack: Energy profiling and analysis of high-performance systems and applications," *IEEE Transactions on Parallel and Distributed Systems*, vol. 21, no. 5, pp. 658–671, May 2010.

[19] A. Miyoshi, C. Lefurgy, E. V. Hensbergen, R. Rajamony, and R. Rajkumar, "Critical power slope: Understanding the runtime effects of frequency scaling," in *ICS*, 2002, pp. 35–44.

[20] C.-H. Hsu and W.-C. Feng, "A power-aware run-time system for high-performance computing," in *SC*, 2005, p. 1.

[21] *Scalable Linear Algebra PACKage*. http://www.netlib.org/scalapack/.

[22] *DPLASMA: Distributed Parallel Linear Algebra Software for Multicore Architectures*. http://icl.cs.utk.edu/dplasma/.

[23] M. Y. Lim, V. W. Freeh, and D. K. Lowenthal, "Adaptive, transparent frequency and voltage scaling of communication phases in MPI programs," in *SC*, 2006, p. 107.

[24] P. Alonso, M. F. Dolz, R. Mayo, and E. S. Quintana-Ortí, "Improving power efficiency of dense linear algebra algorithms on multi-core processors via slack control," in *HPCS*, 2011, pp. 463–470.

[25] C. Liu, A. Sivasubramaniam, M. Kandemir, and M. J. Irwin, "Exploiting barriers to optimize power consumption of CMPs," in *IPDPS*, 2005.

[26] P. Alonso, M. F. Dolz, F. D. Igual, R. Mayo, and E. S. Quintana-Ortí, "DVFS-control techniques for dense linear algebra operations on multi-core processors," *CSRD*, vol. 27, no. 4, pp. 289–298, Nov. 2012.

[27] *CPUSpeed*. http://carlthompson.net/Software/CPUSpeed/.

[28] M. Weiser, B. Welch, A. Demers, and S. Shenker, "Scheduling for reduced CPU energy," in *OSDI*, 1994, p. 2.

[29] *Watts up? Meters*. https://www.wattsupmeters.com/.