New Model-based Methods and Algorithms for Performance and Energy Optimization of Data Parallel Applications on Homogeneous Multicore Clusters

Alexey Lastovetsky, *Member, IEEE,* and Ravi Reddy

Abstract—Modern homogeneous parallel platforms are composed of tightly integrated multicore CPUs. This tight integration has resulted in the cores contending for various shared on-chip resources such as Last Level Cache (LLC) and interconnect, leading to resource contention and non-uniform memory access (NUMA). Due to these newly introduced complexities, the performance and energy profiles of real-life scientific applications on these platforms are not smooth and may deviate significantly from the shapes that allowed traditional and state-of-the-art load balancing algorithms to minimize their computation time.
In this paper, we propose new model-based methods and algorithms for minimization of time and energy of computations for the most general shapes of performance and energy profiles of data parallel applications observed on the modern homogeneous multicore clusters. We formulate the performance and energy optimization problems and present efficient algorithms of complexity $O(p^2)$ solving these problems where p is the number of processors. It is important to note that the globally optimal solutions found by these algorithms may not load-balance the application.
We experimentally study the efficiency and scalability of our algorithms for two data parallel applications, matrix multiplication and fast Fourier transform, on a modern multicore CPU and clusters of such CPUs. We also demonstrate the optimality of solutions determined by our algorithms.

Index Terms—performance, energy, power, performance optimization, energy optimization, functional performance models, energy models, data partitioning, load balancing, homogeneous multicore clusters

1 Introduction

Modern homogeneous parallel platforms are composed of tightly integrated multicore CPUs with highly hierarchical arrangement of cores. This tight integration has resulted in the cores contending for various shared on-chip resources such as Last Level Cache (LLC) and interconnect (For example: Intel's Quick Path Interconnect [1], AMD's Hyper Transport [2]), leading to resource contention and non-uniform memory access (NUMA). These newly introduced complexities have created fresh formidable challenges for model and algorithm designers.

To elucidate these challenges, we analyze the limitations of the state-of-the-art load balancing algorithms for performance optimization [3], [4], [5], [6], [7]. We select two widely known and highly optimized scientific routines, OpenBLAS DGEMM [8] and FFTW [9], for this purpose. The goal of these load balancing algorithms is to minimize the computation time of the application. The intuition behind the notion that load balancing the application improves its performance is the following: a balanced application does not waste processor cycles in waiting at points of synchronization and data exchange, therefore maximizing the utilization of the processors. The key input to these load balancing algorithms is the functional performance model (FPM) [3], [4], [5], which represents the speed of processor by a continuous function of the problem size. In addition to the property of continuity, the speed function is assumed to be smooth enough satisfying one of the following assumptions on its shape:

1) Along each of the problem size variables, the function is monotonically decreasing
2) There exists point x such that
   - On the interval $[0, x]$, the function is
     – monotonically increasing,
     – concave, and
     – any straight line coming through the origin of the coordinate system intersects the graph of the function in no more than one point.
   - On the interval $[x, \infty)$, the function is monotonically decreasing

These very restrictions on the shape of speed functions guarantee that the FPM-based load balancing algorithms, proposed in [6], [10], [11], [7], [12], [13], [14], [15], [16], always return a unique solution that minimizes the computation time. Also these restrictions are trivially satisfied when the speed of processor is modelled by a constant.

The smooth FPMs accurately capture the shapes of real-life scientific applications on platforms consisting of uniprocessors (single-core CPUs). This is clearly illustrated in

• *A. Lastovetsky and R. Reddy are with the School of Computer Science, University College Dublin, Belfield, Dublin 4, Ireland.*
E-mail: alexey.lastovetsky@ucd.ie,ravi.manumachu@ucd.ie

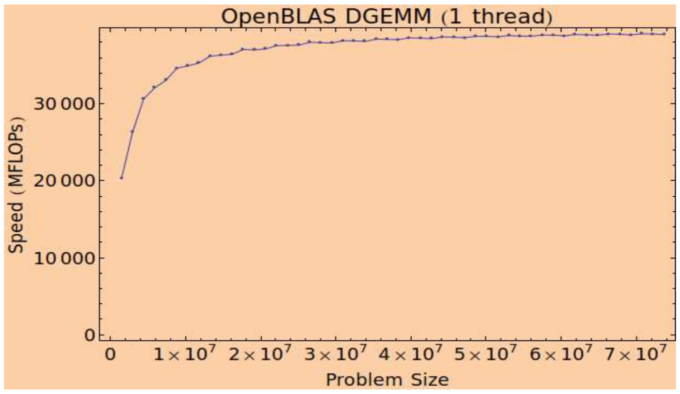


Fig. 1. Speed function of OpenBLAS DGEMM application executed on a single core on the Intel Haswell server.

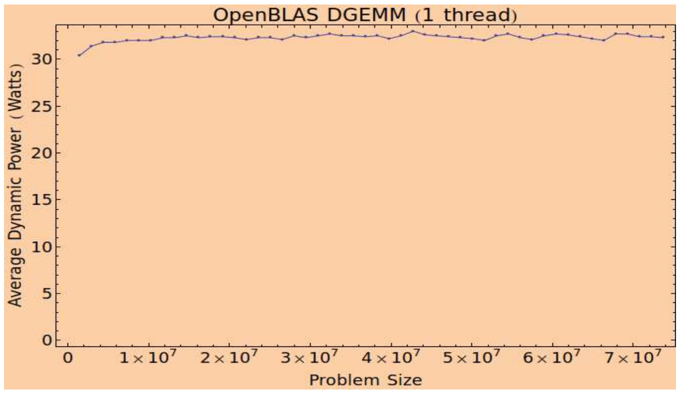


Fig. 2. Average dynamic power consumption of OpenBLAS DGEMM application executed on a single core on the Intel Haswell server.

Figure 1, which shows the speed function of the OpenBLAS DGEMM application built experimentally by executing it on a single core of an Intel Haswell server (specification shown in Table 1). The application multiples two square matrices of size $n \times n$ (problem size being equal to n^2). In these experiments, the *numactl* tool is used to bind the application to one core. Fig. 2 and 3 respectively show the average dynamic power consumption and dynamic energy consumption of the application. The power and energy consumptions are obtained using Watts Up Pro power meter. We must mention that there are two types of energy consumptions, dynamic energy and static energy. We define the static energy consumption as the energy consumption of the platform without the application execution. Dynamic energy consumption is calculated by subtracting this static energy consumption from the total energy consumption of the platform during the application execution measured using the Watts Up Pro. In this work, we consider only the dynamic energy consumption because static energy consumption is a constant (inherent property) of a platform and will be the same for different application configurations. However, we would like to mention that static power consumption can be easily incorporated in our problem formulations and algorithms.

One can observe that the shapes of the performance and energy graphs are smooth with minimal variations, and the performance graphs comfortably satisfy the conditions imposed by the FPMs that are crucial for the correct operation of the load balancing algorithms.

Now, due to the newly introduced complexities in modern homogeneous multicore clusters such as resource contention and NUMA, the performance and energy profiles of real-life scientific applications executing on these platforms are not smooth and may deviate significantly from the shapes that allowed traditional and state-of-the-art load balancing algorithms to find optimal solutions. This is illustrated in Fig. 4, 5, and 6 which show respectively the speed function, average dynamic power, and dynamic energy consumption graphs of OpenBLAS DGEMM application executed on the Intel Haswell server and employing varying number of threads. One can see that as the number of threads executing in the application increases, the fluctuations increase reaching the peak when the number of threads is 24 equalling the total number of physical cores

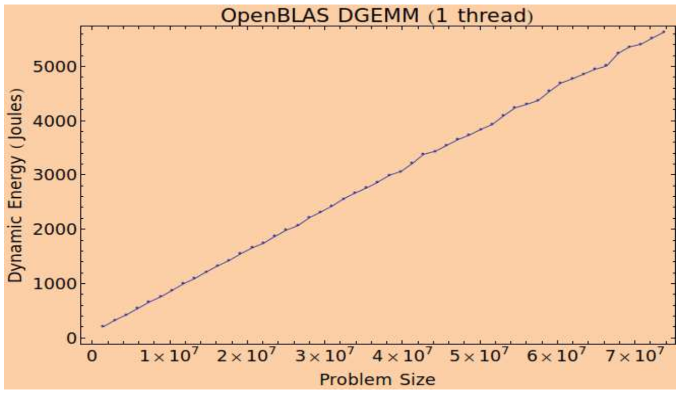


Fig. 3. Dynamic energy consumption of OpenBLAS DGEMM application executed on a single core on the Intel Haswell server.

TABLE 1
Specification of the Intel Haswell server used to build the FPM and energy model.

Technical Specifications	Intel Haswell Server
Processor	Intel E5-2670 v3 @ 2.30GHz
OS	CentOS 7
Microarchitecture	Haswell
Memory	64 GB
Socket(s)	2
Core(s) per socket	12
NUMA node(s)	2
L1d cache	32 KB
L1l cache	32 KB
L2 cache	256 KB
L3 cache	30720 KB
TDP	240 W
Base Power	58 W

in the server. The full speed, power, and energy functions for number of threads equal to 24 are shown in Fig. 7, 8, and 9 respectively.

To make sure the experimental results are reliable, we follow a detailed methodology, which is described in Section 1 of the supplemental. It contains the following main steps: 1). We make sure the server is fully reserved and dedicated to our experiments and is exhibiting clean and normal behaviour by monitoring its load continuously for a week. 2). For each data point in the speed, power and, energy functions of an application, the sample mean is used, which

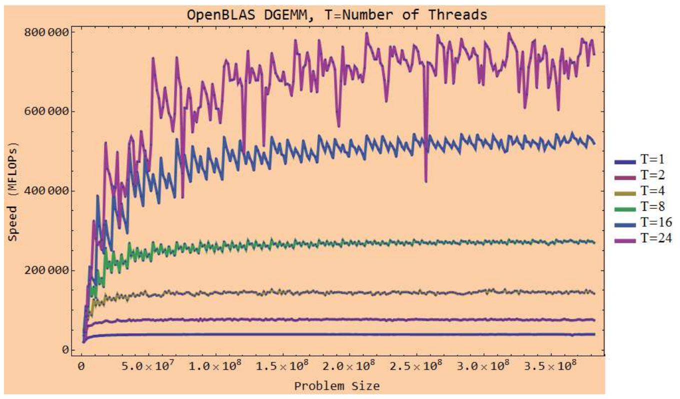


Fig. 4. Speed function of OpenBLAS DGEMM application executing varying number of threads (T) on the Intel Haswell server.

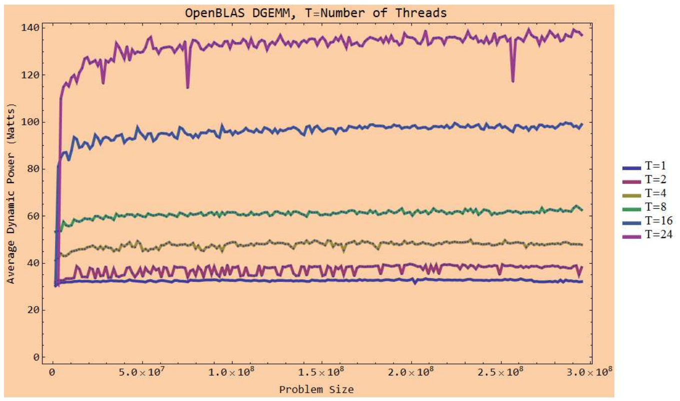


Fig. 5. Function of average dynamic power consumption against problem size for OpenBLAS DGEMM application executing varying number of threads (T) on the Intel Haswell server.

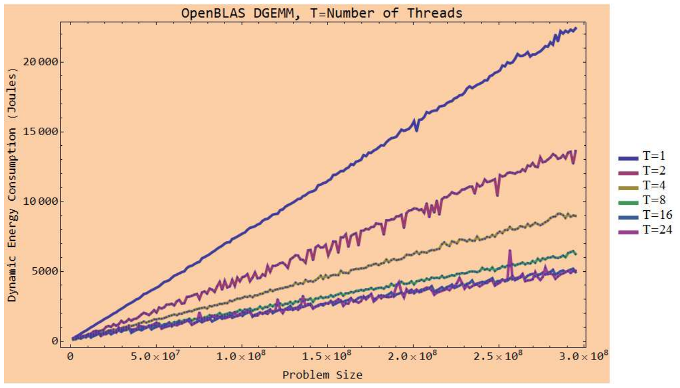


Fig. 6. Function of dynamic energy consumption against problem size for OpenBLAS DGEMM application executing varying number of threads (T) on the Intel Haswell server.

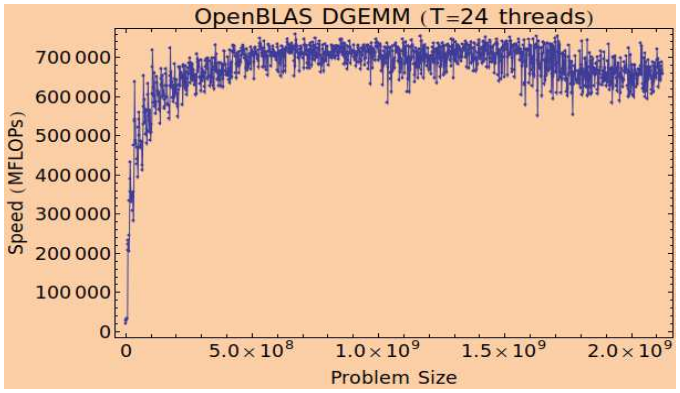


Fig. 7. Speed function of OpenBLAS DGEMM application for $T = 24$ on the Intel Haswell server.

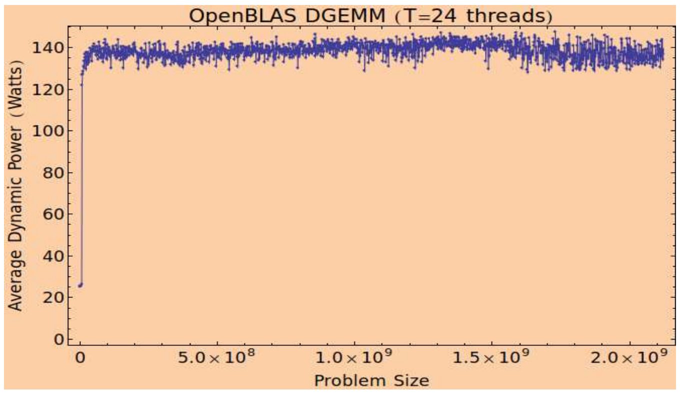


Fig. 8. Function of average dynamic power consumption against problem size for OpenBLAS DGEMM application for $T = 24$ on the Intel Haswell server.

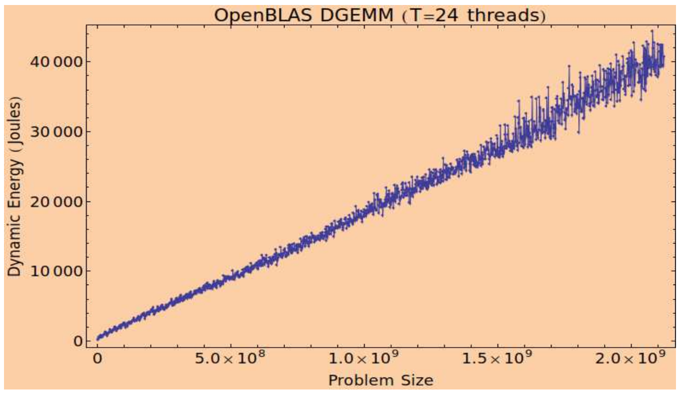


Fig. 9. Function of dynamic energy consumption against problem size for OpenBLAS DGEMM application for $T = 24$ on the Intel Haswell server.

is calculated by executing the application repeatedly until the sample mean lies in the 95% confidence interval and a precision of 0.025 (2.5%) has been achieved. For this purpose, Student's t-test is used assuming that the individual observations are independent and their population follows the normal distribution. We verify the validity of these assumptions by plotting the distributions of observations.

Therefore, the variation observed is not noise but is an inherent trait of applications executing on multicore servers with resource contention and NUMA.

There are some other interesting observations about these variations.

- They are noticeable even when smaller number of threads are used. This can be seen from the average dynamic power and dynamic energy plots of OpenBLAS DGEMM application (Fig. 5 and 6 respectively) for number of threads equal to 2.
- They can be quite large. This is evident from Fig. 10, 11, and 12, which respectively show the speed

function, average dynamic power, and dynamic energy consumption graphs for multi-threaded FFTW application executed with 24 threads on the Intel Haswell server. The application performs a 2D FFT of size $n \times n$ (the problem size being n^2). From the speed function plot, one can observe performance drops of around 70% for many problem sizes.

- The variations presented in the paper cannot be explained by the constant and stochastic fluctuations due to OS activity or a workload executing in a node in common networks of computers. In such networks, a node is persistently performing minor routine computations and communications by being an integral part of the network. Examples of such routine applications include e-mail clients, browsers, text editors, audio applications, etc. As a result, the node will experience constant and stochastic fluctuations in the workload. This changing transient load will cause a fluctuation in the speed of the node in the sense that the speed will vary for different runs of the same workload. One way to represent these inherent fluctuations in the speed is to use a speed band rather than a speed function. The width of the band characterizes the level of fluctuation in the speed due to changes in load over time [3], [4], [5]. For a node with single-core CPUs, the width of the band has been shown to decrease as the problem size increases. For a node with a very high level of network integration, typical widths of the speed bands were observed to be around 40% for small problem sizes and narrowing down to 3% for large problem sizes. Therefore, as the problem size increases, the width of the speed band is observed to decrease. Therefore, for long running applications, one would observe the width to become quite narrow (3%). However, this is not the case for variations in the presented graphs. The dynamic energy consumption in Fig. 9 and 12 (for the number of threads equal to 24) show the widths of the variations increasing as problem size increases. These widths reach a maximum of 70% and 125% respectively for large problem sizes. The speed functions in Fig. 7 and 10 (for the number of threads equal to 24) demonstrate that the widths are bounded with the averages around 17% and 60% respectively. This suggests therefore that the variation is largely due to the newly introduced complexities and not due to the fluctuations arising from changing transient load.

We believe that these variations will become typical because chip manufacturers are increasingly favouring and thereby rapidly progressing towards tighter integration of processor cores, memory, and interconnect in their products. To discern how they limit the applicability of the load balancing algorithms, we zoom into the speed function of the multi-threaded OpenBLAS DGEMM application executing 24 threads on the Intel Haswell server. Figure 13 shows the speed function between two arbitrarily chosen points A and B. Assume, for the sake of simplicity, that we are allowed to use only this partial speed function in our algorithms. One can observe that the speed function is characterized

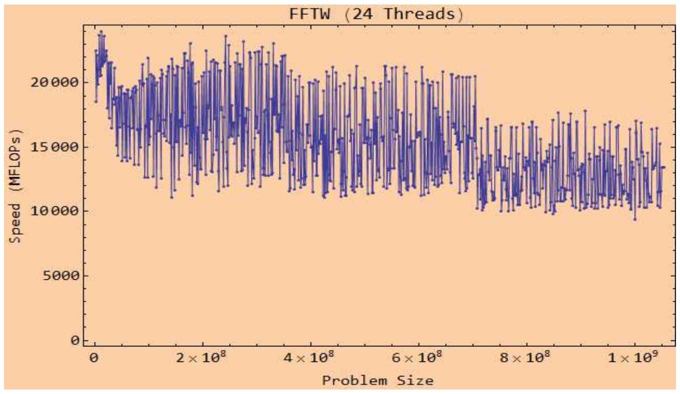


Fig. 10. Speed function of FFTW application executing 24 threads on the Intel Haswell server.

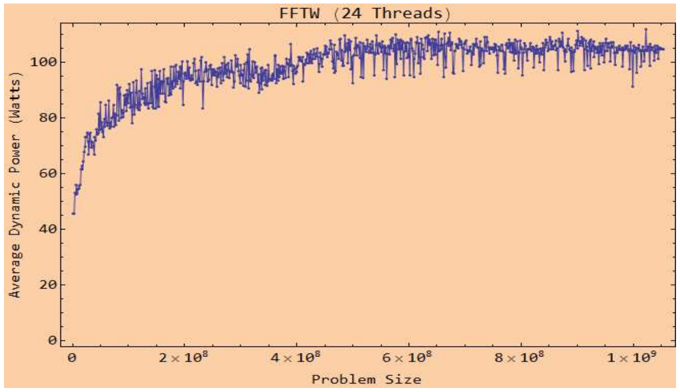


Fig. 11. Function of average dynamic power consumption against problem size for FFTW executing 24 threads on the Intel Haswell server.

by many local minima ($Q_1, Q_2, ...$) and many local maxima ($P_1, P_2, ...$). There is one global maximum P and one global minimum Q. This highly wavering shape is the reason why the load balancing algorithms will return non-optimal solutions. For example, consider 2 processors solving a workload of size $n = 40$. The load balancing algorithms will output the distribution, $(Q, Q) = (\frac{n}{p}, \frac{n}{p}) = (20, 20)$. However, one can see that the solution given by a distribution composed from the neighbouring points, $(P_3, P_4) = (19, 21)$, is better. Similarly, for the other local optima, Q_1, Q_2, and Q_3, one can find neighbouring points that yield better solutions than those given by the load balancing algorithms.

To summarize, the new inherent complexities introduced in modern homogeneous multicore clusters limit the applicability of state-of-the-art performance models and load balancing algorithms thereby necessitating either a thorough redesign or development of novel models and algorithms.

In this paper, we propose novel model-based methods and algorithms for minimization of time and energy of computations for the most general performance and energy profiles of data parallel applications executing on homogeneous multicore clusters. We formulate the performance and energy optimization problems and present efficient algorithms (called POPTA and EOPTA respectively) of complexity $O(p^2)$ solving these problems where p is the number of processors. Unlike load balancing algorithms, optimal solutions found by these algorithms may not load-balance

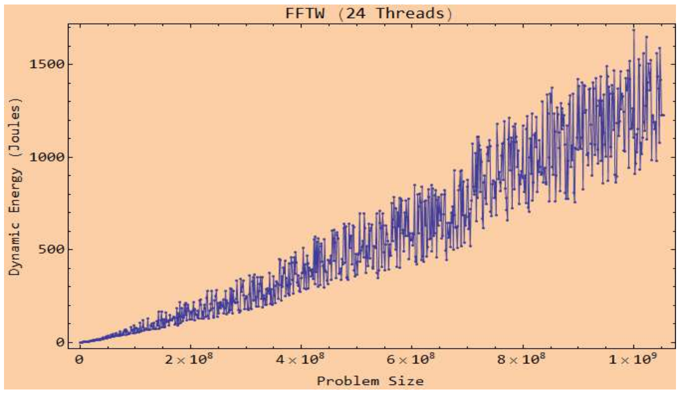


Fig. 12. Function of dynamic energy consumption against problem size for FFTW executing 24 threads on the Intel Haswell server.

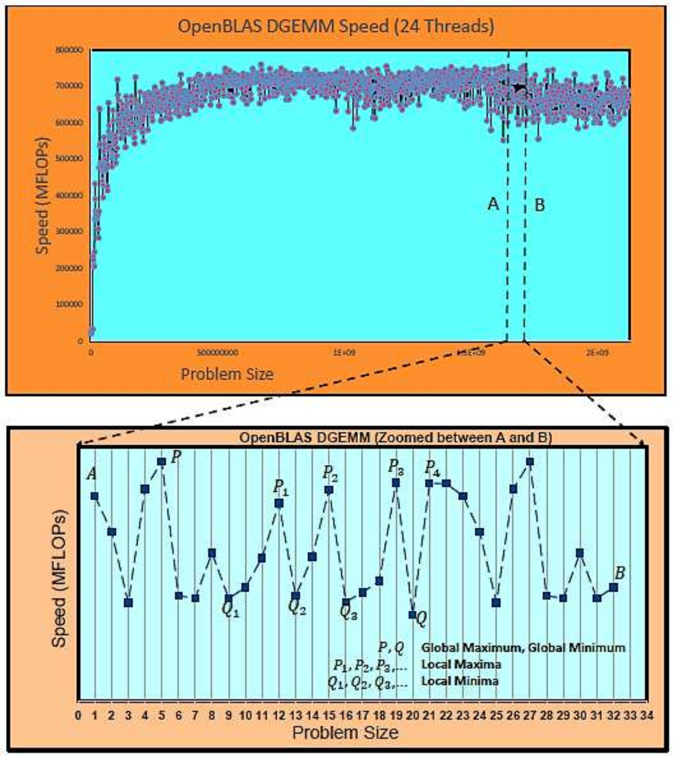


Fig. 13. Zoomed speed function of OpenBLAS DGEMM application between two arbitrarily chosen points A and B. The problem size x is normalized and shown in multiples of minimum granularity.

the application.

We summarize below the fundamental idea behind our optimization technique:

- To apply our algorithms, the user takes a data-parallel application that has been optimized for *load-balanced* execution on a cluster of identical multicore nodes.
- This application is now treated as a black-box and executed on one node for a range of problem sizes to construct its speed, power, and energy functions. When it is executed to obtain an experimental point of a function, the optimal values of the tunables (such as the number of threads) that have already been determined are used. The goal here is not to perform multi-parameter optimization where the user finds the optimal values of these tunables. It is assumed that this optimization has already been done and the optimal values of the tunables have already been determined before this step.
- The speed and energy functions are then input to POPTA and EOPTA, which determine optimal workload distributions that may not load-balance the application and that minimize the execution time and the energy consumption of computations in the parallel execution of the application. Thus, our optimization technique takes a configuration of the application, which has been obtained by one of the traditional load-balancing optimization techniques, and tries to further improve it by applying the single-parameter load-imbalancing optimization algorithms.

To summarize, the main goal of our optimization technique is not intra-node optimizations but inter-node optimization of performance and energy consumption of the application.

Based on our experiments with two data parallel applications on a modern multicore CPU and simulations on clusters of such CPUs, we demonstrate the optimality of solutions determined by our algorithms. We show significant average and maximum percentage improvements in performance and energy compared to traditional load-balanced workload distribution. We also confirm experimentally that the improvements in performance and energy are constant for a fixed problem size per processor. We further find that optimizing for performance alone produces significant energy reduction whereas optimizing for energy alone can cause major degradation in performance. Our conclusions are valid for arbitrary number of processors suggesting large positive implications for extreme-scale parallel platforms.

To summarize, our main contributions in this paper are:

- Explanation of the challenges introduced to model and algorithm design for optimization of performance and energy by the brand new complexities of resource contention and NUMA present in modern homogeneous multicore clusters.
- Efficient algorithms for performance and energy optimization problems for homogeneous multicore clusters.
- Experimental study of efficiency and scalability of our algorithms with conclusions that have large positive implications for extreme-scale parallel platforms.

The rest of the paper is structured as follows. Section 2 presents related work on performance and energy models for homogeneous and heterogeneous parallel platforms. Section 3 contains formulations of performance and energy optimization problems for homogeneous multicore clusters. Section 4 presents an efficient algorithm solving the performance optimization problem for homogeneous multicore clusters. Section 5 presents an efficient algorithm solving the energy optimization problem for homogeneous multicore clusters. Section 6 contains experimental analysis of the algorithms. Section 7 concludes the paper.

2 Related Work

Although the algorithms that we present in this paper use load-imbalancing technique, we briefly look at various categories of load-balancing algorithms since they have been a dominant class of algorithms for performance optimization on parallel platforms. We then discuss the performance and energy models for parallel platforms.

2.1 Load-Balancing Algorithms

There are several classifications of load-balancing algorithms: *static* or *dynamic*, *non-centralized* or *centralized*, *task queue* or *predicting-the-future*. One can find edifying descriptions of these classifications in [11], [17], [18].

Static algorithms, such as those based on data partitioning [19], [20], [4], use *a priori* information about the parallel application and platform. These algorithms are also known as *predicting-the-future* because they rely on accurate performance models as input to predict the future execution of the application. They are particularly useful for applications where data locality is important because they do not require data redistribution. However, these algorithms may be unsuitable for non-dedicated platforms, where load changes with time. Dynamic algorithms, such as task scheduling and work stealing [21], [22], [23], balance the load by moving fine-grained tasks between processors during the execution. They do not require *a priori* information about execution but may incur large communication overhead due to data migration. They can use static partitioning for the initial step due to its provably near-optimal communication cost, bounded tiny load imbalance, and lesser scheduling overhead.

In non-centralised algorithms [24], [25], load is migrated locally between neighbouring processors, while in centralised ones [26], [27], [28], load is distributed based on global load information. Non-centralised algorithms are slower to converge. At the same time, centralised algorithms typically have higher overhead. The centralised algorithms can be further subdivided into two groups: task queue [27] and predicting-the-future [26], [28].

2.2 Performance and Energy Models for Parallel Platforms

Over the years, load balancing algorithms developed for performance optimization on parallel platforms have attempted to take into consideration the real-life behaviour of applications executing on these platforms. This can be discerned from the evolution of performance models for computation used in these algorithms. The simplest models used positive constant numbers and different notions such as normalized processor speed, normalized cycle time, task computation time, average execution time, etc to characterize the speed of an application [29], [30], [31]. A singular feature of these efforts is that the performance of a processor is assumed to have no dependence on the size of the workload. The most advanced load balancing algorithms use functional performance models (FPMs), which are application-specific and represent the speed of a processor by continuous function of problem size but satisfying some assumptions on its shape [3], [4]. These FPMs capture accurately the real-life behaviour of applications executing on nodes consisting of uniprocessors (single-core CPUs).

Modern parallel platforms have complex nodal architectures with highly hierarchical arrangement and tight integration of processors where resource contention and NUMA are inherent complexities. On these platforms, the traditional and state-of-the-art performance models are deficient and the load balancing algorithms based on these models may return non-optimal solutions. Therefore, one must develop novel techniques for performance optimization for these platforms. In [32], [18], the authors propose an optimization technique reusing an advanced performance model of computation (FPMs) but using novel load distribution to minimize the computation time of the application. First, they experimentally build the speed function of the application using a wide range of problem sizes separated by minimum granularity. They then use this function and its connected visual picture to distribute computations unevenly between homogeneous groups of cores of the Xeon Phi co-processor, therefore load imbalancing the application, to achieve performance optimization. This is the first work where the load-imbalancing technique is applied to partition the workload minimizing the computation time of its parallel execution. However, no general partitioning algorithm is proposed in this work.

We now review methods for determining energy consumption of applications executing on modern parallel platforms. We also report the limitations of model-based methods arising from the inherent complexities newly introduced in modern parallel platforms.

There are two methods to determine the dynamic energy consumption of an application execution. One is by direct measurement using a power meter and the other is a model based approach. The dynamic energy consumption is calculated by deducting the base energy consumption (base power multiplied by the execution time) from the total energy consumption. In the model based approach, there are two ways to predict the energy consumption of an application execution:

- *Power* $\times$ *Timing*:
  - Construct a power prediction model but measure execution time explicitly from the execution of the application. The energy consumption is obtained by computing their product. The power prediction model can give a single number, which is the average dynamic power consumption of the application execution. It can also provide a time-series graph of dynamic power for the application execution. The energy consumption can then be calculated by using trapezoidal rule.
  - Construct power prediction and timing prediction models separately and compose the energy consumption from these models. The prediction error in accuracy of energy consumption is compounded by the errors in accuracies of its constituent models (power and timing).
- *Explicit*: By explicitly modelling the energy consumption without any reference to power consumption.

[33] study and compare five full-system real-time power models using a variety of machines and benchmarks. Four of these models are utilization-based whereas the fifth includes CPU performance monitoring counters (PMCs) in the model parameter set along with the utilizations of CPU and disk. They report that PMC-based model is the best overall in terms of accuracy since it is able to account for majority of the contributors to system's dynamic power (especially the memory activity). They also question the generality of their PMC-based model since the PMCs used in their model parameter set may not have the same essence across different architectures (Intel, AMD). [34] evaluate the competence of predictive power models for modern node architectures and show that linear regression models show prediction errors as high as 150%. They suggest that direct physical measurement of power consumption should be the preferred approach to tackle the inherent complexities posed by modern node architectures. [35] survey predictive power and energy models focusing on the highly heterogeneous and hierarchical node architecture in modern HPC computing platforms. They highlight the ineffectiveness of models to accurately predict the dynamic power consumption of modern nodes due to the inherent complexities (contention for shared resources such as Last Level Cache (LLC), NUMA, and dynamic power management).

In this paper, we use a functional model of dynamic energy consumption where the dynamic energy consumption is represented by a function of the problem size. This functional model is input to energy optimization problem and the corresponding algorithm solving the problem. For our experimental results, we use a functional model built from experimental points where each point is obtained by direct measurement of the dynamic energy consumption during the execution of the application using a Watts Up power meter.

3 Formulations of Performance Optimization and Energy Optimization Problems

Consider a workload of size n executed using one or more of p identical processors. Let the speed function of a processor executing a problem size x be represented by $s(x)$. The speed $s(x)$ for a problem size x is calculated as $\frac{x}{t(x)}$, where $t(x)$ is the time of execution of the problem size. The dynamic energy consumption of execution of a problem size x by a processor is represented by $\Omega(x)$. There are two optimization problems to consider.

Performance Optimization Problem, *POPT*(n, p, s, q, d): The problem is to find a partitioning, $d = \{x_1, ..., x_q\}$, of the workload of size n between p identical processors that minimizes the computation time of parallel execution of the workload. The parameters (n, p, s) and the parameters (q,d) are the inputs and outputs respectively of the problem. This problem can be formulated as an integer non-linear program (INLP) as follows:

$$\begin{aligned}
& minimize \max_{i=1}^{q} \frac{x_i}{s(x_i)} \\
\text{Subject to}\quad & x_1 + x_2 + ... + x_q = n \\
& x_i \le n \qquad i = 1, ..., q \\
& x_i \ge 0 \qquad i = 1, ..., q \\
& 1 \le q \le p \\
\text{where}\quad & p, q, n, x_i \in \mathbb{Z}_{>0} \quad \text{and} \quad s(x) \in \mathbb{R}_{>0}
\end{aligned}$$

This INLP problem can be modified to an equivalent integer linear program (ILP) problem as follows:

$$\begin{aligned}
& minimize f \\
\text{Subject to}\quad & f \ge \frac{x_i}{s(x_i)} \quad i = 1, ..., q \\
& x_1 + x_2 + ... + x_q = n \\
& x_i \le n \qquad i = 1, ..., q \\
& x_i \ge 0 \qquad i = 1, ..., q \\
& 1 \le q \le p \\
\text{where}\quad & p, q, n, x_i \in \mathbb{Z}_{>0} \quad \text{and} \quad s(x) \in \mathbb{R}_{>0}
\end{aligned}$$

We propose two formulations for the optimization problem for energy. The first formulation is based on a energy model, which represents the dynamic energy consumption of a processor by a function of problem size. For example, Fig. 6 and 12 respectively show the experimentally built energy functional models of OpenBLAS DGEMM and FFTW applications each executing 24 threads on the Intel Haswell server (Table 1). For each problem size in the energy consumption graph, the energy consumption was measured using Watts Up Pro power meter.

Energy Optimization Problem, *EOPT*(n, p, Ω, q, d): The problem is to find a partitioning, $d = \{x_1, ..., x_q\}$, of the workload of size n between p identical processors that minimizes the dynamic energy consumption of computations in the parallel execution of the workload. The parameters (n, p, Ω) and the parameters (q,d) are the inputs and outputs respectively of the problem. This problem can be formulated as an integer linear program (ILP) as follows:

$$\begin{aligned}
& minimize \sum_{i=1}^{q} \Omega(x_i) \\
\text{Subject to}\quad & x_1 + x_2 + ... + x_q = n \\
& x_i \le n \qquad i = 1, ..., q \\
& x_i \ge 0 \qquad i = 1, ..., q \\
& 1 \le q \le p \\
\text{where}\quad & p, q, n, x_i \in \mathbb{Z}_{>0} \quad \text{and} \quad \Omega(x) \in \mathbb{R}_{>0}
\end{aligned}$$

The second formulation requires a functional performance model and a functional power model, which represents the average dynamic power consumption of a processor by a function of problem size. For example, Fig. 5 and 11 respectively show the experimentally built power functional models of OpenBLAS DGEMM and FFTW applications each executing 24 threads on the Intel Haswell server (Table 1). For each problem size in the power consumption graph, the average dynamic power consumption was calculated from the power measurements obtained using Watts Up Pro power meter during the execution of the problem size.

In this formulation, let the average dynamic power consumption of a processor executing a workload x be $P_d(x)$. The dynamic energy consumption $\Omega(x)$ of execution of the workload is proportional to $P_d(x) \times \frac{x}{s(x)}$. The EOPT problem can therefore be reformulated as follows:

$$EOPT2(n, p, s, P_d, q, d):$$

$$minimize \sum_{i=1}^{q} P_d(x_i) \times \frac{x_i}{s(x_i)}$$

$$\begin{aligned} \text{Subject to} \quad & x_1 + x_2 + ... + x_q = n \\ & x_i \leq n \qquad i = 1, ..., q \\ & x_i \geq 0 \qquad i = 1, ..., q \\ & 1 \leq q \leq p \end{aligned}$$

$$\text{where } p, q, P_d(x), n, x_i \in \mathbb{Z}_{>0} \text{ and } s(x) \in \mathbb{R}_{>0}$$

The functional models for power and energy for an application are built for a wide range of problem sizes where for each problem size, the average dynamic power consumption or the dynamic energy consumption can be determined via either a direct measurement or a prediction model.

POPT and EOPT (EOPT2) are popularly known as *min-max* and *min-sum* problems.

4 POPTA: Algorithm Solving POPT Problem

In this section, we present an efficient algorithm solving the POPT problem where the speed function of a processor, $s(x)$, is represented by a discrete set of experimental points separated by minimum granularity. The problem can be described as follows: Let p identical processors be used to execute the workload of size n, and let $s(x)$ be the speed of execution of the workload of size x by a processor. Let Δx be the minimum granularity of workload so that each processor is allocated a multiple of Δx only. The problem is to find the distribution of the workload of size n between the p processors, which minimizes the computation time of its parallel execution.

We propose an algorithm called POPTA (Algorithm 1) that solves the problem. It should be noted that the traditional load-balancing algorithm returns the workload distribution, $x_i = \frac{n}{p}, \forall i \in [1, p]$.

The inputs to the algorithm are the size of the workload, n, given as multiple of Δx, the number of processors, p, the minimum granularity, Δx, and the speed function represented by two discrete sets, X and S respectively containing problem sizes and speeds. m is the cardinality of the sets X and S. The outputs from the algorithm are the optimal workload distribution, D_{opt}, where the distributions are given in multiples of Δx, and the optimal execution time, t_{opt}. It is important to note that the optimal number of processors that are selected by POPTA in the optimal workload distribution may be less than p.

We will illustrate its execution through an example. Consider $p = 4$ processors involved in parallel execution of a OpenBLAS DGEMM workload of size $n = 64$. Let the minimum granularity Δx be 1. We use a segment of the speed function, $s(x)$, shown in Figure 14. The function is represented by discrete sets X and S composed from the points in the graph. These points are connected by dashed

Algorithm 1 Algorithm determining optimal distribution of workload of size n for maximizing performance.

```
1: procedure POPTA($n, p, \Delta x, X, S, D_{opt}, t_{opt}$)
Input:
    Workload size, $n \in \mathbb{Z}_{>0}$
    Number of processors, $p \in \mathbb{Z}_{>0}$
    Minimum granularity, $\Delta x \in \mathbb{Z}_{>0}$
    Speed function represented by two sets $(X, S)$,
    $X = \{x_1, ..., x_m\}, x_1 < ... < x_m, x_i \in \mathbb{Z}_{>0}, \forall i \in [1, m]$
    $S = \{s(x_1), ..., s(x_m)\}, s(x) \in \mathbb{R}_{>0}$
Output:
    Optimal workload distribution,
    $D_{opt} = \{x^1_{opt}, ..., x^p_{opt}\}, x^i_{opt} \in \mathbb{Z}_{>0}, \forall i \in [1, p]$
    Optimal execution time, $t_{opt} \in \mathbb{R}_{>0}$

2:     for $point \leftarrow 1, m$ do
3:         $(X_\uparrow[point], S_\uparrow[point]) \leftarrow$ Sort↑($point$, $X$, $S$)
4:     end for
5:     $(B, E) \leftarrow$ GetBE($n$,$p$,$\Delta x$,$X$,$S$)
6:     if $n \bmod p = 0$ then
7:         if $E \leq \frac{X_\uparrow[\frac{n}{p}]}{S_\uparrow[\frac{n}{p}]}$ then
8:             $D^i_{opt} \leftarrow \frac{n}{p}, \forall i \in [1, p]$; $t_{opt} \leftarrow \frac{\frac{n}{p} \times \Delta x}{S[\frac{n}{p}]}$
9:             return $(D_{opt}, t_{opt})$
10:        end if
11:    end if
12:    $\forall I \in [1, \frac{n}{p}], J \in [1, p], K \in [1, J]$,
13:        $memorized[I][J][K] \leftarrow (0, 0, 0)$
14:    $(D_{opt}, t_{opt}) \leftarrow$ POPTAKernel(
           $n$,$p$,$\Delta x$,$B$,$E$,$X$,$S$,$X_\uparrow$,$S_\uparrow$,$memorized$)
15:    return $(D_{opt}, t_{opt})$
16: end procedure
```

lines for clarity. The recursive procedure, $POPTAKernel$ (Algorithm 2), examines all the points between the lines B and E as shown in Figure 15. Vertical line B represents $x = \frac{n}{p}$ ($x = 16$ in this example) and line E passes through origin and the point $(\frac{n}{p}, \frac{\frac{n}{p}}{s(\frac{n}{p})})$.

The first step of the algorithm is to create a sorted array of points for each point $a \in [1, m]$ (Lines 2-4). For each point a, the array contains all the points sorted in non-decreasing order of $\frac{X[b]}{S[b]}, \forall b \in [a + 1, m]$. The ratio $\frac{x}{s(x)}$ is proportional to the execution time of the problem size x. The sorted arrays are stored in the arrays, $\{X_\uparrow, S_\uparrow\}$. If the point a has execution time less than or equal to execution times at points greater than it ($x > a$), then the point a represents the optimal workload distribution for workload of size $p \times a$ using p processors. That is, the problem size a is allocated to all the p processors. Therefore, having these sorted arrays of points allows us to avoid recursion of the algorithm at points, which give optimal workload distribution for subproblems. These are essentially points which have execution times less than or equal to execution times at points beyond them.

The procedure, *GetBE* (given in Section 2 of the supplemental), determines the lines B and E and takes into account the case when n is not divisible by p. When n is divisible by p, B is $\frac{n}{p}$ and E is the execution time to solve it, $\frac{n}{p}/s(\frac{n}{p})$. When n is not divisible by p, there are many

Algorithm 2 The kernel of the algorithm 1.

```
1:  function POPTAKernel($n, p, \Delta x, B, E,$
        $X, S, X_\uparrow, S_\uparrow, memorized, D_{opt}, t_{opt}$)
2:    if $p = 1$ then return ($\{n\}, \frac{n\times\Delta x}{S[n]}$) end if
3:    $D^i_{opt} \leftarrow \frac{n}{p}, \forall i \in [1,p]$
4:    $D^i_{opt} \leftarrow D^i_{opt} + 1, \forall i \in [1, n \bmod p]$
5:    $t_{opt} \leftarrow \max_{1\le i\le p}(\frac{D^i_{opt}}{S[D^i_{opt}]})$
6:    for $L \leftarrow memorized[B][p][n \bmod p][1], |X_\uparrow|$ do
7:      $n_r \leftarrow X_\uparrow[L]$
8:      $t_r \leftarrow \frac{n_r}{S_\uparrow[L]}$
9:      if $t_r \ge E$ then break end if
10:     for $r \leftarrow 1, p-1$ do
11:       $n_l \leftarrow n - r \times n_r$
12:       if ($n_l < 0$) then break end if
13:       if $n_l = 0$ and $t_r < t_{opt}$ then
14:         $d^i_{opt} \leftarrow n_r, \forall i \in [1,r]$
15:         $d^i_{opt} \leftarrow 0, \forall i \in [r+1,p]$
16:         $t_{opt} \leftarrow t_r$
17:         continue
18:       end if
19:       $(B_l, E_l) \leftarrow$ GetBE($n_l$,$p-r$,$\Delta x$,$X$,$S$)
20:       if $B_l > |X|$ then continue end if
21:       if ($n_l \bmod (p-r) \ne 0$) or ($E_l > \frac{X_\uparrow[B_l]}{S_\uparrow[B_l]}$) then
22:         $E_l \leftarrow (E_l > t_r)\ ?\ t_r : E_l$
23:         $t_l \leftarrow memorized[\frac{n_l}{p-r}][p-r][n \bmod p][3]$
24:         if ($t_l \le E_l$ and $max(t_r, t_l) < t_{opt}$) then
25:           $\forall i \in [r+1,p], x_i \leftarrow$
26:           $memorized[\frac{n_l}{p-r}][p-r][n \bmod p][2]$
27:         else
28:           $\{(x_{r+1}, ..., x_p), t_l\} \leftarrow$
                POPTAKernel(
                $n_l$,$p-r$,$\Delta x$,$B_l$,$E_l$,
                $X$,$S$,$X_\uparrow$,$S_\uparrow$,$memorized$,$D_{opt}$,$t_{opt}$)
29:         end if
30:       else
31:         $t_l \leftarrow E_l$
32:         if $max(t_r, t_l) < t_{opt}$ then
33:           $x_i \leftarrow B_l, \forall i \in [r+1,p]$
34:         end if
35:       end if
36:       if $max(t_r, t_l) < t_{opt}$ then
37:         $d^i_{opt} \leftarrow n_r, \forall i \in [1,r]$
38:         $d^i_{opt} \leftarrow x_i, \forall i \in [r+1,p]$
39:         $t_{opt} \leftarrow max(t_r, t_l)$
40:         if $t_l \le t_r$ then Go To 43 end if
41:       end if
42:     end for
43:   end for
44:   $memorized[\frac{B}{\Delta x}][p][n \bmod p] \leftarrow (L, d_{opt}, t_{opt})$
45:   return ($D_{opt}, t_{opt}$)
46: end function
```

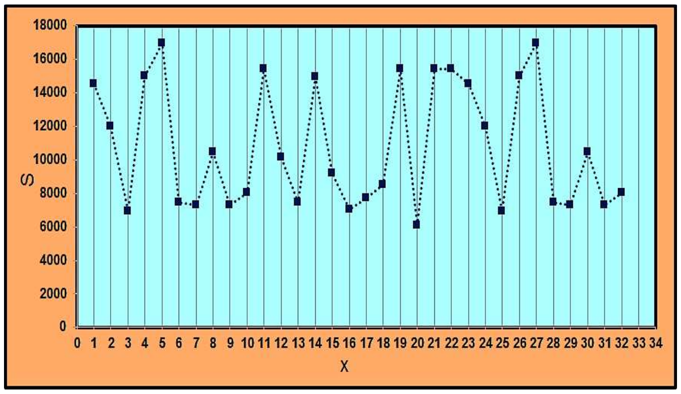


Fig. 14. POPTA example: Speed function of a processor executing the multithreaded OpenBLAS DGEMM application represented by a discrete set of points (connected by dashed lines for clarity).

ways to allocate the extra $n \bmod p$ units. The procedure just picks one combination for the purpose of initialization. Also when n is divisible by p and if the point $(\frac{n}{p}, \frac{n}{p}/s(\frac{n}{p}))$ has execution time less than or equal to execution times at points greater than it ($x > \frac{n}{p}$), then we have the optimal workload distribution, that is, the problem size $\frac{n}{p}$ is allocated to all the p processors. This is the traditional homogeneous load-balanced workload distribution.

A key optimization in the algorithm is the 3D array, $memorized$, of size $O(m \times p^2)$, which memorizes the points that have already been visited during the recursive invocations. This array is initialized to zero before the invocation of the core routine (Algorithm 2). Briefly, for the execution of the problem size n using p processors, the array value $memorized[\frac{n}{p}][p][extra]$ contains the ending index of the range of points examined during the previous invocation. The array entry $memorized[\frac{n}{p}][p]$ is of size p where the $extra$ index represents a problem size ($\frac{n}{p} + n \bmod p$) in the range $[\frac{n}{p}, \frac{n}{p} + p]$. This memorization ensures that there are only $O(m \times p^2)$ recursive invocations of the core kernel (Algorithm 2) to solve a problem size of n using p processors. Along with memorization of the range of points examined, the optimal workload distribution and the optimal execution time are also memorized.

One other optimization (Line 40) is that during the recursive invocation of POPTA (Algorithm 2) for some r (Line 11), if t_l is less than or equal to t_r, then we return from the invocation because we have found the optimal solution for the problem size n using p processors. This is because any other solution will have an execution time greater than or equal to t_r since points to the right of $B = \frac{n_l}{p-r}$ are sorted in non-decreasing order of execution times (as given by the arrays $\{X_\uparrow, S_\uparrow\}$). This is the best case.

Line 2 of the procedure, *POPTA*, deals with the simple case of solving the problem size n using one processor. Lines 3-5 initialize the outputs, D_{opt} and t_{opt}, allocating each extra bit Δx to all the processors, $I \in [1, n \bmod p]$. Lines 6-45 contain the kernel of *POPTA*. The array of sorted points between B and E as shown on lines $L_1, ..., L_q$ in Figure 16 is sequentially examined (Line 6-8). The condition ($t_r > E$) ensures that all points beyond E are not considered. For each point A (on line $L_x, \forall x \in [1,q]$), there are $p-1$ main execution steps in the nested for loop (at Line 10). In a main

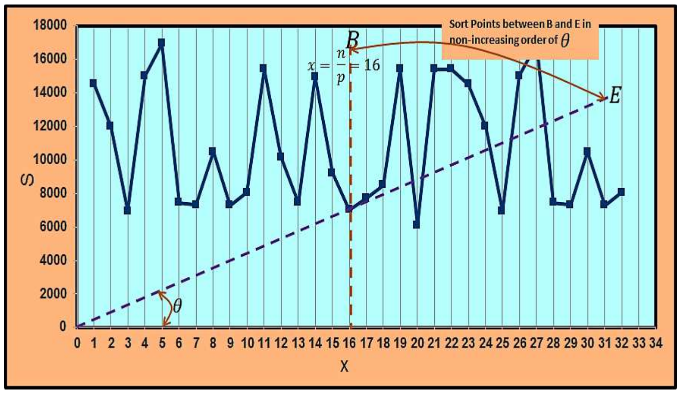


Fig. 15. POPTA example: POPTA sorts points between B and E in non-increasing order of Θ.

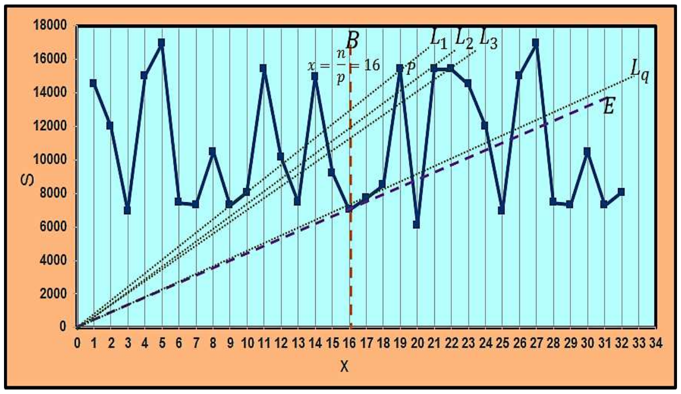


Fig. 16. POPTA example: Points on line L_1 examined followed by points on line L_2 and so on until the points on line L_q.

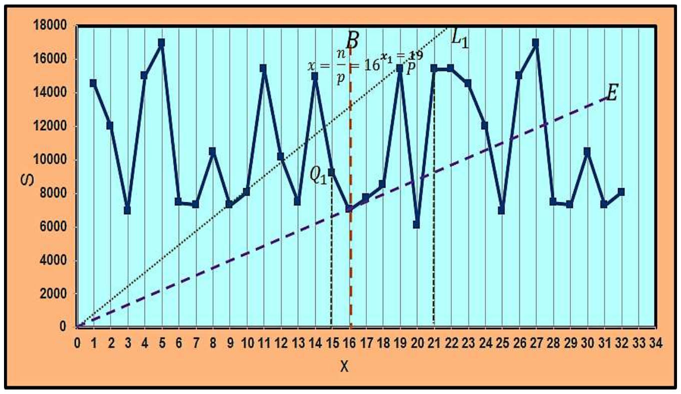


Fig. 17. POPTA example: One processor is allocated problem size 19 to the right. POPTA is invoked for remaining problem size 45 and 3 remaining processors.

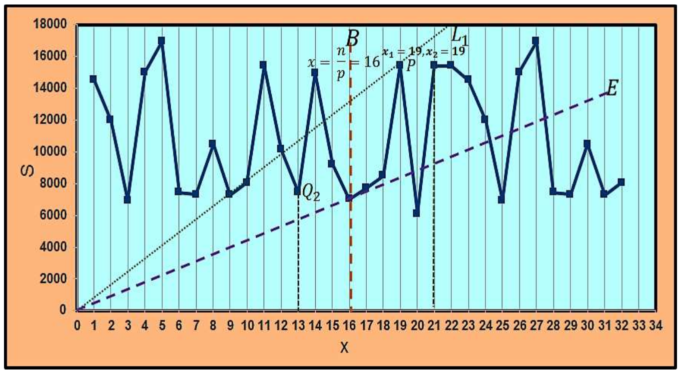


Fig. 18. POPTA example: Two processors are allocated problem size $P = 19$ each to the right. $POPTAkernel$ is invoked to find optimal load distribution for remaining problem size 26 and 2 remaining processors.

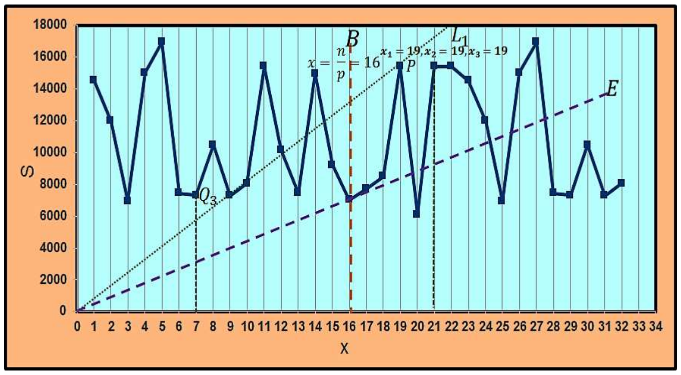


Fig. 19. POPTA example: Three processors are allocated problem size $P = 19$ each to the right. The only remaining processor is allocated problem size $Q3 = 7$.

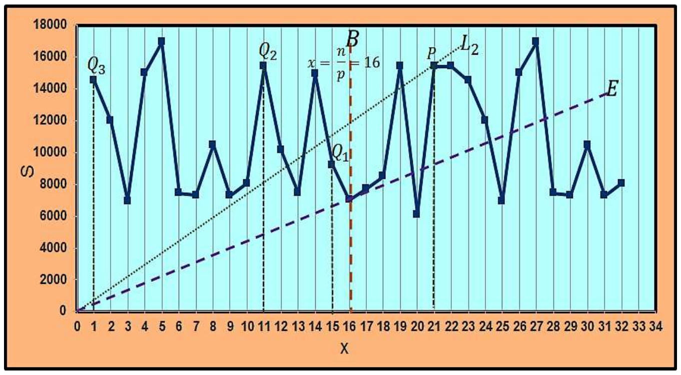


Fig. 20. POPTA example: Point P on line L_2 examined. The corresponding allocations to the left are Q_1, Q_2, and Q_3.

step, each of the r processors is allocated the problem size n_r to the right of B. If the remaining problem size n_l is less than 0, that means there is excessive allocation to the right of B and so we break from the loop since subsequent allocations to the right of B will always result in negative remaining problem size to the left of B. If the remaining problem size n_l is equal to 0, then we save this distribution if ($t_r < t_{opt}$) (Lines 14-16).

Then, we determine the lines B_l and E_l for the recursive invocation of $POPTA$ solving the problem size n_l to the left of B using $p - r$ processors using the function, $getBE$ (Line 19). We invoke $POPTA$ to solve the problem size n_l to the left of B using $p - r$ processors only if n_l is not divisible by $p - r$ and the execution time of the point B_l given by E_l in the recursive invocation is greater than the execution times of the points beyond it (Line 21). This can be determined using the sorted arrays, $(X_{\uparrow}, S_{\uparrow})$. Otherwise, the optimal workload distribution is given by the point B_l for the recursive invocation (Lines 32-34). If the memorized execution time of the recursive invocation (t_l) is less than or equal to E_l, then we use the memorized workload

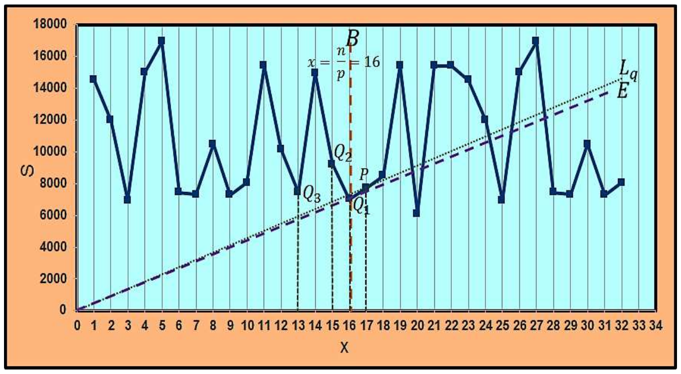


Fig. 21. POPTA example: Point P on the final line L_q examined. The corresponding allocations to the left are Q_1, Q_2, and Q_3.

distribution and avoid recursion (Line 26). Essentially, if a range of points have already been examined, then they will not be re-examined due to the memorization. For the recursive invocation solving the problem size n_l to the left of P using $p-r$ processors, the lines B and E are set in B_l and E_l. E_l is either L_x or $\frac{B_l}{s(B_l)}$, whichever is lesser. If $\frac{B_l}{s(B_l)}$ is less than L_x, then we don't consider points beyond $\frac{B_l}{s(B_l)}$ (i.e., greater than $\frac{B_l}{s(B_l)}$ but less than or equal to L_x) because those points will have worse execution times. That is, when the algorithm is considering the points on a line L_x, this line will always be the limiting line for the recursive invocations.

For a main step, if the execution time of the parallel execution $(\max(t_r, t_l))$ is less than the t_{opt}, we save the improved solution (Lines 37-39). For each problem size n_l solved using $p-r$ processors, the ending index L, which contains the range of points already examined, is saved (Line 44). So, if an invocation for solving this problem size recurs, then recursion is avoided using the memorized arrays (Line 26). Therefore, this memorization ensures that the total number of examined points (including those in the recursive invocations) for a point on a line $Lx, \forall x \in [1, q]$ is not more than $O(m \times p^2)$.

Let us trace the execution of the procedure for the only point P on line L_1. There are 3 main execution steps for this point (for loop in Line 10). In the first step, one processor is allocated the problem size $n_r = 19$ to the right shown by point P in Figure 17. *POPTAKernel* is now invoked to find the optimal workload distribution for problem size $n_l = 45$ and $p-r = 3$ processors. The point Q_1 shown in Figure 17 represents $x = \frac{n}{p} = 15$ for this problem size in the recursive invocation $POPTAKernel(45, 3, 1, Q_1, L_1, X, S, ...)$. The lines B and E for this recursive invocation are set to Q_1 and L_1 respectively. In the second step, 2 processors are allocated the problem size $n_r = 19$ to the right shown by point P in Figure 18. *POPTAKernel* is now invoked to find the optimal workload distribution for problem size $n_l = 26$ and $p-r = 2$ processors. The point Q_2 shown in Figure 18 represents $x = \frac{n}{p} = 26$ for this problem size in the recursive invocation $POPTAKernel(26, 2, 1, Q_2, L_1, X, S, ...)$. The lines B and E for this recursive invocation are set to Q_2 and L_1 respectively. Similarly, for the third step, 3 processors are allocated the problem size $n_r = 19$ to the right shown by point P and one processor is allocated the problem size $n_l = 7$ to the left shown by point Q_3 in Figure 19. The best workload distribution and execution time from the execution of these three steps is saved in D_{opt} and t_{opt}.

After examining all the points on L_1, the algorithm considers the points on Line L_2. There is only one point P on this line as shown in Figure 20. For this point, the points Q_1, Q_2, and Q_3 respectively represent the recursive $POPTAkernel$ invocations to the left of B for $r = 1, r = 2$, and $r = 3$. So, in this manner, the algorithm examines the points on lines L_1, L_2, L_3, and so on until the final line L_q (shown in Figure 21) before (and excluding) E.

At the end of the execution of the algorithm, the optimal workload distribution is returned in d_{Opt} and the optimal execution time is returned in e_{Opt}.

The optimality and complexity proofs of POPTA are presented in Sections 3 and 4 respectively in the supplemental material.

5 EOPTA: Algorithm Solving EOPT Problem

In this section, we present an efficient algorithm solving the EOPT problem where the energy function of a processor, $\Omega(x)$, is represented by a discrete set of experimental points separated by minimum granularity. The problem can be described as follows: Let p identical processors be used to execute the workload of size n, and let $\Omega(x)$ be the dynamic energy consumption of execution of the workload of size x by a processor. Let Δx be the minimum granularity of workload so that each processor is allocated a multiple of Δx only. The problem is to find the distribution of the workload of size n between the p processors, which minimizes the total dynamic energy consumption of the computations in its parallel execution.

We propose an algorithm called EOPTA (Algorithm 3) that solves the problem. It has structure similar to algorithm POPTA (Algorithm 1).

The inputs to the algorithm are size of the workload, n, given as multiple of Δx, the number of processors, p, the minimum granularity, Δx, and the dynamic energy function represented by two discrete sets, X and Ψ respectively containing problem sizes and dynamic energy consumptions. m is the cardinality of the sets X and Ψ. The outputs from the algorithm are the optimal load distribution, D_{opt}, and the optimal dynamic energy consumption, Ω_{opt}. It is important to note that the number of processors that are selected by POPTA in the optimal workload distribution may be less than p. For example, if the dynamic energy function is concave, then EOPTA may select just one processor to execute the workload if the workload size lies in the domain of the dynamic energy function.

The first step of the algorithm is to determine the set of convex points, $ConvS$, using the function, $getConvexSet$ (given in Section 5 in the supplemental). A point Q is defined as convex if $\Psi[I_Q - k] + \Psi[I_Q + k] > 2 \times \Psi[I_Q], \forall k \in [1, 2 \times I_Q]$, where I_Q is the index of point Q. For example, Figure 22 shows these points in a segment of the energy function of OpenBLAS DGEMM application. If all the points in Ψ are convex, then we determine the optimal workload distribution using Lemma 6.1 given in the supplemental. If there are no convex points in Ψ (i.e., $ConvS = \Phi$,

Algorithm 3 Algorithm determining optimal distribution of workload of size n for minimizing energy.

```
1: procedure EOPTA($n, p, \Delta x, X, \Psi, D_{opt}, t_{opt}$)
Input:
    Workload size, $n \in \mathbb{Z}_{>0}$
    Number of processors, $p \in \mathbb{Z}_{>0}$
    Minimum granularity, $\Delta x \in \mathbb{Z}_{>0}$
    Energy function represented by two sets $(X, \Psi)$,
    $X = \{x_1, ..., x_m\}, x_1 < ... < x_m, x_i \in \mathbb{Z}_{>0}, \forall i \in [1, m]$
    $\Psi = \{\Omega(x_1), ..., \Omega(x_m)\}, \Omega(x) \in \mathbb{R}_{>0}$
Output:
    Optimal workload distribution,
    $D_{opt} = \{x^1_{opt}, ..., x^p_{opt}\}, x^i_{opt} \in \mathbb{Z}_{>0}, \forall i \in [1, p]$
    Optimal dynamic energy consumption, $\Omega_{opt} \in \mathbb{R}_{>0}$

2:     $ConvS \leftarrow$ GETCONVEXSET($\Psi$)
3:     $ConvConvS \leftarrow$ GETCONVEXSET($ConvS$)
4:     if ($|ConvS| = m$) or
         ($|ConvConvS| = |ConvS|$) then
5:         return ($d_{Opt}, e_{Opt}$)   ▷ Lemma 6.1 (Supplemental)
6:     end if
7:     if ($|ConvS| = 2$) or ($|ConvConvS| = 2$) then
8:         return ($d_{Opt}, e_{Opt}$)   ▷ Lemma 7.1 (Supplemental)
9:     end if
10:    $\forall I \in [1, \frac{n}{p} + 1], J \in [1, p], K \in [1, p]$
11:        $memorized[I][J][K] \leftarrow (I + 1, 0, 0)$
12:    $(D_{opt}, \Omega_{opt}) \leftarrow$ EOPTAKERNEL(
           $n, p, \Delta x, |ConvS|, X, \Psi, ConvS, memorized$)
13:    return ($D_{opt}, t_{opt}$)
14: end procedure
```

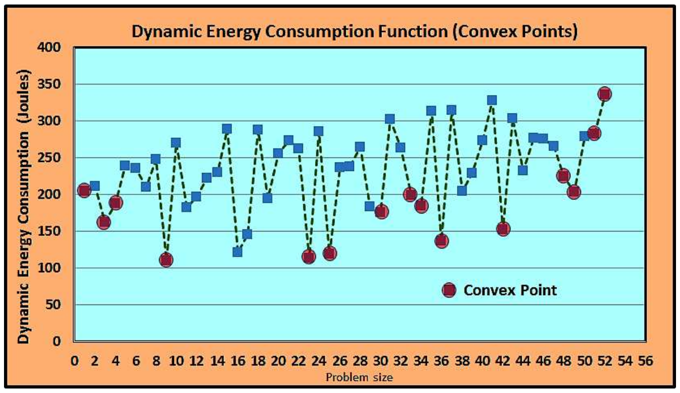


Fig. 22. Convex points shown for a segment of dynamic energy consumption graph of OpenBLAS DGEMM application.

excluding the endpoints), then the dynamic energy consumption function is a concave function and we determine the optimal workload distribution using Lemma 7.1 given in the supplemental. Also, if the set of convex points, $ConvS$, is a convex function or a concave function, then we use Lemmas 6.1 and 7.1 (given in the supplemental) respectively to determine the optimal workload distribution. These are the best cases. One can see from the Figure 22 that the set of convex points is neither a convex function nor a concave function. So, EOPTA will examine all the convex points in the energy function in this case.

Similar to POPTA, a 3D array, $memorized$, of size

Algorithm 4 The kernel of algorithm EOPTA 3.

```
1: function EOPTAKERNEL($n, p, \Delta x, E,$
           $X, \Psi, ConvS, memorized, D_{opt}, t_{opt}$)
2:     if ($p = 1$) then return ($\{n\}, \Psi[n]$) end if
3:     $D^i_{opt} \leftarrow \frac{n}{p}, \forall i \in [1, p]$
4:     $D^i_{opt} \leftarrow D^i_{opt} + 1, \forall i \in [1, n \bmod p]$
5:     $\Omega_{opt} \leftarrow \sum_{proc=1}^{p} \Psi[D^{proc}_{opt}]$
6:     for $L \leftarrow memorized[nbyp][p][extra][1], E$ do
7:         for $r \leftarrow 1, p - 1$ do
8:             $n_r \leftarrow X[Conv[L]]$; $n_l \leftarrow n - r \times n_r$
9:             if $n_l < 0$ and $\Omega_r < \Omega_{opt}$ then
10:                $d^1_{opt} \leftarrow n$; $d^i_{opt} \leftarrow 0, \forall i \in [2, p]$;$\Omega_{opt} \leftarrow \Omega_r$
11:                break
12:            end if
13:            $\Omega_r \leftarrow r \times \Psi[Conv[L]]$
14:            if $n_l = 0$ and $\Omega_r < \Omega_{opt}$ then
15:                $d^i_{opt} \leftarrow n_r, \forall i \in [1, r]$;
16:                $d^i_{opt} \leftarrow 0, \forall i \in [r + 1, p]$; $\Omega_{opt} \leftarrow \Omega_r$
17:                continue
18:            end if
19:            if $memorized[\frac{n_l/\Delta x}{p-r}][p - r][extra] > L$ then
20:                $\Omega_l \leftarrow memorized[\frac{n_l/\Delta x}{p-r}][p - r][extra][3]$
21:                if $\Omega_r + \Omega_l < \Omega_{opt}$ then
22:                    $\forall i \in [r + 1, p], x_i \leftarrow$
23:                    $memorized[\frac{n_l/\Delta x}{p-r}][p - r][extra][2]$
24:                end if
25:            else$\{(x_{r+1}, ..., x_p), \Omega_l\} \leftarrow$ EOPTAKERNEL(
                   $n_l, p-r, \Delta x, L, X, \Psi, ConvS, memorized$)
26:            end if
27:            if $(\Omega_r + \Omega_l) < \Omega_{opt}$ then
28:                $d^i_{opt} \leftarrow n_r, \forall i \in [1, r]$
29:                $d^i_{opt} \leftarrow x_i, \forall i \in [r+1, p]$;$\Omega_{opt} \leftarrow (\Omega_r + \Omega_l)$
30:            end if
31:        end for
32:    end for
33:    $memorized[\frac{n}{p}][p][extra] \leftarrow (L, d_{opt}, \Omega_{opt})$
34:    return ($D_{opt}, \Omega_{opt}$)
35: end function
```

$O(m \times p^2)$ is used to memorize the points that have already been examined during the recursive invocations. Note that each array entry $memorized[I][J[K]$ is initialized to $I + 1$ because the range of points examined in the loop (Line 6, Algorithm 4) is $[I + 1, E]$. While in POPTA, the space of points considered on the right lies between $B = \frac{n}{p}$ and E, the space of points in EOPTA are all the points in the convex set, $ConvS$.

The optimality and complexity proofs of EOPTA are presented in Sections 8 and 9 respectively in the supplemental.

6 Experimental Analysis of POPTA and EOPTA

In this section, we will present analysis of POPTA and EOPTA for two data-parallel applications, OpenBLAS DGEMM [8] and FFTW [9], [36], on a Intel Haswell server shown in Table 1. The experiments are a combination of actual measurements conducted on the server and simulations for clusters containing replicas of the server. We would like to mention that the results from actual experiments on clusters would be no different from the results of our simulations. This is because the algorithms minimize the execution time and energy consumption of computations only in the parallel execution of a workload and do not consider communication overheads. Since the problems sizes ($x_i, \forall i \in [1, p]$) in the optimal workload distributions are members of the set X in the input functions that have been built experimentally, the minimal execution time and energy consumption from actual measurements can not differ from simulations.

The speed and the energy functions that are input to POPTA and EOPTA are experimentally built on the Intel Haswell server. These are built once and for only one processor (the Intel Haswell server in this case). Simulations of POPTA and EOPTA for clusters containing replicas of the Intel Haswell server are then also executed on this server. The speed (X, S) and dynamic energy consumption (X, Ψ) functions of OpenBLAS DGEMM application are shown in Fig. 7 and 9 respectively. The speed of the application multiplying two matrices of size $n \times n$ (and corresponding problem size of n^2) is calculated as $(2 \times n^3)/t$, where t is the execution time. The speed (X, S) and dynamic energy consumption (X, Ψ) functions of FFTW application are shown in Fig. 10 and 12 respectively. The speed of the application performing 2D DFT of size $n \times n$ (and corresponding problem size n^2) is calculated as $(2.5 \times n \times \log_2 n)/t$, where t is the execution time. The total dynamic energy consumption during the application execution is obtained using Watts Up Pro power meter. To ensure reliability of the results, we follow a detailed methodology, which is described in Section 1 of the supplemental. We execute the application repeatedly and stop the measurements only when the sample mean lies in the interval with the confidence level 95% and a precision of 2.5% is achieved. In this work, we use Student's t-test, assuming that the individual observations are independent and their population follows the normal distribution.

The cardinality (m) of the discrete sets ((X, S) and (X, Ψ)) representing the speed function and dynamic energy function of OpenBLAS DGEMM application is 1440. The granularity (Δx) selected for the OpenBLAS DGEMM application is 1478656 (representing DGEMM of a 1216×1216 matrix). The cardinality (m) of the discrete sets representing the speed function and dynamic energy function of FFTW application is 766. The granularity (Δx) selected for the FFTW application is 1327104 (representing 2D DFT of size 1152×1152). There is no specific reason why we picked the particular granularities for OpenBLAS DGEMM and FFTW. Lesser granularity would unveil larger fluctuations in the functional models but would also mean more experimental points thereby increasing the time to build the functional models as well as the execution times of POPTA and EOPTA. As the granularity increases, the functional models become smooth and will resemble those for single-core CPUs therefore disallowing any opportunity for optimization. There are 44 and 59 convex points in the dynamic energy consumption graphs for OpenBLAS DGEMM and FFTW (Fig. 9 and 12 respectively). The starting point in the speed and energy functions contains a problem size that is so selected as to exceed the L2 cache. The last point in the discrete set X (representing DGEMM of a 46080×46080 matrix) for the OpenBLAS DGEMM application contains a problem size, which occupies the whole main memory. For the FFTW application, we restrict the number of points because we observed that very large problem sizes were taking erroneously large execution times to complete possibly due to a software limitation. This was unduly lengthening the experimental building times of the functional models. The last point contains the problem size representing 2D DFT of size 32768×32768.

To demonstrate the benefits of the solutions determined by POPTA and EOPTA, we report the average and maximum improvements in performance and energy provided by the solutions determined by these algorithms. The performance improvement is calculated as follows: Performance Improvement (%) $= \frac{t_{homo} - t_{popta}}{t_{popta}} \times 100$, where t_{homo} is the execution time obtained using traditional homogeneous workload distribution ($x_i = \frac{n}{p}, \forall i \in [1, p]$) and t_{popta} is the execution time obtained using the optimal workload distribution determined by POPTA. The percentage energy reduction is calculated as follows: Energy reduction (%) $= \frac{\Omega_{homo} - \Omega_{eopta}}{\Omega_{eopta}} \times 100$, where Ω_{homo} is the dynamic energy consumption using traditional homogeneous workload distribution and Ω_{eopta} is the dynamic energy consumption using the optimal workload distribution determined by EOPTA. Negative values of these percentages represent performance degradation and increase in energy consumption.

In the analysis of the solutions determined by POPTA, we use two different datasets. The first dataset is composed as follows. We sort the speed function (X, S) in non-decreasing order of speeds. We choose the first half of the points from this sorted dataset. The problem size per processor, $\frac{n}{q}$, is iterated over these selected problem sizes and q is iterated from 2 to 1024. We then solve the resulting workloads of size, $n = \frac{n}{q} \times q$, using p processors in the range, $[q, 1024]$. For the second dataset, we use all the points in the speed function to iterate for $\frac{n}{q}$.

To analyze the solutions determined by EOPTA, we again compose two different datasets. For the first dataset, we sort the energy function (X, Ψ) in non-increasing order of energies and pick the first half of the points to iterate for $\frac{n}{q}$. For the second dataset, we use all the points in the energy function to iterate for $\frac{n}{q}$.

The average percentage improvement in performance and energy is calculated by averaging the percentages obtained from solving all the workloads for the corresponding dataset. The maximum percentage improvement is calculated as the maximum of all these percentages.

For the OpenBLAS DGEMM application, the minimum, average, and maximum percentage improvements in performance for the first dataset are found to be 1%, 17%, and 97% respectively. For the second dataset, the average percentage improvement is 13%. The minimum, average, and maximum percentage reductions in energy for the first dataset are 1%, 23% and 71% respectively. For the second dataset, the average percentage reduction is 18%. One can see that the average and maximum improvements are close to the average and maximum width of variations observed in the Fig. 7 and 9 respectively.

For the FFTW application, the minimum, average, and maximum performance percentage improvements for the first dataset are found to be 15%, 60%, and 95% respectively. For the second dataset, the average percentage improvement is 40%. The minimum, average, and maximum percentage reductions in energy for the first dataset are 1%, 40%, and 127% respectively. For the second dataset, the average percentage reduction is 22%. Again one can observe that the average and maximum improvements are close to the average and maximum width of variations observed in the Fig. 10 and 12 respectively.

For points in the two datasets with the highest speeds and the lowest energies, we confirmed experimentally that the optimal solutions determined by POPTA and EOPTA and the traditional load balancing algorithms are the same. For all the executions, the problem sizes in the optimal workload distributions determined by POPTA and EOPTA are members of the set, X, in the input speed and energy functions. We would also like to mention that the number of processors used in the optimal solutions are less than or equal to the input number of processors, p.

Both the algorithms demonstrated average quadratic polynomial complexity of $O(p^2)$. The execution times ranged from few seconds to few minutes. However, unlike EOPTA, POPTA exhibited a practical complexity of $O(1)$ for many executions.

If the problem size per processor $\frac{n}{p}$ is fixed, the percentage improvements in performance and dynamic energy consumption will be constant for any arbitrary number of processors, p. We confirmed experimentally that this is the case.

To study the interplay between performance and energy, we determined the improvements in energy obtained when using the performance optimizing POPTA as well as the improvements in performance obtained when using the energy optimizing EOPTA. We use the same pairs of datasets in the analysis. To determine the reduction in energy, we use the optimal workload distribution determined by POPTA and calculate the dynamic energy consumption associated with this distribution. We then compare this dynamic energy consumption with the dynamic energy consumption obtained using the traditional workload distribution. We observed that optimizing for performance alone can lead to good reduction in dynamic energy consumption.

For OpenBLAS DGEMM, the minimum, average, and maximum percentage reductions in energy for the first dataset were 1%, 24%, and 68% respectively. For the second dataset, the average percentage reduction is 12%. For FFTW, the minimum, average, and maximum percentage reductions in energy for the first dataset were 1%, 29%, and 55% respectively. For the second dataset, the average percentage reduction is 23%.

To analyze the impact of EOPTA on the performance, we take the optimal workload distribution determined by EOPTA and calculate the execution time associated with this distribution. We then compare this execution time with the execution time obtained using the traditional workload distribution. We observed that optimizing for energy alone can cause major performance degradation.

For OpenBLAS DGEMM, the minimum, average, and maximum performance degradations for the first dataset were 88%, 95%, and 100%. For the second dataset, the average performance degradation is 95%. For FFTW, the minimum, average, and maximum performance degradations for the first dataset were close to 99%, 100%, and 100%. For the second dataset, the average performance degradation is close to 100%.

One of our future goals is to understand the causes for these degradations and pursue pareto-optimality study of performance and energy.

The software implementations of the algorithms presented in this paper can be downloaded from the location, https://git.ucd.ie/manumachu/aleph.

7 Conclusion

Homogeneous parallel platforms now are composed of tightly integrated multicore CPUs with highly hierarchical arrangement of cores. This tight integration has resulted in the cores contending for various shared on-chip resources such as Last Level Cache and interconnect leading to resource contention and NUMA. Due to these newly introduced complexities, the performance and energy profiles of real-life scientific applications executing on these platforms are not smooth and may deviate significantly from the shapes that allowed traditional and state-of-the-art load balancing algorithms to find optimal solutions.

In this paper, we proposed novel model-based methods and algorithms for minimization of time and energy of computations for the most general performance and energy profiles of data parallel applications executing on homogeneous multicore clusters. We formulate the performance and energy optimization problems and present efficient algorithms of complexity $O(p^2)$ solving these problems where p is the number of processors. Unlike load balancing algorithms, optimal solutions found by these algorithms may not load-balance an application. Based on our experiments, we demonstrate the optimality of solutions determined by the algorithms for two data-parallel applications. We also demonstrate significant average and maximum percentage improvements in performance and energy for the two applications compared to traditional load-balanced workload distribution. We further show that optimizing for performance alone can lead to significant reduction in energy. However, optimizing for energy alone can cause major degradation in performance.

In our future work, we will study pareto-optimality of performance and energy for applications executing on homogeneous multicore clusters. We would also try to study optimization problems for performance and energy for applications executing on heterogeneous parallel platforms.

Acknowledgments

This publication has emanated from research conducted with the financial support of Science Foundation Ireland (SFI) under Grant Number 14/IA/2474.

References

[1] QPI, "Intel quickpath interconnect," 2008. [Online]. Available: https://en.wikipedia.org/wiki/Intel_QuickPath_Interconnect
[2] AMDHT, "Hypertransport," 2001. [Online]. Available: https://en.wikipedia.org/wiki/HyperTransport

[3] A. L. Lastovetsky and R. Reddy, "Data partitioning with a realistic performance model of networks of heterogeneous computers," in *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*. IEEE, 2004, p. 104.

[4] A. Lastovetsky and R. Reddy, "Data partitioning with a functional performance model of heterogeneous processors," *International Journal of High Performance Computing Applications*, vol. 21, no. 1, pp. 76–90, 2007.

[5] A. Lastovetsky and J. Twamley, "Towards a realistic performance model for networks of heterogeneous computers," in *High Performance Computational Science and Engineering*. Springer, 2005, pp. 39–57.

[6] A. Lastovetsky and R. Reddy, "Data distribution for dense factorization on computers with memory heterogeneity," *Parallel Computing*, vol. 33, no. 12, Dec. 2007.

[7] D. Clarke, A. Lastovetsky, and V. Rychkov, "Column-based matrix partitioning for parallel matrix multiplication on heterogeneous processors based on functional performance models," in *Euro-Par 2011: Parallel Processing Workshops*, ser. Lecture Notes in Computer Science, vol. 7155. Springer-Verlag, 2012.

[8] OpenBLAS, "OpenBLAS: An optimized BLAS library," 2016. [Online]. Available: http://www.openblas.net/

[9] FFTW, "FFTW: A fast, free c FFT library," 2016. [Online]. Available: http://www.fftw.org/

[10] A. Ilić, F. Pratas, P. Trancoso, and L. Sousa, "High-performance computing on heterogeneous systems: Database queries on CPU and GPU," *High Performance Scientific Computing with Special Emphasis on Current Capabilities and Future Perspectives*, pp. 202–222, 2010.

[11] D. Clarke, A. Lastovetsky, and V. Rychkov, "Dynamic load balancing of parallel computational iterative routines on highly heterogeneous HPC platforms," *Parallel Processing Letters*, vol. 21, no. 02, pp. 195–217, 2011.

[12] X. Liu, Z. Zhong, and K. Xu, "A hybrid solution method for CFD applications on GPU-accelerated hybrid HPC platforms," *Future Generation Computer Systems*, vol. 56, pp. 759–765, 2016.

[13] M. Radmanović, D. Gajić, and R. Stanković, "Efficient computation of galois field expressions on hybrid CPU-GPU platforms." *Journal of Multiple-Valued Logic & Soft Computing*, vol. 26, 2016.

[14] A. Ilic and L. Sousa, "Simultaneous multi-level divisible load balancing for heterogeneous desktop systems," in *Parallel and Distributed Processing with Applications (ISPA), 2012 IEEE 10th International Symposium on*. IEEE, 2012, pp. 683–690.

[15] J. Colaço, A. Matoga, A. Ilic, N. Roma, P. Tomás, and R. Chaves, "Transparent application acceleration by intelligent scheduling of shared library calls on heterogeneous systems," in *Parallel Processing and Applied Mathematics*. Springer, 2013, pp. 693–703.

[16] V. Cardellini, A. Fanfarillo, and S. Filippone, "Heterogeneous sparse matrix computations on hybrid GPU/CPU platforms." in *PARCO*, 2013, pp. 203–212.

[17] Z. Zhong, V. Rychkov, and A. Lastovetsky, "Data partitioning on multicore and multi-GPU platforms using functional performance models," *Computers, IEEE Transactions on*, vol. 64, no. 9, pp. 2506–2518, 2015.

[18] A. Lastovetsky, L. Szustak, and R. Wyrzykowski, "Model-based optimization of EULAG kernel on Intel Xeon Phi through load imbalancing," *IEEE Transactions on Parallel and Distributed Systems*, 08/2016.

[19] Y. Ogata, T. Endo, N. Maruyama, and S. Matsuoka, "An efficient, model-based CPU-GPU heterogeneous FFT library," in *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*. IEEE, 2008, pp. 1–10.

[20] C. Yang, F. Wang, Y. Du, J. Chen, J. Liu, H. Yi, and K. Lu, "Adaptive optimization for petascale heterogeneous CPU/GPU computing," in *Cluster Computing (CLUSTER), 2010 IEEE International Conference on*. IEEE, 2010, pp. 19–28.

[21] M. D. Linderman, J. D. Collins, H. Wang, and T. H. Meng, "Merge: a programming model for heterogeneous multi-core systems," in *ACM SIGOPS operating systems review*, vol. 42, no. 2. ACM, 2008, pp. 287–296.

[22] G. Quintana-Ortí, F. D. Igual, E. S. Quintana-Ortí, and R. A. van de Geijn, "Solving dense linear systems on platforms with multiple hardware accelerators," *SIGPLAN Not.*, vol. 44, no. 4, pp. 121–130, Feb. 2009.

[23] C. Augonnet, S. Thibault, and R. Namyst, "Automatic Calibration of Performance Models on Heterogeneous Multicore Architectures," in *3rd Workshop on Highly Parallel Processing on a Chip (HPPC 2009)*, Aug. 2009.

[24] G. Cybenko, "Dynamic load balancing for distributed memory multiprocessors," *J. Parallel Distrib. Comput.*, vol. 7, no. 2, pp. 279–301, Oct. 1989.

[25] J. M. Bahi, S. Contassot-Vivier, and R. Couturier, "Dynamic load balancing and efficient load estimators for asynchronous iterative algorithms," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 16, no. 4, pp. 289–299, 2005.

[26] A. Legrand, H. Renard, Y. Robert, and F. Vivien, "Mapping and load-balancing iterative computations," *IEEE Trans. Parallel Distrib. Syst.*, vol. 15, no. 6, Jun. 2004.

[27] R. L. Cariño and I. Banicescu, "Dynamic load balancing with adaptive factoring methods in scientific applications," *The Journal of Supercomputing*, vol. 44, no. 1, pp. 41–63, 2008.

[28] J. A. Martínez, E. M. Garzón, A. Plaza, and I. García, "Automatic tuning of iterative computation on heterogeneous multiprocessors with ADITHE," *J. Supercomput.*, vol. 58, no. 2, Nov. 2011.

[29] M. Cierniak, M. J. Zaki, and W. Li, "Compile-time scheduling algorithms for a heterogeneous network of workstations," *The Computer Journal*, vol. 40, no. 6, pp. 356–372, 1997.

[30] O. Beaumont, V. Boudet, F. Rastello, and Y. Robert, "Matrix multiplication on heterogeneous platforms," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 12, no. 10, pp. 1033–1051, 2001.

[31] A. Kalinov and A. Lastovetsky, "Heterogeneous distribution of computations solving linear algebra problems on networks of heterogeneous computers," *J. Parallel Distrib. Comput.*, vol. 61, no. 4, Apr. 2001.

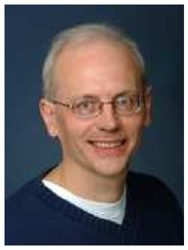
[32] A. L. Lastovetsky, L. Szustak, and R. Wyrzykowski, "Model-based optimization of MPDATA on Intel Xeon Phi through load imbalancing," *CoRR*, vol. abs/1507.01265, 2015. [Online]. Available: http://arxiv.org/abs/1507.01265

[33] S. Rivoire, P. Ranganathan, and C. Kozyrakis, "A comparison of high-level full-system power models," in *Proceedings of the 2008 Conference on Power Aware Computing and Systems*, ser. HotPower'08. USENIX Association, 2008.

[34] J. C. McCullough, Y. Agarwal, J. Chandrashekar, S. Kuppuswamy, A. C. Snoeren, and R. K. Gupta, "Evaluating the effectiveness of model-based power characterization," in *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*, ser. USENIXATC'11. USENIX Association, 2011.

[35] K. O'Brien, I. Pietri, R. Reddy, A. Lastovetsky, and R. Sakellariou, "A survey of power and energy predictive models in HPC systems and applications," *Submitted to ACM Computing Surveys*, 2016.

[36] PFFTW, "Parallel FFTW," 2016. [Online]. Available: http://www.fftw.org/fftw2_doc/fftw_4.html



Alexey Lastovetsky received a PhD degree from the Moscow Aviation Institute in 1986, and a Doctor of Science degree from the Russian Academy of Sciences in 1997. His main research interests include algorithms, models, and programming tools for high performance heterogeneous computing. He has published over a hundred technical papers in refereed journals, edited books, and international conferences. He authored the monographs Parallel computing on heterogeneous networks (Wiley, 2003) and High performance heterogeneous computing (Wiley, 2009).

Ravi Reddy Manumachu received a B.Tech degree from I.I.T, Madras in 1997 and a PhD degree from the School of Computer Science and Informatics, University College Dublin in 2005. His main research interests include high performance heterogeneous computing, energy efficient computing, and complexity theory.