IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received October 3, 2018, accepted October 29, 2018, date of publication November 12, 2018,
date of current version December 7, 2018.

Digital Object Identifier 10.1109/ACCESS.2018.2879228

Parallel Data Partitioning Algorithms for Optimization of Data-Parallel Applications on Modern Extreme-Scale Multicore Platforms for Performance and Energy

RAVI REDDY MANUMACHU AND ALEXEY LASTOVETSKY
School of Computer Science, University College Dublin, Dublin 4, D04 V1W8 Ireland

Corresponding author: Ravi Reddy Manumachu (ravi.manumachu@ucd.ie)

This work was supported by the Science Foundation Ireland under Grant 14/IA/2474.

ABSTRACT Data partitioning algorithms aiming to minimize the execution time and the energy of computations in self-adaptable data-parallel applications on modern extreme-scale multicore platforms must address two critical challenges. First, they must take into account the new complexities inherent in these platforms such as severe resource contention and non-uniform memory access. Second, they must have low practical runtime and memory costs. The sequential data partitioning algorithms addressing the first challenge have a theoretical time complexity of O($m*m*p*p$) where m is the number of points in the discrete speed/energy function and p is the number of available processors. They, however, exhibit high practical runtime cost and excessive memory footprint, therefore, rendering them impracticable for employment in self-adaptable applications executing on extreme-scale multicore platforms. We present, in this paper, the parallel data partitioning algorithms that address both the challenges. They take as input the functional models of performance and energy consumption against problem size and output workload distributions, which are globally optimal solutions. They have a low time complexity of O($m * m * p$) thereby providing a linear speedup of O(p) and low memory complexity of O(n) where n is the workload size expressed as a multiple of granularity. They employ dynamic programming approach, which also facilitates the easier integration of performance and energy models of communications. We experimentally study the practical cost of application of our algorithms in two data-parallel applications, matrix multiplication and fast Fourier transform, on a cluster in Grid'5000 platform. We demonstrate that their practical runtime and memory costs are low making them ideal for employment in self-adaptable applications. We also show that the parallel algorithms exhibit tremendous speedups over the sequential algorithms. Finally, using theoretical analysis for a forecast exascale platform, we demonstrate that the parallel algorithms have negligible execution times compared to the matrix multiplication application executing on the platform.

INDEX TERMS Data parallelism, data partitioning, energy, energy optimization, homogeneous multicore CPU clusters, load balancing, parallel algorithms, performance, performance optimization.

I. INTRODUCTION

Data partitioning algorithms aiming to solve the optimization problems of minimization of time and energy of computations in self-adaptable data-parallel applications on modern extreme-scale multicore CPU platforms must address two formidable challenges.

By self-adaptable applications, we mean applications that automatically adapt at runtime to any set of heterogeneous processors with *a priori* unknown performance characteristics [2]. They are typically executed in dynamic environments or environments where the number of available processors and their performance characteristics can be different for different runs of the same application. They must adapt at runtime to dynamic changes in the environment in even a single run. Such applications should be able to optimally distribute computations between the processors of the executing platform assuming that this platform is different and *a priori* unknown for each run of the application.

Self-adaptability is essential not only due to the changing underlying execution environment but also due to the specific characteristics/requirements of the application domains (for example: adaptive mesh refinement, particle simulations, transient dynamics calculations, etc), and autotuning softwares. We furnish real-life use cases that highlight its importance below. They are detailed in the Appendix B, supplemental.

- *Self-adaptability of the solver* is vital in adaptive mesh refinement on clusters for solving large computational fluid dynamics (CFD) and computational mechanics (CM) problems where the computational load varies throughout the evolution of the solution [3]–[6].
- *Autotuning parallel softwares* perform an empirical search by generating numerous versions of a program at runtime, which are then executed to find the best configuration of a program. The key building blocks that enable them to prune and accomplish this search in reasonable runtime are fast data-partitioning algorithms employing realistic computation and communication performance models [7]–[9].
- *Supercomputer administrators* routinely report that nodes closer to the hotter regions (hotspots) execute codes slower than the nodes closer to the cooler regions in the supercomputing centres due to variations in the airflow caused by the layout of the cooling systems [10]–[13]. Therefore, static data partitioning strategies are not ideally suitable to address this situation
- *Shared environments* such as cloud computing systems today are placing great emphasis in facilitating easier migration and execution of HPC workloads by striving to remove impediments to this process. The leading objectives for optimization for the cloud service providers are performance, energy consumption, cost, and reliability. Self-adaptable applications employing fast data partitioning algorithms for optimization of their performance and energy evidently and directly address the first two concerns [14].

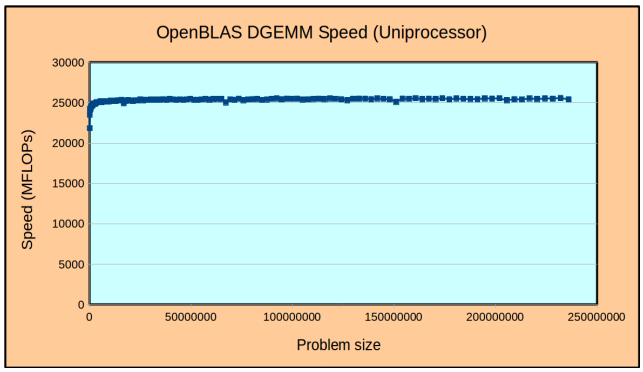
We describe now the two crucial challenges to data partitioning algorithms posed by self-adaptable data-parallel applications executing on modern extreme-scale multicore CPU platforms.

To elucidate the first challenge, we compare the typical shapes of real-life scientific data-parallel applications on platforms consisting of uniprocessors and modern multicore CPUs. For this purpose, we select two widely used and highly optimized scientific routines, dense matrix-matix multiplication (OpenBLAS DGEMM) [15] and fast Fourier transform (FFTW) [16], [17].

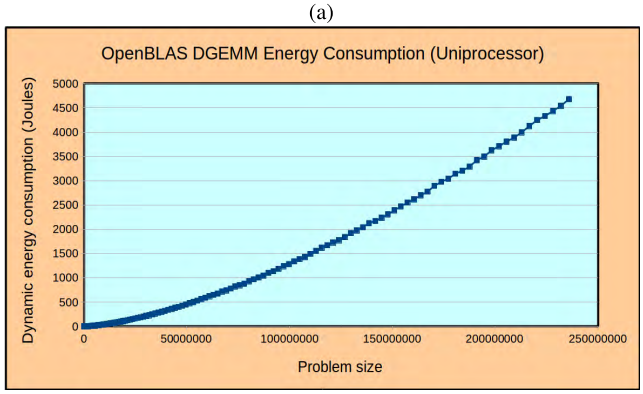
Consider the shapes of the speed and dynamic energy consumption functions of the OpenBLAS DGEMM application built experimentally by executing it on a single core of an Intel Haswell workstation (specification shown in Table 1). The application multiplies two dense square matrices of size $n \times n$ (problem size is equal to n^2). For a problem size n^2 in the speed function, the speed is equal to $\frac{2\times n^3}{t}$ where t is execution time taken to multiply two $n \times n$ square matrices. In these experiments, the *numactl* tool binds the application to one core. The dynamic energy consumptions are obtained using *Watts Up Pro* power meter (Appendix C).

TABLE 1. Specification of the intel haswell workstation used to build the uniprocessor speed and energy models.

Technical Specifications	Intel Haswell i5-4590
Processor	Intel i5-4590 3.3 GHz
Microarchitecture	Haswell
Memory	8 GB
Socket(s)	1
Core(s) per socket	4
L1d cache	32 KB
L11 cache	32 KB
L2 cache	256 KB
L3 cache	6144 KB
TDP	84 W
Base Power	22.3 W
Max Turbo Frequency	3.7 GHz



(a)

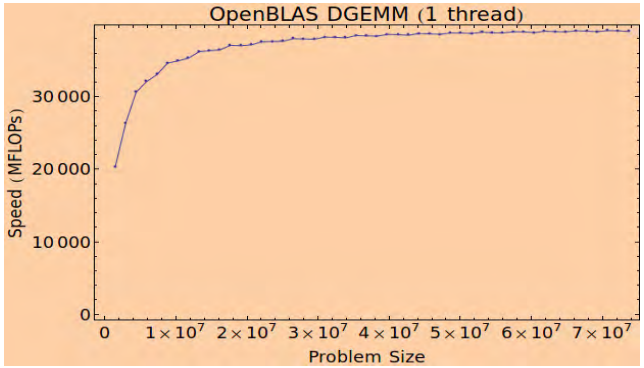


(b)

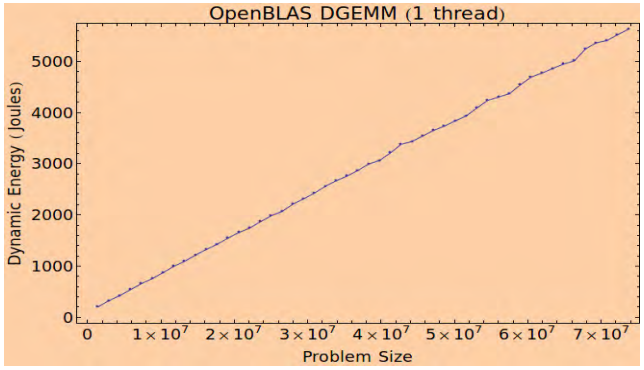
FIGURE 1. a). Speed function of OpenBLAS DGEMM application executed on a single core on the Intel Haswell workstation. b). Dynamic energy consumption of OpenBLAS DGEMM application executed on a single core on the Intel Haswell workstation.

Figures 1a and 1b show the shapes of the speed and energy functions. The salient properties of the shapes are below:

- The functions are smooth.
- The speed function satisfies the following properties:
  - Monotonically increasing.



(a)



(b)

FIGURE 2. a). Speed function of OpenBLAS DGEMM application executed on a single core on the Intel Haswell server. b). Dynamic energy consumption of OpenBLAS DGEMM application executed on a single core on the Intel Haswell server.

  - Concave.
  - Any straight line passing through the origin of the coordinate system intersects the graph of the function in no more than one point.
- The dynamic energy consumption is a monotonically increasing convex function of problem size.

For such shapes, Lastovetsky and Reddy [18] prove that the solutions determined by the traditional (based on constant performance model) and the state-of-the-art load-balancing algorithms based on functional performance models (FPMs) [19]–[26], simultaneously minimize the execution time and dynamic energy consumption of computations in the parallel execution of the application. Figures 2a and 2b show the shapes of the speed and dynamic energy consumption functions for the same application built experimentally by executing it on a single core of an Intel Haswell server (specification shown in Table 2). While the shape of the speed function is the same as before, the shape of the dynamic energy consumption function is linear. This implies that all workload distributions will result in same dynamic energy consumption and therefore parallelization has no effect on the dynamic energy consumption of computations in the parallel execution of the application.

To summarize, on platforms composed of uniprocessors, the shapes of the performance and energy functions are smooth with minimal variations. The performance functions

TABLE 2. Specification of the intel haswell server used to build the FPM and energy model for multithreaded OpenBLAS DGEMM and FFTW applications.

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
L11 cache	32 KB
L2 cache	256 KB
L3 cache	30720 KB
TDP	240 W
Base Power	58 W

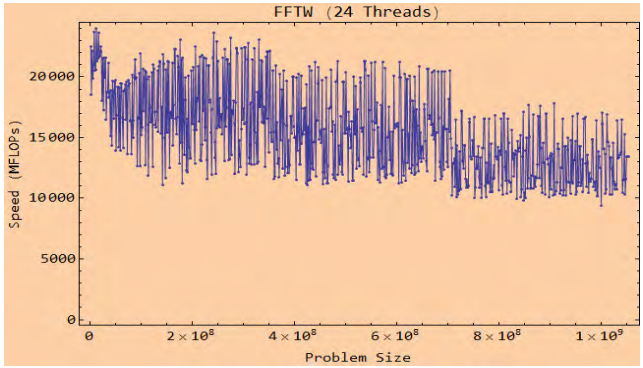
comfortably satisfy the conditions imposed by the FPMs that are crucial for the correct operation of the load balancing algorithms.

On modern homogeneous clusters composed of multicore CPUs, the performance and energy profiles of real-life scientific applications executing on these platforms are not smooth and may deviate considerably from the shapes observed before. This is due to the newly introduced inherent complexities such as resource contention and NUMA. Figures 3 and 4 illustrate the shapes for the speed and dynamic energy consumption graphs for multi-threaded OpenBLAS DGEMM and FFTW applications executed with 24 threads on the Intel Haswell server. The FFTW application performs a 2D FFT of size $n \times n$ (the problem size being n^2). For a problem size n^2 in the speed function, the speed is equal to $\frac{5.0 \times n^2 \times \log_2 n^2}{t}$ where t is execution time taken to compute 2D complex DFT of size n^2.

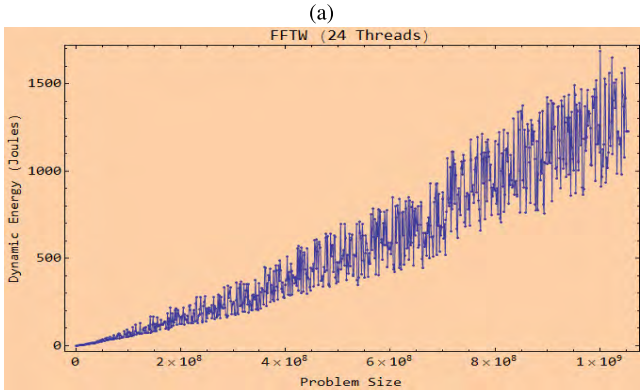
For each function, solid lines connect the points in its graph to highlight the variations. To make sure the experimental results are reliable, we follow a statistical methodology described in the Appendix D. Briefly, for every data point in the functions, the automation software executes the application repeatedly until the sample mean lies in the 95% confidence interval with a precision of 0.025 (2.5%). For this purpose, we use Student's t-test assuming that the individual observations are independent and their population follows the normal distribution. We verify the validity of these assumptions using Pearson's chi-squared test.

Therefore, the variation observed is not noise but is an inherent trait of applications executing on multicore servers with resource contention and NUMA. In a function (speed or energy f), it is the difference of function values between two subsequent local minima (f_1) and maxima (f_2) defined as following:

$$variation(\%) = \frac{|f_1 - f_2|}{\min(f_1, f_2)} \times 100 \quad (1)$$

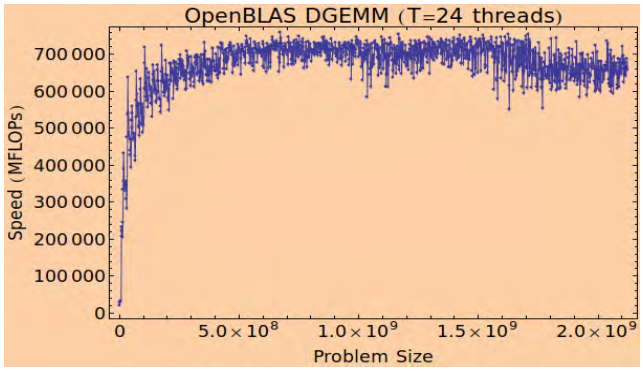


(a)

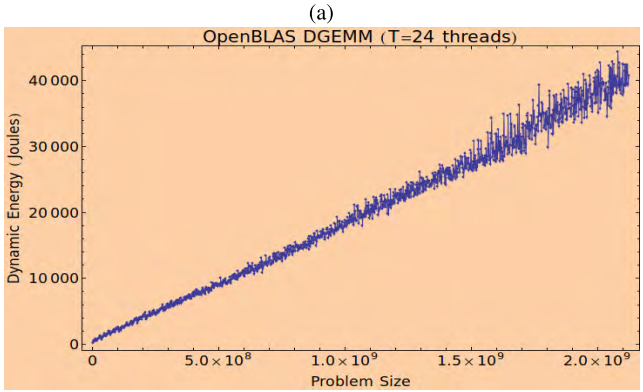


(b)

FIGURE 3. a). Speed function of FFTW executing 24 threads on the Intel Haswell server. b). Function of dynamic energy consumption against problem size for FFTW executing 24 threads on the Intel Haswell server.



(a)



(b)

FIGURE 4. a). Speed function of OpenBLAS DGEMM executing 24 threads on the Intel Haswell server. b). Function of dynamic energy consumption against problem size for OpenBLAS DGEMM executing 24 threads on the Intel Haswell server.

The salient points about the variations are below:

- They can be large. This is evident from the speed and energy functions shown in Fig. 3a and 3b. From the speed function plot, one can observe performance drops of around 70% for multiple problem sizes.
- The variations is not due to the constant and stochastic fluctuations owing to OS activity or a workload executing in a node in common networks of computers. 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. For a node with uniprocessors, the width of the band decreases as the problem size increases. For a node with a 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 decreases. Therefore, for long running applications, the width would become narrow (3%). This however is not the case for variations in the presented graphs. The dynamic energy consumption in the Figures 3b and 4b show the widths of the variations increasing as problem size increases. These widths reach a maximum of 70% and 125% for large problem sizes. The speed functions in the Figures 3a and 4a demonstrate that the widths are bounded with the averages around 17% and 60%. This suggests therefore that the variation is due to the newly introduced complexities and not due to the fluctuations arising from changing transient load.

Therefore, these variations are not singular and will become natural because chip manufacturers are increasingly favoring and thereby rapidly progressing towards tighter integration of processor cores, memory, and interconnect in their products.

Due to these variations, optimization problems of minimization of time and energy of computations for the most general shapes of performance and energy profiles of data-parallel applications have become difficult to solve. To demonstrate why this is the case, we zoom into the energy function of the OpenBLAS DGEMM application to analyze its properties (which are also exhibited by the speed functions). Figure 5 shows the energy function between two arbitrarily chosen points A and B.

- One can observe that the energy function is characterized by multiple local minima ($Q_1, Q_2, ...$) and multiple local maxima ($P_1, P_2, ...$). There is one global maximum P and one global minimum Q.
- The function (feasible region) is non-linear and non-convex.

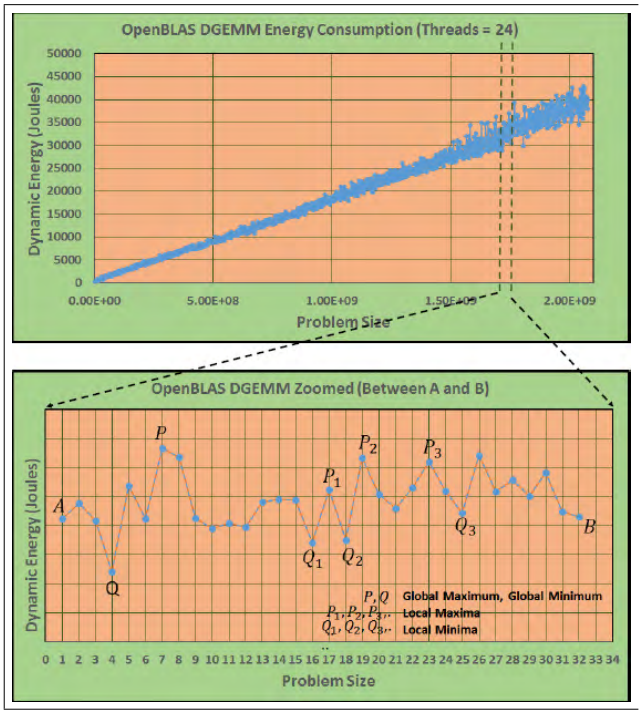


FIGURE 5. Zoomed energy function of OpenBLAS DGEMM application between two arbitrarily chosen points *A* and *B*. Dashed lines connect the points for clarity.

- Since deriving an analytical formula for such a function is non-trivial, optimization solvers that rely on the existence of derivatives cannot be directly applied.

We summarize the challenges below:

1) Modern extreme-scale multicore CPU platforms are composed of tightly integrated multicore CPUs with highly hierarchical arrangement of cores. This tight integration has resulted in the cores contending for shared on-chip resources such as Last Level Cache (LLC) and interconnect (For example: Intel's Quick Path Interconnect, AMD's Hyper Transport), 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 executing on these platforms are not smooth and may deviate considerably from the shapes observed before. These behaviors limit the applicability of state-of-the-art load balancing algorithms (based on functional performance models (FPMs)) thereby necessitating either a thorough redesign or development of novel models and algorithms.
2) The data partitioning algorithm employed in a self-adaptable application to optimally distribute computations must have low practical runtime and memory costs compared to that of the application. This would make them ideal for application even in domains where time and memory are more critical.

We present an overview of the latest research efforts addressing the challenges before highlighting their drawbacks that make them impracticable for employment in self-adaptable applications.

Lastovetsky and Reddy [18] 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. They formulate the performance and energy optimization problems and present efficient algorithms of complexity $O(m^2 \times p^2)$ solving these problems where m is the cardinality of the discrete sets representing the speed/energy functions and p is the number of available processors. The memory complexity of the algorithms are $O(n \times p^2)$. Unlike load balancing algorithms, optimal solutions found by these algorithms may not load-balance an application. Manumachu and Lastovetsky [27] study the bi-objective optimization problem for performance and energy (BOPPE) for data-parallel applications on homogeneous clusters of modern multicore CPUs. It employs only one but heretofore unstudied decision variable, the problem size. They present an efficient and exact global optimization algorithm called ALEPH that solved the BOPPE. It takes as inputs, functions of performance and dynamic energy consumption against problem size, and outputs the globally Pareto-optimal set of solutions. They prove the complexity of the algorithm to be $O(m^2 \times p^2)$ where m is the cardinality of the discrete sets representing the speed/energy functions and p is the number of available processors. The memory complexity of the algorithm is $O(n \times p^2)$.

The proposed data-partitioning algorithms are, however, sequential, recursive, and have high practical runtime and memory costs for large values of p (in the order of hundreds). For such large values of p, the runtime cost is in the order of minutes and the memory cost is also high causing severe degradation of performance due to paging. Therefore, these two prohibitive costs render them unsuitable for employment in self-adaptable applications executing on extreme-scale multicore platforms.

We present in this work parallel data partitioning algorithms that address both the challenges. Like the sequential algorithms, they take as input the functional models of performance and energy consumption against problem size and output workload distributions, which are globally optimal solutions. They have low time complexity of $O(m^2 \times p)$ thereby providing linear speedup of $O(p)$ and low memory complexity of $O(n)$ where n is the workload size expressed as a multiple of granularity. They employ dynamic programming approach, which also facilitates easier integration of performance and energy models of communications.

We experimentally study the practical runtime costs of the algorithms in two data parallel applications, matrix multiplication and fast Fourier transform, on a cluster in Grid'5000 platform. We demonstrate that their practical runtime and memory costs are low making them ideal for employment in self-adaptable applications. We also show that the parallel algorithms exhibit tremendous speedups over the sequential algorithms. Finally, using theoretical analysis for

a forecast exascale platform, we demonstrate that the parallel algorithms have negligible execution times compared to the matrix multiplication application executing on the platform.

To summarize, the main contributions in this work are:

- Efficient parallel data partitioning algorithms for performance and energy optimization problems for data-parallel applications executing on homogeneous multicore CPU clusters. The algorithms address the challenges posed by complexities of NUMA and severe resource contention inherent in modern multicore platforms and have low computational and memory complexity thereby making them suitable for employment in self-adaptable data-parallel applications. While there are other possible solution methods for parallelization, our solution using dynamic programming technique is holistic since it allows easy integration of performance and energy models of communication.
- Practical application where we show how a loop containing recursive invocations is carefully restructured to allow parallel computation of the loop iterations.
- Illustration of how the cost of communications during the execution of the data-parallel application can be seamlessly integrated in our parallel data partitioning algorithms.
- Experimental study of practical cost of the parallel data partitioning algorithms for large clusters and theoretical study of the execution times of the algorithms on extreme-scale parallel platforms.
- Efficient implementation of the parallel data partitioning algorithms that employ hybrid MPI+OpenMP programming model with threaded MPI communications for minimizing the execution time.

We organize the rest of the paper as follows. Section II presents related work on data partitioning techniques targeted for self-adaptable applications. Section III contains formulations of performance and energy optimization problems for homogeneous multicore clusters. Section IV presents an overview of the sequential data partitioning algorithms solving the performance and energy optimization problems for homogeneous multicore clusters. Section V presents the parallel data partitioning algorithms. Section VI contains experimental and theoretical analysis of the algorithms. Section VII concludes the paper.

II. RELATED WORK

We classify our literature survey into four categories summarized in Table 3.

The first category deals with data partitioning algorithms employed for performance optimization on HPC platforms. The second category presents self-adaptable applications and dynamic runtime schedulers. The third category surveys efforts that investigate loop and data transformations to improve performance of regular and irregular codes. Final category deals specifically with works that have proposed data partitioning techniques targeted for self-adaptable applications on heterogeneous platforms.

TABLE 3. Main categories in the related work.

Category	Description
II-A	Data partitioning algorithms employing load balancing techniques
II-B	Self-adaptable Applications and Runtime Schedulers
II-C	Loop and Data Transformations
II-D	Dynamic Data Partitioning on Heterogeneous Platforms

A. DATA PARTITIONING ALGORITHMS BASED ON LOAD BALANCING

There are different classifications of load-balancing algorithms: *static* or *dynamic*, *non-centralized* or *centralized*, *task queue* or *predicting-the-future*.

Static algorithms, such as those based on data partitioning [19], [28], [29], 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. They however are unsuitable for non-dedicated platforms, where load changes with time. Dynamic algorithms, such as task scheduling and work stealing [30]–[32], 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 small load imbalance, and lesser scheduling overhead.

In non-centralized algorithms [33], [34], load is migrated locally between neighboring processors, while in centralized ones [35]–[37], load is distributed based on global load information. Non-centralized algorithms are slower to converge. At the same time, centralized algorithms typically have higher overhead. The centralized algorithms can be further subdivided into two groups: task queue [36] and predicting-the-future [35], [37].

B. SELF-ADAPTABLE APPLICATIONS AND RUNTIME SCHEDULERS

Hendrickson and Devine [6] survey approaches addressing the dynamic load balancing problem that arises in computational mechanics applications where the computation must adapt dynamically during the simulation (for e.g., adaptive mesh refinement, particle simulations and transient dynamics calculations). Among the essential properties identified by them that a dynamic load balancer should possess are parallel speed and modest memory usage.

Runtime schedulers such as KAAPI [38], StarPU [39], and DAGuE [40] schedule an application described as a Direct Acyclic Graph (DAG) or task graph onto parallel

platforms. The DAG expresses different types of tasks and the data dependencies between them and is created either statically or dynamically. Little information exists on the computational performance and memory utilization of DAG schedulers. They cater to particular classes of applications (sparse, irregular, etc) that are not applicable to our data partitioning algorithms. For the data-parallel applications, it is a edifying research task to conduct a comprehensive comparison between them and static and dynamic load-balancing and load-imbalancing algorithms.

C. LOOP AND DATA TRANSFORMATIONS

Loop and data transformations to improve the locality of regular and irregular codes is an active area of research. Han and Tseng [41] propose low overhead data and computation re-ordering techniques to improve the locality of irregular codes. Jo and Kulkarni [42] propose a locality enhancing technique that is based on loop tiling to improve the performance of applications that use recursive data structures such as trees and graphs. Ravishankar *et al.* [43] present an approach for parallel execution of a class of irregular loop computations using a combination of static and runtime analysis. Venkat *et al.* [44] describe a generalized loop-coalescing transformation to represent and transform computations. In this work, we transform a sequential, recursive loop using dynamic programming technique so that the main loop iterations are executed in parallel.

D. DYNAMIC DATA PARTITIONING ON HETEROGENEOUS PLATFORMS

All the works examined in this category strive to achieve dynamic load balance. They use a simple principle of ''using the past to predict the future'' where they employ the information (speeds, execution times, etc) from the current iteration to redistribute work for the ensuing iterations. Self-adaptable applications, which are typically executed in dynamic environments, may invoke a data partitioning algorithm multiple times due to which cost of data redistribution or migration is incurred.

Legrand *et al.* [35] study mapping of iterative computations onto heterogeneous clusters. At each iteration, local computations are performed in parallel and some communications (boundary information) take place between consecutive processors in the ring. The authors consider the problem of optimal partitioning the workload in each iteration taking into account the computations and communications so that the total execution time is minimized. They prove the NP-completeness of the problem and design an efficient heuristic. Mahanti and Eager [45] study different data redistribution policies when processors (or nodes) are added or removed during the execution of a data parallel application in a dynamic heterogeneous environment. In their approach, the cost of the data partitioning algorithm is not the main concern.

Galindo *et al.* [46] propose a dynamic load balancing approach to balance the workload of iterative algorithms in dedicated heterogeneous platforms. Before the start of execution of the iterative algorithm, homogeneous distribution of the workload is used. The speeds of the processors are determined after the execution of one iteration. These speeds are used to determine new workload distribution for the next iteration. Martínez *et al.* [37], [47] propose a dynamic load balancing approach to balance the workload of iterative algorithms in heterogeneous dedicated and non-dedicated platforms composed of multiprocessor nodes.

Clarke *et al.* [2] propose a data partitioning algorithm, which is practicable for employment in self-adaptable applications due to its low runtime cost. This algorithm does not require as input the full functional performance model (FPM). Unlike algorithms which require construction of full FPMs as a prerequisite, it builds a partial estimate of the FPM and uses it to determine optimal data partitioning with a given accuracy. Sanjuan-Estrada *et al.* [48] propose a dynamic load balancing strategy, which determines the number of threads at runtime (at different stages of an application execution) based on two decisions. These are the completed work and the existence of a sleeping thread in the application. The execution of an application starts with one thread. The strategy uses these decisions to determine if a thread needs to be created to maintain load balance at different (predetermined or equidistant) stages of the application. Wang *et al.* [49] present a self-adaptive and parallelized maximum likelihood evaluation (MLE) framework. It consists of a master process and a set of worker processes in a distributed environment where the master is responsible for re-distributing the computing tasks to workers and the workers compute tasks. The goal of the framework is to achieve load balance of workload between the workers. The workload distribution is determined using the execution times of the workers. Acosta *et al.* [50] propose a dynamic load balancing approach to balance the workload of iterative algorithms in homogeneous and heterogeneous multi-GPU platforms. The approach is not novel and is similar to the efforts presented earlier. Zhang *et al.* [51] report significant non-deterministic variations for applications that are not bound to the cores of the executing multicore platform. Their approach is to try to reduce the non-deterministic variations by using different execution patterns.

The data partitioning algorithm that we propose has noteworthy differences. First, it is a parallel algorithm. Second, it takes as input a functional performance model and not a constant performance model such as an execution time to determine the workload distribution. Third, workload distribution (which is globally optimal) found by it may not load-balance the application.

III. FORMULATIONS OF PERFORMANCE OPTIMIZATION AND ENERGY OPTIMIZATION PROBLEMS

Before we present the formulations, we would like to define the meaning of the terms ''problem size'' and ''workload size'' used in this work. These two terms are used synonymously in the literature. The problem size is defined as a set of one, two or more parameters characterizing the amount and

layout of data stored and processed during the execution of a computational task. It also represents the size of a computational task that is allocated to a processor during the parallel execution of a data-parallel application. The workload size is defined as the size of the workload of the data-parallel application that is executed using one or more processors. It is a multiple of one or more computational tasks, whose size is defined to be the problem size. By data-parallel workloads, we mean computations involving dense objects such as dense matrices or grids (for example, dense linear algebra routines such as matrix-matrix multiplication of dense matrices, fast Fourier transform of a dense signal matrix, etc.).

Consider a data-parallel application workload of size n executed using p number of identical processors. Let the speed function of a processor executing a problem size x be represented by $s(x)$. Here the speed can be measured in floating point operations per second or any other fixed-size computation units per unit time. The size of workload can be characterized by the problem size (for example, the number of cells in the computational domain or the matrix size) or just by the number of equal-sized computational units. 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 x. n and x are considered one-dimensional. Since the processors involved in the execution of the workload are identical, the input to the problem is a single speed function. We do not specify how to build the speed function. It may be constructed using one or more processors.

The formulation for the performance optimization problem follows:

A. 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 q number of identical processors that minimizes the computation time of parallel execution of the workload. The parameters (n, p, s) are the inputs to the problem. The parameters (q,d) are the outputs. The formulation of the problem (as an integer non-linear program (INLP)) follows:

$$\begin{aligned} &\textit{minimize } \max_{i=1}^{q} \frac{x_i}{s(x_i)} \\ &\text{Subject to } x_1 + x_2 + ... + x_q = n \\ &\qquad x_i \leq n \quad i = 1, ..., q \\ &\qquad x_i > 0 \quad i = 1, ..., q \\ &\qquad 1 \leq q \leq p \end{aligned}$$

where

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

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

$$\begin{aligned} &\textit{minimize } f \\ &\text{Subject to } f \geq \frac{x_i}{s(x_i)} \quad i = 1, ..., q \\ &\qquad x_1 + x_2 + ... + x_q = n \\ &\qquad x_i \leq n \quad i = 1, ..., q \\ &\qquad x_i > 0 \quad i = 1, ..., q \\ &\qquad 1 \leq q \leq p \end{aligned}$$

where

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

Informally speaking, the objective function in *POPT* is a function of workload distribution $D = \{x_1, ..., x_p\}$, of a given workload n executed using p number of identical processors. For each given D, it returns the time of computations involved in its parallel execution, which is calculated as the time taken by the longest running processor to execute its assigned problem size. The distribution that minimizes this function is considered optimal as its execution time of workload n using the p processors cannot be improved. The optimal distribution may utilize number of processors (q) less than or equal to p.

The formulation for optimization problem for energy is based on an energy model, which represents the dynamic energy consumption of a processor by a function of problem size. The dynamic energy consumption of execution of a problem size x by a processor is represented by $\Omega(x)$. We explain the rationale behind using dynamic energy consumption in Appendix C. Since the processors involved in the execution of the workload are identical, the input to the problem is a single energy function. We do not specify how to build the energy function. It may be constructed using one or more processors.

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 q number of identical processors that minimizes the dynamic energy consumption of computations in the parallel execution of the workload. The parameters (n, p, Ω) are the inputs. The parameters (q,d) are the outputs. The formulation of the problem (as an integer linear program (ILP)) follows:

$$\begin{aligned} &\textit{minimize } \sum_{i=1}^{q} \Omega(x_i) \\ &\text{Subject to } x_1 + x_2 + ... + x_q = n \\ &\qquad x_i \leq n \quad i = 1, ..., q \\ &\qquad x_i > 0 \quad i = 1, ..., q \\ &\qquad 1 \leq q \leq p \end{aligned}$$

where

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

Informally speaking, the objective function in *EOPT* is a function of workload distribution $D = \{x_1, ..., x_p\}$, of a given workload n executed using p number of identical processors. For each given D, it returns the energy consumption of the computations involved in its parallel execution, which is calculated as the sum of the energy consumptions of all processors executing problem sizes assigned to them. The distribution that minimizes this function is considered

Algorithm 1 Algorithm Determining Optimal Distribution of Workload of Size n for Maximizing Performance

1: **procedure** POPTA(n, p, Δx, 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: $(X_\uparrow[point], S_\uparrow[point]) \leftarrow \text{Sort}\uparrow(point, X, S)$
4: **end for**
5: $(B, E) \leftarrow \text{GetBE}(n,p,\Delta x,X,S)$
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**

optimal as its energy consumption of workload n using the p processors cannot be improved. The optimal distribution may utilize number of processors (q) less than or equal to p.

POPT and EOPT are also known as *min-max* and *min-sum* problems.

IV. *POPTA* AND *EOPTA*: SEQUENTIAL DATA PARTITIONING ALGORITHMS SOLVING POPT AND EOPT

We present an overview of the sequential data partitioning algorithms [18] solving the performance and energy optimization problems.

A. POPTA: ALGORITHM SOLVING POPT PROBLEM

The algorithm POPTA (Algorithm 1) solves POPT. The inputs to POPTA 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, containing problem sizes and speeds. m is the cardinality of the sets X and S. The outputs are the

Algorithm 2 The Kernel of the Algorithm 1

1: **function** POPTAKernel(n, p, Δx, 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: $t_{opt} \leftarrow \max_{1 \leq i \leq p} (\frac{D^i_{opt}}{S[D^i_{opt}]})$
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** $t_r \geq E$ **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: $(B_l, E_l) \leftarrow \text{GetBE}(n_l,p-r,\Delta x,X,S)$
20: **if** $B_l > |X|$ **then continue end if**
21: **if** $(n_l \bmod (p-r) \neq 0)$ 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** $(t_l \leq E_l$ 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** $t_l \leq t_r$ **then Go To** 44 **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**

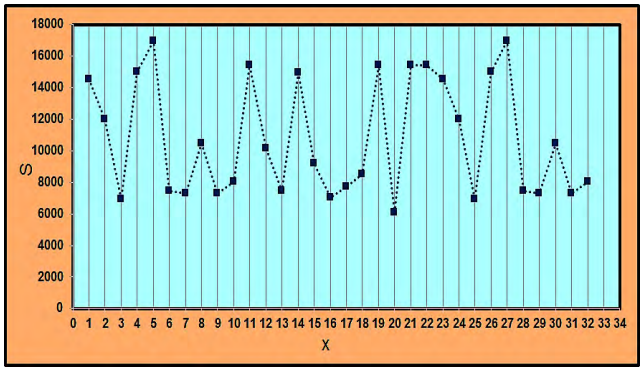


FIGURE 6. POPTA example: Speed function of a processor executing the multithreaded OpenBLAS DGEMM application represented by a discrete set of points (connected by dashed lines for clarity).

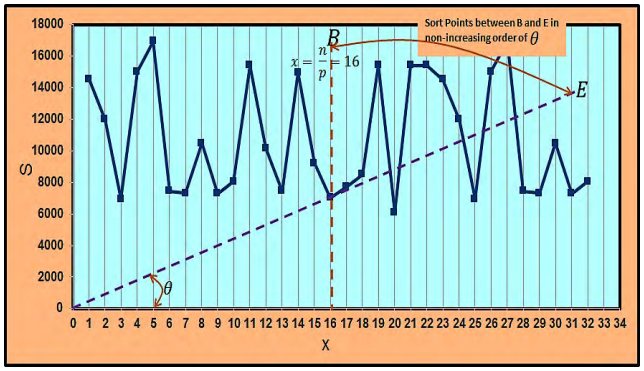


FIGURE 7. POPTA example: POPTA sorts points between B and E in non-increasing order of Θ.

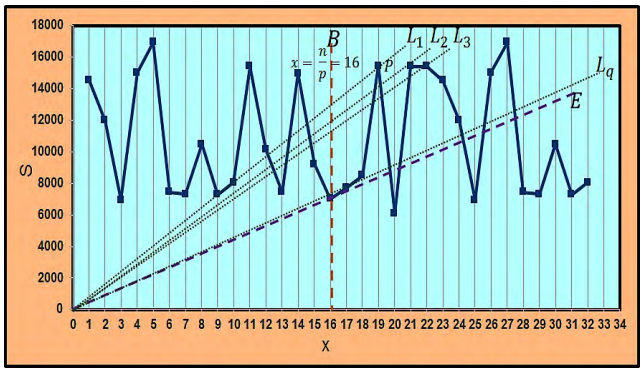


FIGURE 8. POPTA example: Points on line L_1 examined followed by points on line L_2 and so on until the points on line L_q.

optimal workload distribution, D_{opt}, where the distributions are given in multiples of Δx, and the optimal execution time, t_{opt}. The optimal number of processors that are selected by POPTA in the optimal workload distribution may be less than p. The traditional load-balancing algorithm returns the workload distribution, $x_i = \frac{n}{p}, \forall i \in [1, p]$.

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 6. The function is represented by discrete sets X and S containing the points in the graph connected by dashed lines for clarity. The recursive procedure, *POPTAKernel* (Algorithm 2), examines all the points between the lines B and E as shown in Figure 7. 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 POPTA 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 equal 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. The sorted arrays of points prevents recursion at points, which give optimal workload distribution for sub-problems.

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. A key optimization in POPTA is the 3D array, *memorized*, of size $O(m \times p^2)$, which memorizes the points that were visited already 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 8 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 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 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 *POPTA* 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 9. *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 9 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

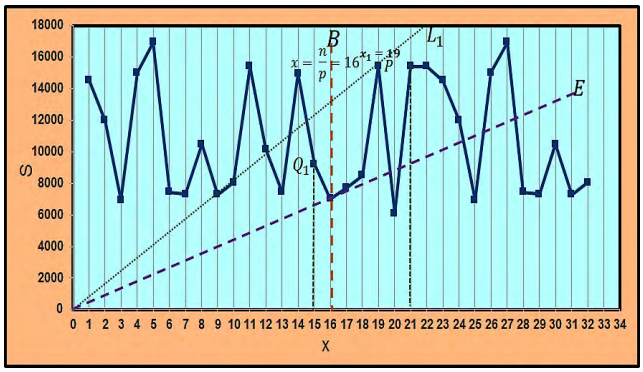


FIGURE 9. **POPTA example: One processor is allocated problem size 19 to the right. POPTA is invoked for remaining problem size 45 and 3 remaining processors.**

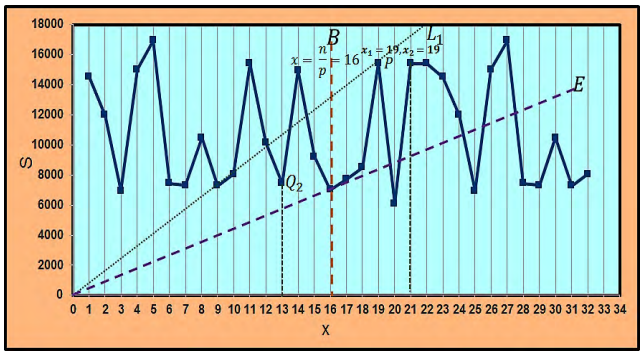


FIGURE 10. **POPTA example: Two processors are allocated problem size $P = 19$ each to the right. *POPTAkernel* is invoked to find optimal load distribution for remaining problem size 26 and 2 remaining processors.**

size $n_r = 19$ to the right shown by point P in Figure 10. *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 10 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 11. 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, POPTA considers the points on Line L_2. There is only one point P on this line as shown in Figure 12. 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, POPTA examines the points on lines L_1, L_2, L_3, and so on until the final line L_q (shown in Figure 13) before (and excluding) E.

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

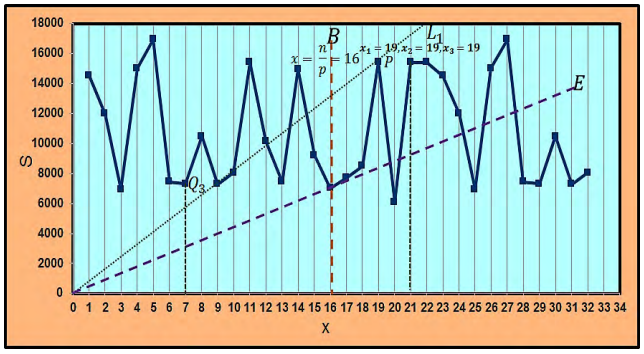


FIGURE 11. POPTA example: Three processors are allocated problem size $P = 19$ each to the right. The only remaining processor is allocated problem size $Q3 = 7$.

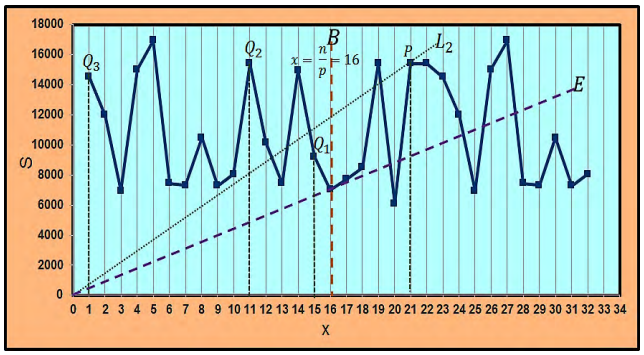


FIGURE 12. POPTA example: Point P on line L_2 examined. The corresponding allocations to the left are Q_1, Q_2, and Q_3.

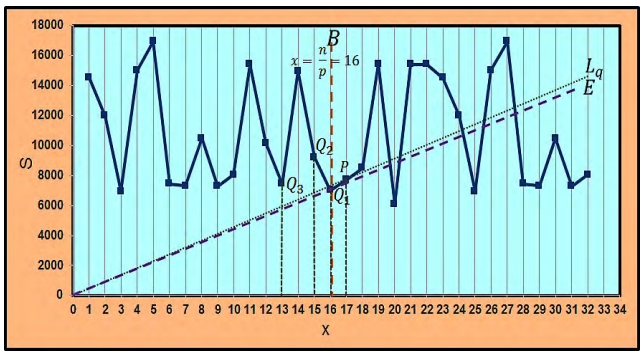


FIGURE 13. POPTA example: Point P on the final line L_q examined. The corresponding allocations to the left are Q_1, Q_2, and Q_3.

B. EOPTA: ALGORITHM SOLVING EOPT PROBLEM

The algorithm EOPTA solving EOPT has code structure similar to POPTA (Algorithm 1). The inputs to EOPTA 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 are the optimal load distribution, D_{opt}, and the optimal dynamic energy consumption, Ω_{opt}. The number of processors selected by EOPTA 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.

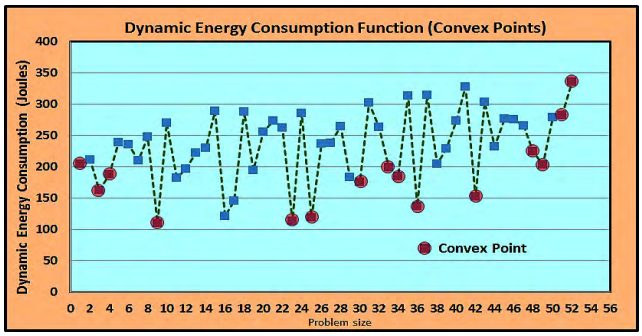


FIGURE 14. Convex points shown for a segment of dynamic energy consumption graph of OpenBLAS DGEMM application.

Unlike *POPTA*, which examines a subset of points in the speed function, *EOPTA* examines only the convex points in the energy function $(\mathcal{X}, \Psi)$. 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 14 shows these points in a segment of the energy function of OpenBLAS DGEMM application.

Similar to *POPTA*, a 3D array, *memorized*, of size $O(m \times p^2)$ is used to memorize the points that have been examined during the recursive invocations. 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 convex points in the energy function. While *POPTA* solves a *min-max* problem, *EOPTA* solves a *min-sum* problem.

The optimality and complexity proofs of POPTA and EOPTA are presented in [18].

V. *PARALEPH*: PARALLEL DATA PARTITIONING ALGORITHMS USING DYNAMIC PROGRAMMING

We examined the main loop in *POPTA* and *EOPTA* and observed two crucial aspects that formed the motivation for redesigning the algorithms using dynamic programming approach. First, the sequential algorithms employ *top-down* approach where they start evaluating points from $x = \frac{n}{p}$ onwards until $x = m$, which is the end point of discrete speed/dynamic energy functions. Second, a 3D array, *memoized*, is used to memorize the points that were visited already during the recursive invocations. The *top-down* strategy and memoization optimization guided us to design a solution that used *bottom-up* approach with memoization to reuse already saved work. Dynamic programming (DP) technique manifested as the perfect fit.

Once we designed the basic building blocks of a dynamic programming solution (recurrence relations, tabular computation, traceback), our next main objective was to transform the main loop so that table cells in the DP technique can be evaluated in parallel. To summarize, the loop containing

recursive invocations in the sequential algorithms is carefully restructured to allow parallel computation of the loop iterations.

We now present the parallel data partitioning algorithm *PARALEPH* employing the dynamic programming approach. The number of processors available for its execution is p, which is also the number of processors available for execution of the data-parallel application in which *PARALEPH* is applied. We illustrate *PARALEPH* using an implementation employing hierarchical two-level parallelism. The first parallelism is intra-process using OpenMP where each process executes t threads and the second parallelism is inter-process using q MPI processes.

Algorithm 3 shows the implementation details of *PARALEPH*.

The inputs to *PARALEPH* are the type of minimization problem represented by $\mathcal{F}$ where $\mathcal{F} = \max$ represents POPT and $\mathcal{F} = \sum$ represents EOPT, size of the workload n expressed as multiple of Δx, the number of available processors p executing the data-parallel application, the number of parallel processes q executing *PARALEPH*, the number of threads per process t executing *PARALEPH*, a function represented by two discrete sets, $\mathcal{X}$ and Ψ, where $\mathcal{X}$ contains problem sizes. When solving the POPT problem, the discrete set Ψ contains the execution times. For solving the EOPT problem, the discrete set Ψ contains the dynamic energy consumptions. m is the cardinality of the sets $\mathcal{X}$ and Ψ. It is assumed that the problem sizes in the discrete set $\mathcal{X}$ are separated by constant granularity, Δx.

The outputs are the optimal workload distribution, D_{opt}, and the optimal execution time or total dynamic energy consumption, Ω_{opt}. The optimal number of processors selected by *PARALEPH* in the optimal workload distribution may be less than p.

All the inputs are assumed to be available only at process 0. The outputs are also only available at process 0 after the termination of *PARALEPH*. There are two internal data structures employed in *PARALEPH*, which are distributed between the q processes. We assume p is divisible by q for the clarity of exposition. The dynamic programming table *dpt* is distributed between the q processes where each process stores $\frac{p}{q}$ columns of the table. The structure *tb* contains traceback pointers, which allows to reconstruct the optimal workload distribution once the tabular computation is completed. Each cell contains two such pointers. Therefore, there are $\frac{p}{q} \times 2$ pointers stored at each process.

Figures 15, 16 illustrate *PARALEPH*. Since *PARALEPH* uses dynamic programming approach, it has three core components: a). Recurrence relation, b). Tabular computation, and c). Traceback.

The recurrence relations are as follows:

$$
\begin{aligned}
dpt(1,h) &= \Omega(1), \quad \forall h \in [1,p] \\
dpt(v,1) &= \Omega(v), \quad \forall v \in [1,m] \\
dpt(v,1) &= \infty, \quad \forall v \in [m+1,\infty]
\end{aligned}
$$

Algorithm 3 Parallel Algorithm Determining Optimal Distribution of Workload of Size n for Maximizing Performance or Minimizing Dynamic Energy

1: **procedure** PARALEPH($\mathcal{F}, n, p, q, t, X, \Psi, D_{opt}, \Omega_{opt}$)

Input:

Objective to minimize, $\mathcal{F} \in \{\max, \sum\}$
Workload size, $n \in \mathbb{Z}_{>0}$
Number of available processors, $p \in \mathbb{Z}_{>0}$
Number of parallel processes executing *PARALEPH*, $q \in \mathbb{Z}_{>0}$
Number of threads executed by each process executing *PARALEPH*, $t \in \mathbb{Z}_{>0}$
Execution time or Energy function represented by two sets (X, Ψ),
$X = \{x_1, ..., x_m\}, x_1 < ... < x_m, x_i \in \mathbb{Z}_{>0}, \forall i \in [1, m]$
$\Psi = \{\Omega(x_1), ..., \Omega(x_m)\}, \Omega(x) \in \mathbb{R}_{>0}$

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 execution time or dynamic energy consumption,
$\Omega_{opt} \in \mathbb{R}_{>0}$

2: **if** $(p = 1)$ **then return** $(\{n\}, \Psi[n])$ **end if**
3: $me \leftarrow$ MPI_Comm_rank(MPI_COMM_WORLD)
4: **for** $h \leftarrow 1, \frac{p}{q}$ **do**
5: $dpt(1, h) \leftarrow \Psi[1]$
6: **end for**
7: **if** $me = 0$ **then**
8: **for** $v \leftarrow 2, m$ **do**
9: $dpt(v, 1) \leftarrow \Psi[v]$
10: **end for**
11: **end if**
12: $\#pragma\ \ omp\ \ parallel\ \ num\_threads(t)$
13: **for** $L \leftarrow 4, n + p$ **do**
14: $ncells \leftarrow$ getncells(n, p, L)
15: $(dpt, tb) \leftarrow$ computeL($\mathcal{F}, \Psi, me, n, p, q, m, L, ncells, dpt, tb$)
16: **end for**
17: **if** $me = 0$ **then**
18: $MPI\_Recv(\Omega_{opt}, 1, , q-1, ...)$
19: **end if**
20: **if** $me = (q-1)$ **then**
21: $MPI\_Send(\&dpt(n, \frac{p}{q}), 1, , 0, ...)$
22: **end if**
23: $D_{opt} \leftarrow$ Traceback$(n, p, p, q, q-1, tb)$
24: **return** (D_{opt}, Ω_{opt})
25: **end procedure**

$$
\begin{aligned}
dpt(v,h) = \min(&\mathcal{F}(dpt(I,1), dpt(v-I, h-1))), \\
&\mathcal{F}(dpt(v, h-1))) \\
&\forall I \in [1, \min(v-1, m)]
\end{aligned}
$$

where

$$
\begin{aligned}
\mathcal{F} &= \max, \quad \text{for solving POPT} \\
\mathcal{F} &= \sum, \quad \text{for solving EOPT}
\end{aligned}
$$

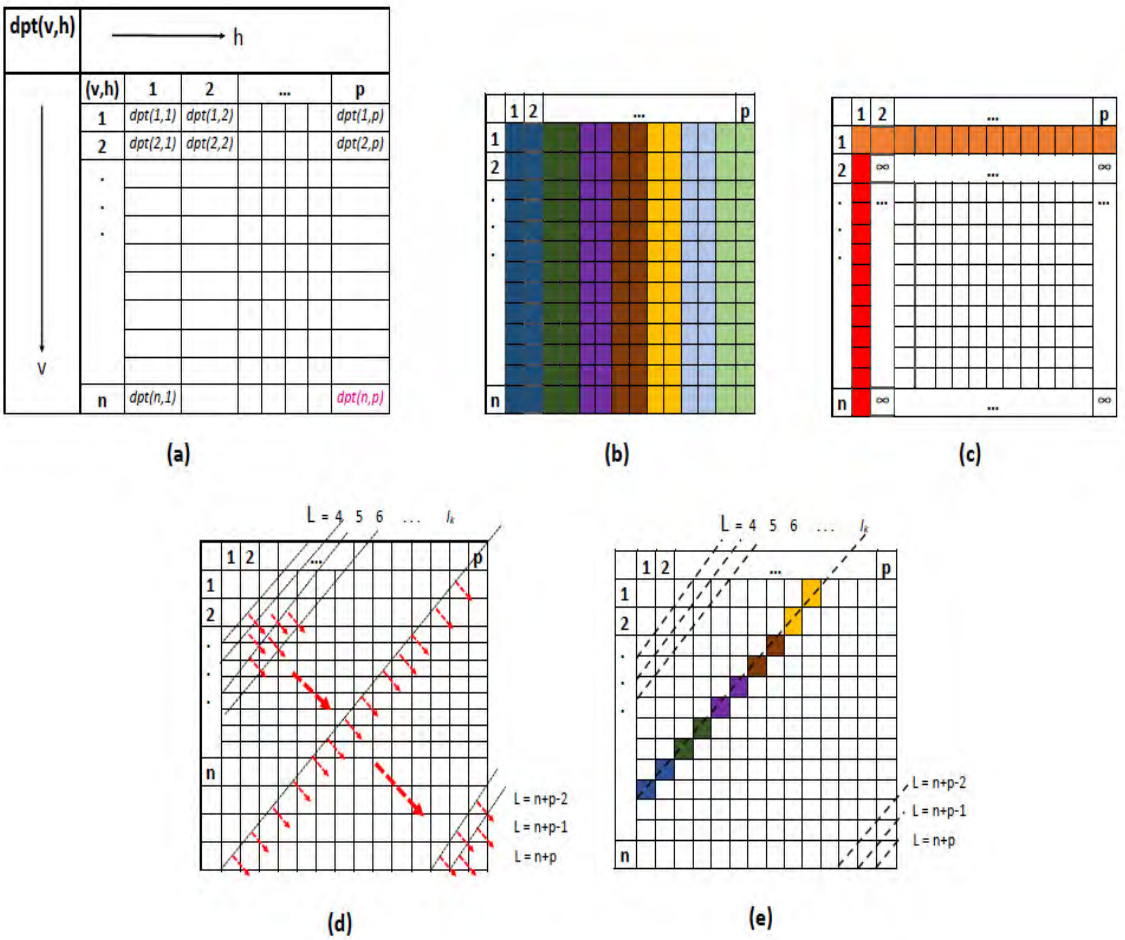


FIGURE 15. (a). The tabular computation in the dynamic programming approach in *PARALEPH*. The problem size n is a multiple of granularity, which is considered here to be 1 for effective illustration. (b). The distribution of the columns of the dynamic programming table amongst the processes, which are distinguished by colors. It is assumed that each process owns two columns each. (c). The base conditions in the dynamic programming approach. The orange cells in the first row, $dpt(1,h), \forall h \in [1,p]$, are initialized to $\Psi[1]$. The red cells in the first row, $dpt(v,1), \forall v \in [1,n]$, are initialized to $\Psi[v], \forall v \in [2,m]$. Rest of the cells are initialized to ∞. (d). The anti-diagonals are $L = 4, 5, \ldots, k, \ldots, n+p-2, n+p-1, n+p$. The cells are evaluated in the direction of the minor anti-diagonals as shown by the red arrows. The anti-diagonals are evaluated sequentially in this direction. (e). All the table cells in an anti-diagonal can be evaluated in parallel.

The relations are used to compute the values of cells in the dynamic programming table, dpt, as shown in the Figure 15(a). The table cell value $dpt(v,h), \forall v \in [1,n], h \in [1,p]$ contains the minimum execution time or total dynamic energy consumption to solve the workload of size v using h processors. At the end of execution of *PARALEPH*, the table cell value $dpt(n,p)$ contains the minimum execution time or total dynamic energy consumption to solve the workload of size n using p processors.

The base initialization consists of computing the values of the cells in the first row and first column as shown in the Figure 15(c). The value of each table cell in an anti-diagonal $L \in [4, n+p]$ depends only on the cells above the diagonal as shown in Figure 16(c). Hence these cells can be computed independently and in parallel. The anti-diagonals, which start from 4, are however executed sequentially. Consider the anti-diagonal, L=4. There are three cells, {(3,1),(2,2),(1,3)}, that need to be computed. Two cells {(3,1),(1,3)} have been computed in the base initialization. The value of the remaining cell, (2,2), depends only on the values of the cells above the diagonal, which are {(1,1),(1,2),(2,1)}. It is computed by the process highlighted in blue. For anti-diagonal L=5, there are four cells, {(4,1),(3,2),(2,3),(1,4)}, out of which two cells {(4,1),(1,4)} have been computed in the base initialization. Two processes compute the values of the cells (3,2) and (2,3) in parallel. Their values depend only on the values of the cells above the diagonal (L=5). So, in this manner, all the cells in the anti-diagonals $L \in [4, n+p]$ are computed.

We will now describe the pseudocode presented in the algorithm 3. To simplify the description, we assume that p is divisible by q. Line 2 deals with the simple case of solving the problem size n using one processor. A process identifies its rank, me, on Line 3. Lines 4-11 contain the base conditions of the recurrence relations. The base condition,

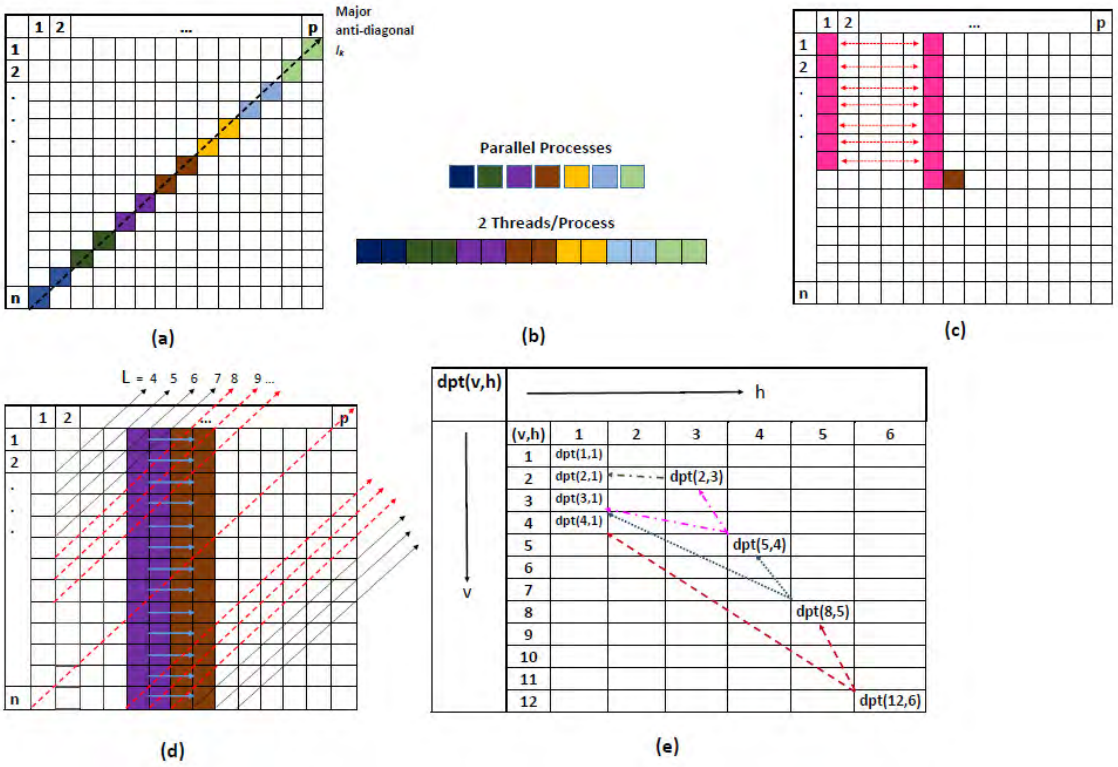


FIGURE 16. (a). All the table cells in the anti-diagonal l_k are evaluated in parallel by q parallel processes. In the figure, each process executes $t = 2$ threads and is responsible for computing 2 cells in an anti-diagonal. (b). The parallel processes are shown using different colors. (c). The value of a cell in an anti-diagonal, for example: the brown cell, is determined purely based on the values in the purple cells above the anti-diagonal. The arrow $i < -- > j$ shows that cells i and j are compared and max or sum is computed. (d). Communications between neighboring processes. For example: communications during the course of execution of *PARALEPH* between processes represented by purple and brown cells are shown as blue arrows. For each of the iterations of L represented by dotted red arrows, there is communication of only one cell between these two processes. (e). Traceback to reconstruct the optimal workload distribution. In the example, workload size $n = 12$ is solved using $p = 6$ available processors. *PARALEPH* is executed using $q = 3, t = 2$ configuration. The optimal workload distribution returned by *PARALEPH* is {2, 3, 3, 4}.

$dpt(1, h), \forall h \in [1, \frac{p}{q}]$, represents the objective function value solving a workload of size 1 using $h \in [1, \frac{p}{q}]$ processors. These cells are shown in orange color in the Figure 16(b). The base condition, $dpt(v, 1), \forall v \in [1, n]$, signifies the objective function value solving a workload of size $v \in [1, n]$ using 1 processor. These cells are shown in green color in the Figure 16(b). Since process 0 stores the first column, it initializes the cells in the first column.

Line 13 starts the core loop of *PARALEPH*. For every process t threads are created before this point (Line 12) and reused during the execution of *PARALEPH*. The loop variable $L = v+h$ goes from 4 until and including $n+p$. Each iteration represents an anti-diagonal shown in the Figures 15,16 and is computed using the routine, *computeL* (Algorithm 4). The number of cells in an anti-diagonal L, *ncells*, are computed using the routine, *getncells* (given in Appendix E-B). There can only be a maximum of p cells in any anti-diagonal. The routine, *getmyncells*, returns the number of cells, *myncells*, in an anti-diagonal that is owned by the process *me* and the number of cells, *ncellsbeforeme*, that precede the cells belonging to process *me*.

Lines 4-12 (Algorithm 4) present the communications in *PARALEPH*, which are illustrated in Figure 16(d). Just one cell value is communicated between neighboring processes i and $j = i + 1$ in each iteration of L. If the process $me \neq 0$ and it is computing a cell in its first *dpt* column given by the condition $(h - me \times \frac{p}{q} = 1)$, it needs the values of the cells in the *dpt* column preceding this column from neighboring process $me - 1$. The buffer *dptLeft* stores/accumulates these values with every increment of L. The index where to store the neighboring cell value is given by the variable, *leftcell*. Therefore, there is no need to communicate all the cells from the process $me - 1$ except for the last cell value, $dpt(myv, \frac{p}{q})$. If the process $me \neq (q - 1)$ and is not the last process owning a cell in the anti-diagonal L given by the condition $((ncellsbeforeme + myncells) < ncells)$, it sends the cell value, $dpt(myv, \frac{p}{q})$, to neighboring process, $me + 1$.

After the communications are completed, Line 13 (Algorithm 4) starts the core computations, which is that all the cells owned by a process, *myncells*, are computed independently by its t threads using *OpenMP pragma for*.

The routine, *computecell* (Algorithm 6), computes the table cell value $dpt(v, h)$. When solving the POPT problem, the $\mathcal{F}$ function is min operator. When solving the EOPT problem, the $\mathcal{F}$ function is $\sum$ operator. The value of the table cell, $dpt(v, h)$ depends only on the cells above the diagonal,

Algorithm 4 All Cells in the Minor Anti-Diagonal L are Computed in Parallel

```
1:  function computeL(F, Ψ, me,
            n, p, q, m, L, ncells, dpt, tb)
2:      start ← min(L − 2, n)
3:      (myncells, ncellsbeforeme) ← getmyncells(
            me, L, start, ncells, p, q)
4:      h ← L − start + ncellsbeforeme
5:      leftcell ← start − ncellsbeforeme
6:      #pragma omp for{
7:      if me ≠ 0 and h ≠ 2 and h − me × p/q = 1 then
8:          MPI_Recv(&dptLeft[leftcell],
             1, , me − 1, ...);
9:      end if
10:     if me ≠ q − 1 and
            (ncellsbeforeme + myncells) < ncells then
11:         myv ← start − myncells − ncellsbeforeme
12:         MPI_Send(&dpt(myv, p/q), 1, , me + 1, ...)
13:     end if
14:     }
15:     #pragma omp for
16:     for c ← 1, myncells do
17:         v ← start − ncellsbeforeme − c
18:         h ← L − start + ncellsbeforeme + c
19:         myh ← h − me × p/q
20:         (dpt(v, myh), tb(v, myh)) ←
21:             computecell(F, Ψ,
                    me, m, v, h, dptLeft, dpt, tb)
22:     end for
23:     return (dpt,tb)
24: end function
```

$\{(I, 1), (v - I, h - 1)\}, \forall I \in [1, v], (v, h - 1)$, as shown in Figure 15(c). *myh* represents the column number that is stored in a process. When $myh = 1$ and the process *me* is not 0, the array, *dptLeft*, contains the cells that have been received by the process *me* from process $me-1$ before the computation of the cell, $dpt(v, h)$.

Lines 9-19 shows the invocation of binary operator $\mathcal{F}$ on cells $(I, 1)$ and $(v-I, myh-1)$ as shown in the Figure 16(c). For the computation of the cell $dpt(v, h)$, there are $min(m, v-1)$ evaluations of the operator $\mathcal{F}$, where m is the cardinality of the discrete sets representing the speed/energy functions. Therefore, there are a maximum of m evaluations of the operator $\mathcal{F}$ for the table cell $dpt(n, p)$.

At the end of the execution of *PARALEPH*, the value in the cell $dpt(n, p)$ is sent by the process $q - 1$ to process 0. It contains the minimum execution time or total dynamic energy consumption to solve the workload of size n using p processors.

Once the tabular computation is completed, traceback of pointers as shown in Figure 16(e) is used to reconstruct and return the optimal workload distribution. Line 23, Algorithm 3, invokes the traceback routine. The traceback algorithm is provided in Appendix E-D. In the Figure 16(e), workload size $n = 12$ is solved using $q = 3$ processes, each process executing $t = 2$ threads. The optimal execution time for the table cell $dpt(12, 6)$ is composed from $(dpt(4, 1), dpt(8, 5))$. The optimal execution time for $dpt(8, 5)$ is derived from $(dpt(3, 1), dpt(5, 4))$. The optimal execution time for $dpt(5, 4)$ is derived from $(dpt(3, 1), dpt(2, 3))$. Finally, the optimal value for the table cell $dpt(2, 3)$ is composed from $dpt(2, 1)$. Therefore, the optimal workload distribution is $\{2, 3, 3, 4\}$.

A. OPTIMALITY PROOF OF PARALEPH

Proposition 5.1: Let Δx be the minimum granularity of workload so that each processor is allocated a multiple of Δx only. Let the execution time function of a processor, Ψ, be represented by a discrete set of experimental points separated by Δx. Then *PARALEPH* solves the *POPT* problem.

Proof: The optimal workload distribution is a combination of cells in the first column, $(v, 1), \forall v \in [1, m]$.

Assume that the solution provided by *PARALEPH* has the workload distribution, $(k_1, k_2, ...k_u)$, $\Psi(k_1) > \Psi(k_2) > ... > \Psi(k_u)$, $\sum_{i=1}^{u}(k_i) = n$. We need to show that this workload distribution is optimal. That is, it gives the optimal execution time, e_u.

We demonstrate using proof by contradiction that a workload distribution, $(l_1, l_2, ...l_w)$, $\Psi(l_1) > \Psi(l_2) > ... > \Psi(l_w)$, $w < u$, $\sum_{i=1}^{w}(l_i) = n$, which is assumed to give execution time, $e_{opt} < e_u$, does not exist. We assume that $k_1 \neq l_1, ..., k_w \neq l_w$. From our hypothesis, we can assume without any loss of generality, $\Psi(l_1) < \Psi(k_1)$, $\Psi(l_2) < \Psi(k_2)$, ..., $\Psi(l_w) < \Psi(k_w)$.

Since $\Psi(l_1) < \Psi(k_1)$, the workload distribution for $(n - k_1, p-1)$, which is equal to $(k_2, ...k_u)$, must have an execution time $dpt(n-k_1, p-1)$ greater than the execution time, $dpt(n-l_1, p-1)$, given by the workload distribution, $(l_2, ...l_w)$. This is because *PARALEPH* compares $dpt(k_1, 1)$ with $dpt(n-k_1, p-1)$ and $dpt(l_1, 1)$ with $dpt(n-l_1, p-1)$ and picks the minimum of the two maximums. Otherwise, it would have picked the combination $(l_1, 1)$ and $(n - l_1, p - 1)$ instead.

Since $\Psi(l_2) < \Psi(k_2)$, the workload distribution for $(n - k_1 - k_2, p-2)$, which is equal to $(k_3, ...k_u)$, must have an execution time $dpt(n - k_1 - k_2, p - 2)$ greater than the execution time, $dpt(n-l_1-l_2, p-2)$, given by the workload distribution, $(l_3, ...l_w)$. Proceeding in this manner, we come to the case where since $\Psi(l_{w-1}) < \Psi(k_{w-1})$, the workload distribution for $(n - \sum_{i=1}^{w-1}(k_i), 1)$, which is equal to $(k_w, ..., k_u)$, must have an execution time $dpt(n - \sum_{i=1}^{w-1}(k_i), 1)$ greater than the execution time, $dpt(n - \sum_{i=1}^{w-1}(l_i), 1)$, given by the workload distribution, (l_w). It is however not possible that $\Psi(k_{w-1}) > \Psi(l_{w-1})$ and $\Psi(k_w) > \Psi(l_w)$ since in this case *PARALEPH* would have picked (l_{w-1}, l_w) instead of $(k_{w-1}, k_w, ..., k_u)$.

Since *PARALEPH* picked $(k_{w-1}, k_w, ..., k_u)$, the maximum of $(k_{w-1}, k_w, ..., k_u)$ must be less than the maximum of (l_{w-1}, l_w). This means that $\Psi(k_{w-1}) < \Psi(l_{w-1})$ or $\Psi(k_{w-1}) = \Psi(l_{w-1})$ and $\Psi(k_w) \leq \Psi(l_w)$. The second condition is ruled out because of our assumptions.

Working bottom-up now, $\Psi(k_{w-2}) < \Psi(l_{w-2})$ since otherwise *PARALEPH* would select $(l_{w-2}, k_{w-1}, ..., k_u)$ instead of $(k_{w-2}, k_{w-1}, ..., k_u)$. So, in this manner, we proceed to show that $\Psi(k_1) < \Psi(l_1)$ contradicting our hypothesis. □

The proof that *PARALEPH* solves *EOPT* follows on similar lines.

B. COMPUTATIONAL COMPLEXITY OF PARALEPH

Proposition 5.2: The time complexity of *PARALEPH* is $O(m^2 \times p)$, where n is the workload size expressed as a multiple of granularity, p is the number of available processors in the execution of *PARALEPH*, and m is the cardinality of the discrete sets representing the speed/energy functions.

Proof: We will first compute the complexity of computations.

The base initialization of the table *dpt*, which consists of evaluation of first row and first column in the table, can be accomplished in $O(n + p)$ arithmetic operations.

There are $n+p-4 \approxeq O(n+p)$ anti-diagonals in the tabular computation (Algorithm *PARALEPH*, Line 13). There can only be a maximum of p cells in an anti-diagonal. The number of cells increase from 1 to p and then decrease to 1 with increasing anti-diagonal index. We are excluding the cells in the first row and first column, which have been initialized.

All the cells in an anti-diagonal are computed in parallel by p processors. The number of evaluations of operator $\mathcal{F}$ is bounded by $O(m)$ in the case of table cell $dpt(n, p)$ (Algorithm *ComputeCell*, Lines 9-19). So, the total complexity is equal to $O(n + p) \times O(m) \approx O(m \times (n + p))$.

The traceback of the pointers to reconstruct the optimal data distribution can be accomplished in $O(p)$ arithmetic operations.

The total complexity of *PARALEPH* is equal to $O(n+p)+O(m \times (n + p)) + O(p) \approx O(m \times (n + p))$.

Since $n \approxeq O(m \times p)$ and $O(m \times p + p) \approxeq O(m \times p)$, the complexity of computations becomes $O(m^2 \times p)$.

Since the communication complexity is bounded by $O(m^2 \times p)$, the time complexity is $O(m^2 \times p)$. □

Let us estimate the cost of communications where we assume *PARALEPH* is executed by q parallel processes ($q > 1$). Assume the cost of communication of a cell (a double precision floating-point number) in the *dpt* table between two processors is represented by $\alpha + \beta \times 8$ using Hockney model where α is the latency for every message and β is the reciprocal of network bandwidth. During the execution of *PARALEPH*, in a iteration of L, process me, $\forall me \in [0, q-2]$ sends one element in its last local *dpt* column to the neighboring process $me+1$. So, the total number of cells it sends is equal to the number of elements in this column, which is n. Therefore, the total cost of communications of sending these cells is $(\alpha+\beta\times8)\times n$. Since the total number of processes involved in the communications is $q - 1$, the communication complexity of *PARALEPH* is equal to $(\alpha+\beta\times8)\times n\times(q-1) \approxeq O(n\times q)$. The communication complexity of *PARALEPH* is therefore $(\alpha + \beta \times 8) \times (n \times q)$. Since $n \approxeq O(m \times p)$ and $q \approxeq O(p)$, the complexity remains $O(m^2 \times p)$.

The memory complexity of *PARALEPH* is $O(n)$, where n is the workload size expressed as a multiple of granularity. The DP table *dpt* of size $n \times p$ is distributed between the p processors where each processor stores one column of the table. Each processor also stores two traceback pointers per cell. Therefore, the memory complexity per processor is just $O(n)$.

Therefore, we can conclude that the potential speedup delivered by the parallel algorithms is $O(p)$, which is the case for embarassingly-parallel algorithms.

C. INCORPORATION OF COST OF COMMUNICATIONS

We present how the cost of communications during the execution of a data-parallel application employing our data partitioning algorithm can be seamlessly integrated in *PARALEPH*.

The core idea is the use of analytical approach to estimate the cost of communications. Rico-Gallego *et al.* [52] formulate and implement this idea where they find the optimal communication scheme without expensive testing on the executing platform to estimate the communication cost of different configurations of the application. They propose and discuss an extension of the τ-Lop communication performance model to cover heterogeneous architectures. So, to summarize, there are now two performance models, which are input to our data partitioning algorithm that optimizes data-parallel application for performance. They are an experimentally constructed computation performance model and a communication performance model employing analytical formulas. Energy models for communications is a open research problem and hence we do not consider our data partitioning algorithm for energy optimization in this discussion.

As described already, *PARALEPH* uses Dynamic Programming technique (DP) where the execution time of computations for two cells (n_1, p_1) and (n_2, p_2) are compared in each step of the algorithm, L, in the function *ComputeCell* to solve a problem size $(n_1 + n_2, p)$ where $p_1 + p_2 <= p$. To take into account the cost of communications, we add a simple extension to the function, *ComputeCell*, in the algorithm.

To simplify the exposition, we will consider parallel matrix multiplication application employing *SUMMA* algorithm [53]. We will use an analytical model (Hockney) to parameterize the cost of communications, $(C_{n,p})$, to solve a problem size n using p processors. The execution time of communications considering that SUMMA uses scatter-allgather broadcast is estimated using the Hockney model [54]. Scatter is implemented using a binomial tree and allgather, a ring algorithm in which the data from each processor are sent around a virtual ring of processors in $p - 1$ steps.

$$\begin{aligned} C(n,p) &= T^{comm}_{SUMMA}(N/1024, p, b, \alpha, \beta) \\ &= (\log_2 p + 2 \times (\sqrt{p} - 1)) \times \alpha \times \frac{N}{b} \\ &\quad + 4 \times (1 - \frac{1}{\sqrt{p}}) \times \beta \times \frac{N^2}{\sqrt{p}} \end{aligned}$$

where $N = n \times 1024$, b is the block size, α is the latency, and β is the reciprocal of network bandwidth. α and β are properties of the communication network that are experimentally obtained.

Therefore, when two cells, (n_1, p_1) and (n_2, p_2), are compared, we compare the total execution times, which are obtained by summing the time of computations (from the functional performance model) and the cost of communications estimated from the analytical communication performance model.

The computational complexity of *PARALEPH* remains the same since the computation of the cost of communications takes a constant number of arithmetic operations.

D. GUIDELINES FOR APPLICATION OF PARALEPH

In this section, we present a brief on how *PARALEPH* can be applied practically. The optimal values of the two parameters, (q, t), are typically found via experimentation. One important constraint is, $q \times t \leq p$, to prevent over-subscription of resources. If the data-parallel application is executed in one node and employs p threads, *PARALEPH* can be executed by q processes each executing t threads such that $q \times t \leq p$. If $q = 1$, then only OpenMP is used in the execution of *PARALEPH* and the cells in the anti-diagonals are divided equally between the t threads using the *OpenMP pragma for*. If $t = 1$, then *PARALEPH* can be executed using q MPI processes ($q \leq p$) where the columns of the *dpt* table are divided equally between the q processes. It is however recommended that *PARALEPH* be executed using p threads where one thread is mapped to one core.

Consider a data-parallel application which is executed in a cluster of p nodes and employs p processes where one process is mapped to one node and each process employs t threads where one thread is mapped to one core in a node. For such an application, one can choose to execute *PARALEPH* using either $q = p$ processes where each process executes $t = 1$ thread or q processes where each process executes t threads such that $q \times t = p$.

E. APPLICATION DOMAINS FOR PARALEPH

In Appendix F, we provide examples of classes of applications where our data partitioning algorithms can be employed. From our experience, there are multiple classes of applications that benefit from our data partitioning algorithms. A dominant class contains applications where the speed of an application is a function of problem size, which is defined as a set of one, two or more parameters characterizing the amount and layout of data stored and processed during the execution of a computational task.

The research focus in this work is focused entirely on traditional HPC and does not target data analytical or Big Data applications such as MapReduce, Hadoop, Spark, etc. We believe that MapReduce like applications are more applicable to domains that are highly data-driven and not compute-driven in the sense that the ratio of in-memory compute times to the data processing times (due to specialized storage in filesystems such as HDFS, etc) is low. Our research work, however, is mainly directed towards scientific applications with high in-memory computational complexity and, therefore, the most important concern is to reduce this complexity.

VI. EXPERIMENTAL RESULTS AND DISCUSSION

In this section, we demonstrate the practical cost of applying *PARALEPH* in two data-parallel applications, OpenBLAS DGEMM [15] and FFTW [17]. We also demonstrate the tremendous speedups of the parallel algorithms over the sequential algorithms. We conclude with theoretical analysis of *PARALEPH* using parallel matrix-matrix multiplication on extreme-scale platforms.

A. EXPERIMENTAL ANALYSIS OF PARALEPH

We demonstrate the low runtime cost of *PARALEPH* using experiments performed in the Grid'5000 platform hosted in France (http://www.grid5000.fr). The platform contains 24 clusters distributed over 10 sites (nine in France and one in Luxembourg), which includes 1006 nodes, 8014 cores. We used the Graphene cluster in Nancy site for our experiments. We used a total of 576 cores from 144 nodes. Each node has a disk of 298 GB storage, 16 GB of memory, and a quad-core Intel Xeon X3440 CPU. The nodes in the cluster are interconnected via 20 Gb/s Infiniband. For the MPI communications, OpenMPI-1.6.5 is used. *gcc* compiler version used for compilation is 4.9.2.

Before we present the comparison of execution times of the data-parallel applications and the data partitioning algorithms, we present a brief on how the execution time and energy functions are built since these are input to the data partitioning algorithms. The execution times of the algorithms do not include the cost of building these functions since they are assumed to be the inputs.

The execution time and the energy functions are built separately experimentally using an automated build procedure using 144 parallel processes where one process is mapped to one node. To make sure the experimental results are reliable, an experimental methodology described in detail in the Appendix D is used. The inputs to the automation procedure are the application and application parameters (problem size, number of threads, etc), range of problem sizes, and granularity. To obtain a data point for each function, the software executes the application repeatedly until the sample mean lies in the 95% confidence interval with precision of 0.025 (2.5%). For this purpose, we use Student's t-test. The software outputs a set of points, which represents the discrete function. The total dynamic energy consumption during the application execution is obtained using Watts Up Pro power meter.

The cardinality m of the discrete sets representing the execution time and dynamic energy functions is chosen to be 1024. So, the first 144 points are built experimentally in parallel by the 144 parallel processes, the next 144 points are built experimentally in parallel by the same processes again, and so on. The granularity (Δx), separating the points in the functions, is 524288 representing DGEMM of a

TABLE 4. Execution times of *PARALEPH* and DGEMM in seconds. *PARALEPH* solves *POPT*. p in first column. $\frac{n}{p}$ in first row. $\frac{n}{p} \in \{8, 16, 32\}$. Each cell in the table contains the execution time of the data-parallel application, the execution time of *PARALEPH*, the time of communications in *PARALEPH*, and the speedup of *PARALEPH* over the sequential data-partitioning algorithms. 'F' indicates failure of the sequential algorithm.

$p \downarrow, \frac{n}{p} \rightarrow$	8	16	32
2	(10, 0.0004 , 7e-06, 2276)	(32, 0.0004 , 2e-06, 2308)	(74, 0.0004 , 1.3e-05, 1799)
4	(16, 0.0001 , 0.0005, 1559)	(37, 0.001 , 0.0005, 704)	(111, 0.001 , 0.00058, 95)
16	(44, 0.04 , 0.023, 101)	(108, 0.096 , 0.024, 39)	(350, 0.16 , 0.054, 21)
36	(121, 0.025 , 0.0087, 49)	(269, 0.16 , 0.053, 21)	(738, 0.037 , 0.026, 18)
64	(304, 0.16 , 0.037, 6)	(613, 0.15 , 0.062, 221)	(1394, 0.27 , 0.1, 149)
100	(405, 0.054 , 0.017, 97)	(872, 0.1 , 0.064, 984)	(2295, 0.097 , 0.045, 986)
144	(648, 0.25 , 0.078, 298)	(1342, 0.25 , 0.12, 2139)	(3325, 0.17 , 0.09, 3330)
196	(948, 0.22 , 0.15, 881)	(1870, 0.2 , 0.11, 7240)	(4559, 0.3 , 0.18, 8416)
256	(1280, 0.27 , 0.1, 2223)	(2723, 0.22 , 0.1, **F**)	(6224, 0.29 , 0.14, **F**)
324	(1609, 0.3 , 0.15, 2697)	(3230, 0.32 , 0.16, **F**)	(7956, 0.48 , 0.29, **F**)
400	(2093, 0.3 , 0.12, 6056)	(4202, 0.37 , 0.17, **F**)	(9904, 0.49 , 0.24, **F**)
484	(2485, 0.39 , 0.16, **F**)	(4944, 0.5 , 0.24, **F**)	(12066, 0.55 , 0.25, **F**)
576	(2975, 0.46 , 0.16, **F**)	(5952, 0.55 , 0.23, **F**)	(14443, 0.72 , 0.32, **F**)

TABLE 5. Execution times of *PARALEPH* and DGEMM in seconds. $\frac{n}{p} \in \{64, 128\}$. Each cell in the table contains the execution time of the data-parallel application, the execution time of *PARALEPH*, the time of communications in *PARALEPH*, and the speedup of *PARALEPH* over the sequential data-partitioning algorithms. 'F' indicates failure of the sequential algorithm.

$p \downarrow, \frac{n}{p} \rightarrow$	64	128
2	(213, 0.0004 , 7e-06, 1116)	(590, 0.0007 , 1.8e-05, 530)
4	(295, 0.002 , 0.0011, 174)	(948, 0.0036 , 0.0023, 98)
16	(722, 0.046 , 0.036, 11)	(2126, 0.027 , 0.016, 3)
36	(1468, 0.15 , 0.11, 113)	(3245, 0.08 , 0.049, 270)
64	(3166, 0.23 , 0.15, 679)	(4955, 0.23 , 0.15, 1366)
100	(4214, 0.24 , 0.17, 2655)	(6541, 0.26 , 0.15, 6033)
144	(6382, 0.29 , 0.17, 6385)	(8461, 0.42 , 0.26, **F**)
196	(8466, 0.37 , 0.22, **F**)	(10496, 0.52 , 0.29, **F**)
256	(10451, 0.44 , 0.25, **F**)	(12846, 0.73 , 0.44, **F**)
324	(13196, 0.65 , 0.39, **F**)	(15411, 1.1 , 0.7, **F**)
400	(16262, 0.7 , 0.38, **F**)	(18190, 1.1 , 0.64, **F**)
484	(19651, 0.91 , 0.51, **F**)	(22184, 1.7 , 0.91, **F**)
576	(23362, 1.1 , 0.6, **F**)	(25392, 1.8 , 0.99, **F**)

512×512 matrix and 2D FFT of 512×512. Lesser granularity would unveil larger fluctuations but would also mean more experimental points thereby increasing the time to build the speed and energy functions. As the granularity increases, the functional models become smooth and will resemble those for uniprocessors therefore disallowing any opportunity for optimization. We observed, however, that the variations are drastic for this particular granularity compared to other granularities. The execution times of building the FPMs in parallel using the 144 nodes in Graphene in Grid'5000 for OpenBLAS DGEMM and FFTW are 4900 seconds and 63 seconds.

The applications are executed for different values of $p \in \{2, 4, 16, 36, 64, 100, 144, 196, 256, 324, 400, 484, 576\}$ and granularities (problem size per processor) $\frac{n}{p} \in \{8, 16, 32, 64, 128\}$. Although each node is oversubscribed for values of $p > 144$, we show the results just to emphasize that sequential algorithms fail for large values of n and p due to high memoization cost while parallel algorithms do not exhibit any performance degradation or failures. We do not have out-of-core implementations for the sequential algorithms to compare against parallel algorithms for such values of n and p. This we hope to address in our future work.

The parallel matrix-matrix application is based on SUMMA [53] and employs heterogeneous two-dimensional block-cyclic distribution of matrices [53], [55]. In this application, the square matrices A, B, and C of size $(n \times \Delta x) \times (n \times \Delta x)$ are distributed over a two-dimensional arrangement of processors, $p_1 \times p_2, p_1 = \sqrt{p}, p_2 = \frac{p}{p_1}$. The local computations are performed using the DGEMM routine from the optimized OpenBLAS2.18 library.

PARALEPH is executed on just one node for values of $p < 36$. For larger values of p, it is executed using p parallel processes each executing 1 OpenMP thread. We have

TABLE 6. Execution times of *PARALEPH* and DGEMM in seconds. *PARALEPH* solves *EOPT*. $\frac{n}{p} \in \{8, 16, 32\}$. Each cell in the table contains the execution time of the data-parallel application, the execution time of *PARALEPH*, the time of communications in *PARALEPH*, and the speedup of *PARALEPH* over the sequential data-partitioning algorithms. 'F' indicates failure of the sequential algorithm.

$p \downarrow, \frac{n}{p} \rightarrow$	8	16	32
2	(10, 0.00036 , 7e-06, 3.5)	(32, 0.0003 , 2e-06, 3.0)	(74, 0.0004 , 2e-06, 2.6)
4	(16, 0.001 , 0.0005, 2.2)	(37, 0.001 , 0.0005, 1.2)	(111, 0.001 , 0.00059, 2.5)
16	(44, 0.018 , 0.009, 1.05)	(108, 0.13 , 0.036, 1.2)	(350, 0.11 , 0.033, 2.5)
36	(121, 0.031 , 0.018, 1.03)	(269, 0.12 , 0.034, 2)	(738, 0.024 , 0.012, 2.5)
64	(304, 0.077 , 0.023, 3)	(613, 0.26 , 0.085, 10)	(1394, 0.15 , 0.097, 12.6)
100	(405, 0.059 , 0.024, 11)	(872, 0.084 , 0.047, 38)	(2295, 0.15 , 0.074, 60)
144	(648, 0.16 , 0.038, 48)	(1342, 0.099 , 0.036, 137)	(3325, 0.15 , 0.076, 227)
196	(948, 0.17 , 0.096, 133)	(1870, 0.2 , 0.12, 291)	(4559, 0.34 , 0.2, 495)
256	(1280, 0.17 , 0.05, 272)	(2723, 0.25 , 0.12, 572)	(6224, 0.29 , 0.14, 708)
324	(1609, 0.3 , 0.17, 485)	(3230, 0.34 , 0.16, 1154)	(7956, 0.43 , 0.24, 1802)
400	(2093, 0.36 , 0.13, 796)	(4202, 0.43 , 0.22, 1939)	(9904, 0.47 , 0.23, 3116)
484	(2485, 0.61 , 0.38, 1682)	(4944, 0.5 , 0.17, 3387)	(12066, 0.6 , 0.29, **F**)
576	(2975, 0.52 , 0.23, 914)	(5952, 0.58 , 0.25, **F**)	(14443, 0.7 , 0.3, **F**)

TABLE 7. Execution times of *PARALEPH* and DGEMM in seconds. *PARALEPH* solves *EOPT*. $\frac{n}{p} \in \{64, 128\}$. Each cell in the table contains the execution time of the data-parallel application, the execution time of *PARALEPH*, the time of communications in *PARALEPH*, and the speedup of *PARALEPH* over the sequential data-partitioning algorithms. 'F' indicates failure of the sequential algorithm.

$p \downarrow, \frac{n}{p} \rightarrow$	64	128
2	(213, 0.0004 , 9e-06, 1.9)	(590, 0.0007 , 1.1e-05, 1.3)
4	(295, 0.002 , 0.0011, 1.9)	(948, 0.003 , 0.002, 1.3)
16	(722, 0.16 , 0.072, 1.9)	(2126, 0.046 , 0.02, 1.3)
36	(1468, 0.063 , 0.039, 3.7)	(3245, 0.085 , 0.04, 6)
64	(3166, 0.099 , 0.058, 18.7)	(4955, 0.16 , 0.084, 48)
100	(4214, 0.26 , 0.18, 81.3)	(6541, 0.26 , 0.15, 159)
144	(6382, 0.3 , 0.18, 345.6)	(8461, 0.43 , 0.26, 490)
196	(8466, 0.38 , 0.24, 755.2)	(10496, 0.6 , 0.37, 1161)
256	(10451, 0.56 , 0.34, 1334)	(12846, 0.74 , 0.43, 1734)
324	(13196, 0.59 , 0.33, 2703)	(15411, 1 , 0.61, **F**)
400	(16262, 0.74 , 0.4, **F**)	(18190, 1.3 , 0.78, **F**)
484	(19651, 0.9 , 0.48, **F**)	(22184, 1.5 , 0.83, **F**)
576	(23362, 1.3 , 0.73, **F**)	(25392, 1.9 , 1.1, **F**)

found this configuration of *PARALEPH* to be the optimal (Appendix G). The sequential data-partitioning algorithms (*POPTA*, *EOPTA*) are executed using one core in one single node.

We verified experimentally that the optimal solutions returned by the sequential and parallel data partitioning algorithms (*POPTA*, *EOPTA*, *PARALEPH*) are the same.

Tables 4, 5 and 6, 7 shows the execution times of *PARALEPH* in DGEMM solving optimization problems for performance and energy. Tables 8, 9 and 10, 11 shows the execution times of *PARALEPH* in FFTW solving optimization problems for performance and energy.

Each cell in the tables contains four values, (t_{app}, $t_{PARALEPH}$, t_{comm}, *speedup*), where t_{app} is the execution time of the data-parallel application, $t_{PARALEPH}$ is the execution time of *PARALEPH*, t_{comm} is the time of communications in *PARALEPH*, and *speedup* is the speedup of *PARALEPH* over the sequential data-partitioning algorithms (*POPTA* or *EOPTA*). The speedup is the ratio of execution time of the sequential data-partitioning algorithm over the execution time of *PARALEPH*.

We can conclude from the tables that the execution times of *PARALEPH* are negligible compared to the execution times of the applications. *PARALEPH* gives tremendous speedups over the sequential algorithms. For large values of n and p, the sequential algorithms fail due to their high memory cost indicated by *F* in the tables.

B. THEORETICAL ANALYSIS OF PARALEPH FOR EXTREME SCALE PLATFORMS

We consider the execution of parallel matrix-matrix multiplication application based on SUMMA [53] and employing *PARALEPH* on a theoretical exascale platform published in

TABLE 8. Execution times of *PARALEPH* and FFTW in seconds. *PARALEPH* solves *POPT*. $\frac{n}{p} \in \{8, 16, 32\}$. Each cell in the table contains the execution time of the data-parallel application, the execution time of *PARALEPH*, the time of communications in *PARALEPH*, and the speedup of *PARALEPH* over the sequential data-partitioning algorithms. 'F' indicates failure of the sequential algorithm.

$p \downarrow, \frac{n}{p} \rightarrow$	8	16	32
2	(1.5, 9.5e-05 , 6e-06, 619)	(3.4, 0.0001 , 7e-06, 655)	(7.4, 0.0001 , 3e-06, 502)
4	(2.8, 0.0003 , 0.0005, 435)	(5, 0.0003 , 0.0005, 318)	(9.8, 0.00047 , 0.0006, 57)
16	(13, 0.0062 , 0.018, 42)	(27, 0.02 , 0.024, 17)	(37, 0.005 , 0.002, 3)
36	(31, 0.011 , 0.023, 14)	(61, 0.014 , 0.024, 4.4)	(66, 0.018 , 0.021, 61)
64	(39, 0.024 , 0.026, 154)	(63, 0.039 , 0.032, 136)	(80, 0.057 , 0.099, 147)
100	(39, 0.032 , 0.016, 397)	(78, 0.061 , 0.12, 231)	(126, 0.073 , 0.1, 945)
144	(39, 0.051 , 0.02, 475)	(81, 0.065 , 0.032, 911)	(153, 0.11 , 0.12, 3117)
196	(41, 0.09 , 0.12, 1665)	(80, 0.1 , 0.1, 4062)	(201, 0.14 , 0.13, 10457)
256	(83, 0.13 , 0.1, 4433)	(163, 0.32 , 0.12, 6542)	(246, 0.37 , 0.21, **F**)
324	(105, 0.25 , 0.16, 6464)	(195, 0.25 , 0.064, **F**)	(301, 0.38 , 0.2, **F**)
400	(125, 0.34 , 0.056, **F**)	(228, 0.39 , 0.2, **F**)	(347, 0.51 , 0.21, **F**)
484	(139, 0.4 , 0.1, **F**)	(263, 0.48 , 0.19, **F**)	(401, 0.6 , 0.24, **F**)
576	(154, 0.52 , 0.19, **F**)	(273, 0.45 , 0.2, **F**)	(550, 0.59 , 0.19, **F**)

TABLE 9. Execution times of *PARALEPH* and FFTW in seconds. *PARALEPH* solves *POPT*. $\frac{n}{p} \in \{64, 128\}$. Each cell in the table contains the execution time of the data-parallel application, the execution time of *PARALEPH*, the time of communications in *PARALEPH*, and the speedup of *PARALEPH* over the sequential data-partitioning algorithms. 'F' indicates failure of the sequential algorithm.

$p \downarrow, \frac{n}{p} \rightarrow$	64	128
2	(6, 0.0002 , 9e-06, 328)	(37, 0.0004 , 9e-06, 144)
4	(7, 0.00083 , 0.0012, 0.9)	(54, 0.002 , 0.002, 23)
16	(93, 0.037 , 0.049, 2.9)	(125, 0.04 , 0.033, 23)
36	(196, 0.034 , 0.049, 59)	(246, 0.05 , 0.05, 282)
64	(211, 0.064 , 0.049, 573)	(302, 0.12 , 0.16, 1783)
100	(288, 0.099 , 0.073, 3240)	(430, 0.17 , 0.1, 6713)
144	(481, 0.17 , 0.21, 7575)	(565, 0.27 , 0.17, **F**)
196	(611, 0.23 , 0.18, **F**)	(721, 0.36 , 0.27, **F**)
256	(747, 0.32 , 0.18, **F**)	(852, 0.56 , 0.39, **F**)
324	(875, 0.48 , 0.31, **F**)	(963, 0.71 , 0.53, **F**)
400	(1045, 0.66 , 0.34, **F**)	(1201, 1 , 0.53, **F**)
484	(1240, 0.73 , 0.36, **F**)	(1375, 1.3 , 0.65, **F**)
576	(1400, 0.86 , 0.42, **F**)	(1560, 1.4 , 0.72, **F**)

an exascale architecture roadmap [56]. The salient parameters for the platform are shown in the Table 12. We do not consider parallel fast Fourier transform application since we do not have theoretical complexity of cost of communications for it. This we would address in our future work.

In the parallel matrix multiplication application, the square matrices A, B, and C of size $N \times N$ are distributed over a two-dimensional grid of processors, $\sqrt{p} \times \sqrt{p}$, where $N = n \times \Delta x$. The block size is b. This algorithm has $\frac{N}{b}$ steps. In each step, the processors broadcast a pivot row of matrix B and a pivot column of matrix A. We assume the communications are serialized. We also assume no overlap between computations and communications in the executions of the application as well as *PARALEPH*.

We compare the execution times of the application and *PARALEPH* for varying values of $\frac{n}{p}$. To simplify the exposition, we assume that configuration used in the execution of *PARALEPH* is $(q, t) = (p, 1)$. We also assume that the optimal workload distributions result in average percentage improvement of 25% in time of computations ($\tau = 0.75$). The execution time of the application considering that SUMMA uses scatter-allgather broadcast is estimated as shown below using the Hockney model [54]. Scatter is implemented using a binomial tree and allgather, a ring algorithm in which the data from each processor are sent around a virtual ring of processors in $p - 1$ steps.

$$T_{SUMMA} = T^{comp}_{SUMMA} + T^{comm}_{SUMMA} \tag{2a}$$

$$T^{comp}_{SUMMA} = \frac{2 \times N^3}{p} \times \frac{\tau}{\gamma} \tag{2b}$$

$$T^{comm}_{SUMMA} = (\log_2 p + 2 \times (\sqrt{p} - 1)) \times \alpha \times \frac{N}{b} + 4 \times (1 - \frac{1}{\sqrt{p}}) \times \beta \times \frac{N^2}{\sqrt{p}} \tag{2c}$$

TABLE 10. Execution times of *PARALEPH* and FFTW in seconds. *PARALEPH* solves *EOPT*. $\frac{n}{p} \in \{8, 16, 32\}$. Each cell in the table contains the execution time of the data-parallel application, the execution time of *PARALEPH*, the time of communications in *PARALEPH*, and the speedup of *PARALEPH* over the sequential data-partitioning algorithms. 'F' indicates failure of the sequential algorithm.

$p \downarrow, \frac{n}{p} \rightarrow$	8	16	32
2	(1.5, 0.0001 , 6e-06, 2.4)	(3.4, 0.00012 , 2e-06, 2)	(7.4, 0.00015 , 4e-06, 1.8)
4	(2.8, 0.0003 , 0.0005, 1.5)	(5, 0.00035 , 0.00052, 2)	(9.8, 0.00045 , 0.0006, 1.6)
16	(13, 0.0064 , 0.012, 1.3)	(27, 0.0071 , 0.014, 2)	(37, 0.005 , 0.003, 1.7)
36	(31, 0.0081 , 0.0063, 1.3)	(61, 0.062 , 0.058, 1.7)	(66, 0.027 , 0.055, 8.9)
64	(39, 0.04 , 0.032, 5.7)	(63, 0.03 , 0.029, 14)	(80, 0.056 , 0.075, 29.1)
100	(39, 0.035 , 0.035, 28.6)	(78, 0.064 , 0.04, 73)	(126, 0.062 , 0.047, 141)
144	(39, 0.078 , 0.13, 70)	(81, 0.087 , 0.1, 225)	(153, 0.12 , 0.15, 255)
196	(41, 0.14 , 0.16, 210)	(80, 0.1 , 0.1, 554)	(201, 0.15 , 0.16, 912)
256	(83, 0.26 , 0.13, 435)	(163, 0.28 , 0.082, 1177)	(246, 0.38 , 0.15, 1920)
324	(105, 0.33 , 0.13, 738)	(195, 0.2 , 0.21, 1421)	(301, 0.44 , 0.17, 2928)
400	(125, 0.37 , 0.12, 1865)	(228, 0.41 , 0.18, 3470)	(347, 0.42 , 0.23, 6530)
484	(139, 0.41 , 0.15, 2576)	(263, 0.49 , 0.19, 6060)	(401, 0.6 , 0.23, **F**)
576	(154, 0.46 , 0.13, 4213)	(273, 0.54 , 0.19, **F**)	(550, 0.61 , 0.26, **F**)

TABLE 11. Execution times of *PARALEPH* and FFTW in seconds. *PARALEPH* solves *EOPT*. $\frac{n}{p} \in \{64, 128\}$. Each cell in the table contains the execution time of the data-parallel application, the execution time of *PARALEPH*, the time of communications in *PARALEPH*, and the speedup of *PARALEPH* over the sequential data-partitioning algorithms. 'F' indicates failure of the sequential algorithm.

$p \downarrow, \frac{n}{p} \rightarrow$	64	128
2	(6, 0.00021 , 8e-06, 1.3)	(37, 0.00042 , 2.8e-05, 0.9)
4	(7, 0.00083 , 0.0011, 1.3)	(54, 0.0018 , 0.0022, 0.9)
16	(93, 0.016 , 0.036, 1.6)	(125, 0.026 , 0.049, 1.9)
36	(196, 0.063 , 0.093, 8.8)	(246, 0.082 , 0.16, 13.5)
64	(211, 0.078 , 0.095, 48)	(302, 0.13 , 0.15, 166)
100	(288, 0.11 , 0.063, 257)	(430, 0.2 , 0.21, 611)
144	(481, 0.18 , 0.19, 775)	(565, 0.33 , 0.33, 1574)
196	(611, 0.23 , 0.18, 1620)	(721, 0.52 , 0.29, 2913)
256	(747, 0.36 , 0.18, 2832)	(852, 0.67 , 0.43, 5307)
324	(875, 0.45 , 0.31, 5676)	(963, 0.88 , 0.47, **F**)
400	(1045, 0.72 , 0.33, **F**)	(1201, 1.1 , 0.56, **F**)
484	(1240, 0.79 , 0.46, **F**)	(1375, 1.3 , 0.7, **F**)
576	(1400, 1 , 0.45, **F**)	(1560, 1.4 , 0.71, **F**)

TABLE 12. Salient parameters of the forecasted exascale platform.

Parameters	Values
Peak performance (γ)	1E18 flops
Network Latency (α)	500 ns
Network Bandwidth ($\mathcal{B}$)	100 GB/s
Number of processors (p)	2^{20}
Block size (b)	256

where $N = n \times 1024$, α is the latency, and β is the reciprocal of network bandwidth ($\frac{1}{\mathcal{B}} = 0.01\ ns$).

The total execution time of *PARALEPH* is estimated to be sum of total cost of computations and communications as follows:

$$T_{PARALEPH} = T^{comp}_{PARALEPH} + T^{comm}_{PARALEPH} \tag{3a}$$

$$T^{comp}_{PARALEPH} = m \times (n+p) \times \frac{1}{\gamma} \tag{3b}$$

$$T^{comm}_{PARALEPH} = (\alpha + \beta \times 8) \times n \times p \tag{3c}$$

where n is a multiple of granularity, 1024×1024, and $m = 1024$.

Table 13 shows that the execution time of *PARALEPH* is negligible compared to the execution time of the application. The percentage ratio of its execution time over the application decreases as the granularity ($\frac{n}{p}$) increases.

VII. CONCLUSION

Self-adaptable data-parallel applications executing on modern extreme-scale multicore CPU platforms pose two formidable challenges to data partitioning algorithms aiming to minimize the execution time and energy of computations in these applications.

TABLE 13. Percentage ratio of execution times of *PARALEPH* and parallel matrix-matrix application based on SUMMA and employing *PARALEPH*.

$\frac{n}{p}$	$t_{PARALEPH}/t_{SUMMA}$
1	1.2E-06
2	6.1E-07
4	3.0E-07
8	1.5E-07
16	7.6E-08
32	3.8E-08
64	2.0E-08
128	9.5E-09
256	4.8E-09
512	2.4E-09
1024	1.2E-09

The first challenge arises from the new inherent complexities introduced in multicore platforms such as severe resource contention and non-uniform memory access (NUMA) due to tight integration of cores that contend for shared on-chip resources such as Last Level Cache (LLC) and interconnect (For example: Intel's Quick Path Interconnect, AMD's Hyper Transport). The second challenge is that the runtime and memory overheads of a data partitioning algorithm employed in these applications must be insignificant compared to that of the application.

The sequential data partitioning algorithms addressing the first challenge have theoretical time complexity of $O(m^2 \times p^2)$ where m is the number of points in the discrete speed/energy function and p is the number of available processors. Their practical runtime and memory costs are high therefore rendering them impracticable for employment in self-adaptable applications executing on extreme-scale multicore platforms.

We presented in this work parallel data partitioning algorithms that address both the challenges. Like the sequential algorithms, they take as input the functional models of performance and energy consumption against problem size and output workload distributions, which are globally optimal solutions. They have low time complexity of $O(m^2 \times p)$ thereby providing linear speedup of $O(p)$ and low memory complexity of $O(n)$ where n is the workload size expressed as a multiple of granularity. They employ dynamic programming approach, which also facilitates easier integration of performance and energy models of communications.

We experimentally demonstrated the low practical cost of our algorithms for two data parallel applications, matrix multiplication and fast Fourier transform, on homogeneous extreme-scale multicore clusters. We show that the parallel algorithms exhibit tremendous speedups over the sequential algorithms. Using simulations based on a forecast exascale platform, we show that the algorithms also have negligible execution times for large values of n and p.

The software for the parallel algorithms can be downloaded from the URL [1].

In our future work, we would try to develop extensions of these algorithms for clusters of p heterogeneous processors.

APPENDIX A
SUPPLEMENTAL MATERIAL

The following materials supplement the main manuscript:

- Real-life use cases highlighting the importance of self-adaptability.
- Why we use dynamic energy consumption in our problem formulations and algorithms.
- Experimental methodology followed to obtain speed and energy functions presented in the main manuscript.
- Exposition of all the helper functions called in the parallel data partitioning algorithm, *PARALEPH*.
- Classes of applications where our data partitioning algorithms can be applied.
- Experimental Results for *PARALEPH* solving *POPT* and *EOPT* for DGEMM and FFTW for varying (q, t).

APPENDIX B
SELF-ADAPTABILITY IN HPC

Self-adaptability is an important feature and the need for it arises not only due to the changing underlying execution environment but also due to the specific characteristics/requirements of the application domains (for example: adaptive mesh refinement, particle simulations, transient dynamics calculations, etc), and autotuning softwares. We furnish real-life use cases below:

1) *Self-adaptability of the solver* is vital in adaptive mesh refinement on clusters for solving large computational fluid dynamics (CFD) and computational mechanics (CM) problems where the computational load varies throughout the evolution of the solution. For example, solving for flow or stress in different parts of the domain in a multiphysics casting simulation. Williams [3] proposes parallel mesh-distribution algorithms for a solution-adaptive Laplace solver. Walshaw *et al.* [4] propose a parallel method for the dynamic partitioning of unstructured meshes. Arulananthan *et al.* [5] describe a method for determining how frequently to partition in unstructured mesh computational mechanics applications. Hendrickson and Devine [6] present qualities that a good dynamic load balancer employed in dynamic mesh partitioning must possess: a). It must distribute the mesh between the processors at runtime so that the computational load is evenly balanced and the amount of interprocessor communication is minimized. b). It must be fast so as to not dominate the computation of the main solution method. Ideally, it should have parallel capability and its memory usage must be modest. c). It must minimize the cost of data redistribution arising from dynamic partitioning. d). It must have a neat abstraction and easy-to-use user interface.
2) *Autotuning parallel softwares* perform an empirical search by generating numerous versions of a program

at runtime, which are then executed to find the best configuration of a program. A key building block that enables them to prune and accomplish this search in reasonable runtime is a fast data-partitioning algorithm that is based on realistic computation and communication performance models. Chen *et al.* [7] describe their LAPACK for clusters (LFC) software in their Self-Adapting Numerical Software (SANS) system that addresses both computational time and space complexity issues in a manner as transparent to the user as possible. Franchetti *et al.* [8] describe their program generation system, which entirely autonomously generates platform-tuned implementations of discrete Fourier transform on multicores, IBM Cell, and GPUs. HeteroScaLAPACK [9] is a linear algebra package for heterogeneous clusters that determines the optimal number and arrangement of processors to be used during the execution of a linear algebra kernel. One important reason how the mapping runtime in this software accomplishes this task in a reasonable runtime is the invocation of fast data partitioning algorithms that are based on realistic computation and communication performance models, which are efficiently constructed at runtime. From our experience and reports of expert HPC programmers, we can affirm that autotuning is a necessary feature not only in dynamic environments but even in dedicated environments where performance and energy profiles of the applications can change for different day-to-day runs with the same application configuration.

3) *Supercomputer administrators* routinely report that nodes closer to the hotter regions (hotspots) execute codes slower than the nodes closer to the cooler regions in the supercomputing centers due to variations in the airflow caused by the layout of the cooling systems [10]. Thermal-aware workload scheduling techniques [11]–[13] take into account these temperature variations to optimize for performance and energy. Therefore, static data partitioning strategies are not ideally suitable to address this situation. Production codes executed in the supercomputers still continue to use static techniques mainly to avoid the enormous cost and risk of bugs incurred by changing their codes to employ dynamic load balancing.
4) *Shared environments* such as cloud computing systems today are placing great emphasis in facilitating easier migration and execution of HPC workloads by striving to remove impediments to this process. The leading objectives for optimization for the cloud service providers are performance, energy consumption, cost, and reliability. Self-adaptable applications employing fast data partitioning algorithms for optimization of their performance and energy evidently and directly address the first two concerns. Atif and Strazdins [14] present a framework that performs live migration of jobs to improve the overall throughput and performance of a cloud infrastructure. The core of this framework is a lightweight runtime profiler, which constructs computation and communication models at runtime at low cost, to guide the migration decisions.

Algorithm 5 Function Determining the Sample Mean of an Experimental Run Using Student's t-Test

1: **procedure** MeanUsingTtest(*app*, *minReps*, *maxReps*, *maxT*, *cl*, *eps*, *repsOut*, *clOut*, *etimeOut*, *epsOut*, *mean*)

Input:
The application to execute, *app*
The objective function to measure, *Objective*, *TIME* or *ENERGY*
The minimum number of repetitions, $minReps \in \mathbb{Z}_{>0}$
The maximum number of repetitions, $maxReps \in \mathbb{Z}_{>0}$
The maximum time allowed for the application to run, $maxT \in \mathbb{R}_{>0}$
The required confidence level, $cl \in \mathbb{R}_{>0}$
The required accuracy, $eps \in \mathbb{R}_{>0}$

Output:
The number of experimental runs actually made, $repsOut \in \mathbb{Z}_{>0}$
The confidence level achieved, $clOut \in \mathbb{R}_{>0}$
The accuracy achieved, $epsOut \in \mathbb{R}_{>0}$
The elapsed time, $etimeOut \in \mathbb{R}_{>0}$
The mean, $mean \in \mathbb{R}_{>0}$

2: $reps \leftarrow 0$; $stop \leftarrow 0$; $sum \leftarrow 0$; $etime \leftarrow 0$
3: **while** ($reps < maxReps$) and (!$stop$) **do**
4: $st \leftarrow$ measure(*TIME*)
5: $start \leftarrow$ measure(*Objective*)
6: Execute(*app*)
7: $end \leftarrow$ measure(*Objective*)
8: $et \leftarrow$ measure(*TIME*)
9: $reps \leftarrow reps + 1$
10: $etime \leftarrow etime + et - st$
11: $ObjArray[reps] \leftarrow$ query(*Objective*)
12: $sum \leftarrow sum + ObjArray[reps]$
13: **if** $reps > minReps$ **then**
14: $clOut \leftarrow$ fabs(gsl_cdf_tdist_Pinv(cl, $reps - 1$)) $\times$ gsl_stats_sd(*ObjArray*, 1, *reps*) / sqrt(*reps*)
15: **if** $clOut \times \frac{reps}{sum} < eps$ **then**
16: $stop \leftarrow 1$
17: **end if**
18: **if** $etime > maxT$ **then**
19: $stop \leftarrow 1$
20: **end if**
21: **end if**
22: **end while**
23: $repsOut \leftarrow reps$; $epsOut \leftarrow clOut \times \frac{reps}{sum}$
24: $etimeOut \leftarrow etime$; $mean \leftarrow \frac{sum}{reps}$
25: **end procedure**

Algorithm 6 The Cell (v, h) in the Anti-Diagonal L is Computed

function computecell($\mathcal{F}$, Ψ, me, $m, p, v, h, dptLeft, dpt, tb$)
  $myh \leftarrow h - me \times \frac{p}{q}$
  if $myh = 1$ **and** $me \neq 0$ **then**
    $dpt(v, myh) \leftarrow dptLeft(v)$
  else
    $dpt(v, myh) \leftarrow dpt(v, myh - 1)$
  end if
  $tb(v) \leftarrow (v, h - 1)$
  for $I \leftarrow 1, \min(m, v - 1)$ **do**
    if $myh = 1$ **and** $me \neq 0$ **then**
      $tmp \leftarrow \mathcal{F}(dpt(I, 1), dptLeft(v - I))$
    else
      $tmp \leftarrow \mathcal{F}(dpt(I, 1), dpt(v - I, myh - 1))$
    end if
    if $dpt(v, myh) > tmp$ **then**
      $dpt(v, myh) \leftarrow tmp$
      $tb(v) \leftarrow ((I, 1), (v - I, h - 1))$
    end if
  end for
  return (dpt,tb)
end function

Algorithm 7 Get Number of Cells in a Minor Anti-Diagonal L

function getncells($L, n, p, ncells$)
  $n_1 \leftarrow n - 1; n_2 \leftarrow p - 1$
  $max12 \leftarrow \max(n_1, n_2); min12 \leftarrow \min(n_1, n_2)$
  if $(L - 3) \leq max12$ **then**
    $ncells \leftarrow ncells + 1$
    if $ncells > min12$ **then**
      $ncells \leftarrow min12$
    else
      $ncells \leftarrow ncells - 1$
    end if
  end if
  return ncells
end function

Algorithm 8 Get Number of Cells in a Minor Anti-Diagonal L Belonging to Process me

function getmyncells($me, L, start, ncells, p, q$)
  $myncells \leftarrow 0; ncellsbeforeme \leftarrow 0$
  for $c \leftarrow 1, ncells$ **do**
    $col \leftarrow L - start + v$
    if $col < me \times \frac{p}{q}$ **then**
      $ncellsbeforeme \leftarrow ncellsbeforeme + 1$
    end if
    if $col \geq me \times \frac{p}{q}$ **and** $col < (me + 1) \times \frac{p}{q}$ **then**
      $myncells \leftarrow myncells + 1$
    end if
    if $col \geq (me + 1) \times \frac{p}{q}$ **then break end if**
  end for
  if $myncells = 0$ **then** $ncellsbeforeme \leftarrow 0$ **end if**
  return (myncells,ncellsbeforeme)
end function

Algorithm 9 Get Number of Cells in a Minor Anti-Diagonal L Belonging to Process me

function fillD($me, v, h, p, q, root, tb, D_{opt}$)
  if $(v = 0) and (h = 0)$ **then**
    ;
  else if $(v = 1) and (h = 1)$ **then**
    if $me = 0$ **then**
      $D_{opt}(rec) \leftarrow 1; rec \leftarrow rec + 1$
    end if
  else if $v = 1$ **then**
    if $me = 0$ **then**
      $D_{opt}(rec) \leftarrow 1; rec \leftarrow rec + 1$
      $D_{opt}(proc) \leftarrow 0, \forall proc \in [2, h]$
    end if
  else
    for $r \leftarrow 0, q$ **do**
      if $(h \geq r \times \frac{p}{q}) and (h < (r + 1) \times \frac{p}{q})$ **then**
        $root \leftarrow r$; **break**
      end if
    end for
    Traceback($v, h, q, root, tb, D_{opt}$)
  end if
  return (D_{opt})
end function

APPENDIX C
RATIONALE BEHIND USING DYNAMIC ENERGY CONSUMPTION INSTEAD OF TOTAL ENERGY CONSUMPTION

In this section, we describe the terms related to energy predictive models used in this work. We also explain the rationale behind using only the dynamic energy consumption in our problem formulations and algorithms.

There are two types of power consumptions in a component: dynamic power and static power. Dynamic power consumption is caused by the switching activity in the component's circuits. Static power is the power consumed when the component is not active or doing work. Static power is also known as idle power or base power. From an application point of view, we define dynamic and static power consumption as the power consumption of the whole system with and without the given application execution. From the component point of view, we define dynamic and static power consumption of the component as the power consumption of the component with and without the given application utilizing the component during its execution.

We obtain the power consumption during the application execution using *Watts Up Pro* power meter.

There are two types of energy consumptions, static energy and dynamic energy. We define the static energy consumption as the energy consumption of the platform without the

Algorithm 10 Traceback of Pointers to Reconstruct the Optimal Distribution of Workload of Size n

1: **procedure** traceback($v, h, p, q, root, tb, D_{opt}$)
Input:
The row index of the dpt cell, $v \in \mathbb{Z}_{>0}$
The column index of the dpt cell, $h \in \mathbb{Z}_{>0}$
Number of parallel processes executing $PARALEPH$, $q \in \mathbb{Z}_{>0}$
root, The root process owning the cell (v, h), $root \in [1, q]$
Traceback pointers array, tb
Output:
Optimal workload distribution,
$D_{opt} = \{x^1_{opt}, ..., x^p_{opt}\}, x^i_{opt} \in \mathbb{Z}_{>0}, \forall i \in [1, p]$
2: $me \leftarrow MPI\_Get\_rank(MPI\_COMM\_WORLD)$
3: **if** $me = root$ **then**
4: $myh \leftarrow h - me \times \frac{p}{q}$
5: $tbvh \leftarrow tb(v-1, myh)$
6: **end if**
7: $MPI\_Bcast(tbvh, 4, , root, ...)$
8: $(n_1, p_1, n_2, p_2) \leftarrow (tbvh[0], tbvh[1], tbvh[2], tbvh[3])$
9: $fillD(me, n_1, p_1, p, q, root, tb, D_{opt})$
10: $fillD(me, n_2, p_2, p, q, root, tb, D_{opt})$
11: **return** (D_{opt})
12: **end procedure**

given application execution. Dynamic energy consumption is the difference between the total energy consumption of the platform during the given application execution and the static energy consumption. That is, if P_S is the static power consumption of the platform, E_T is the total energy consumption of the platform during the execution of an application, which takes T_E seconds, then the dynamic energy E_D can be calculated as,

$$E_D = E_T - (P_S \times T_E) \tag{4}$$

We consider only the dynamic energy consumption in our problem formulations and algorithms for reasons below:

1) Static energy consumption is a hard constant (or a inherent property) of a platform that can not be optimized. That is, it does not depend on the application configuration and will be the same for different application configurations.
2) Although static energy consumption is a major concern in embedded systems, it is becoming less compared to the dynamic energy consumption due to advancements in hardware architecture design in HPC systems.
3) We target applications and platforms where dynamic energy consumption is the dominating energy dissipator.
4) Finally, we believe its inclusion can underestimate the true worth of an optimization technique that minimizes the dynamic energy consumption. We elucidate using two examples from published results.
   - In our first example, consider a model that reports predicted and measured total energy consumption of a system to be 16500J and 18000J. It would report the prediction error to be 8.3%. If it is however known that the static energy consumption of the system is 9000J, then the real prediction error (based on dynamic energy consumptions only) would be 16.6% instead.
   - In our second example, consider two different energy prediction models (M_A and M_B) with same prediction errors of 5% for an application execution on two different machines (A and B) with same total energy consumption of 10000J. One would consider both the models to be equally accurate. But supposing it is known that the dynamic energy proportions for the machines are 30% and 60%. Now, the true prediction errors (using dynamic energy consumptions only) for the models would be 16.6% and 8.3%. Therefore, the second model M_B should be considered more accurate than the first.

APPENDIX D
EXPERIMENTAL METHODOLOGY TO BUILD THE SPEED/PERFORMANCE AND ENERGY FUNCTIONS

To make sure our experimental results are reliable, we follow the methodology described below:

- The server is fully reserved and dedicated to our experiments during their execution. We also made certain that there are no drastic fluctuations in the load due to abnormal events in the server by monitoring its load continuously for a week using the tool *sar*. We observed insignificant variation in the load during this monitoring period suggesting normal and clean behavior of the server.
- An application is bound to the physical cores using the *numactl* tool during its execution.
- To obtain a data point in the speed and energy function, the application is repeatedly executed until the sample mean lies in the 95% confidence interval with a precision of 0.025 (2.5%). For this purpose, we use Student's t-test assuming that the individual observations are independent and their population follows the normal distribution. We verify the validity of these assumptions using Pearson's chi-squared test.
  The function $MeanUsingTtest$, shown in Algorithm 5, describes this step. For each data point, the function is invoked, which repeatedly executes the application app until one of the following three conditions is satisfied:
  1) The maximum number of repetitions ($maxReps$) is exceeded (Line 3).
  2) The sample mean falls in the confidence interval (satisfying the precision of measurement eps) (Lines 15-17).

TABLE 14. Execution times of *PARALEPH* for DGEMM solving *POPT* for varying values of t in seconds. Each tuple in a cell contains the number of threads used in the execution of *PARALEPH*, the execution time of *PARALEPH*, and the time of communications in *PARALEPH*.

$p \downarrow, \frac{n}{p} \rightarrow$	8	16	32	64	128
2	(2, 0.000, 0.000) (4, 0.000, 0.000) (8, 0.000, 0.000)	(2, 0.000, 0.000) (4, 0.000, 0.000) (8, 0.000, 0.000)	(2, 0.000, 0.000) (4, 0.000, 0.000) (8, 0.000, 0.000)	(2, 0.000, 0.000) (4, 0.000, 0.000) (8, 0.000, 0.000)	(2, 0.001, 0.000) (4, 0.001, 0.000) (8, 0.001, 0.000)
4	(2, 0.002, 0.000) (4, 0.000, 0.000) (8, 0.000, 0.000)	(2, 0.002, 0.000) (4, 0.000, 0.000) (8, 0.000, 0.000)	(2, 0.002, 0.000) (4, 0.001, 0.000) (8, 0.001, 0.000)	(2, 0.003, 0.001) (4, 0.001, 0.000) (8, 0.017, 0.000)	(2, 0.004, 0.002) (4, 0.003, 0.000) (8, 0.002, 0.000)
16	(2, 0.009, 0.003) (4, 0.005, 0.002) (8, 0.026, 0.024)	(2, 0.010, 0.004) (4, 0.007, 0.004) (8, 0.091, 0.034)	(2, 0.014, 0.006) (4, 0.029, 0.023) (8, 0.109, 0.039)	(2, 0.022, 0.011) (4, 0.020, 0.011) (8, 0.044, 0.028)	(2, 0.038, 0.019) (4, 0.070, 0.050) (8, 0.065, 0.030)
36	(2, 0.021, 0.008) (4, 0.040, 0.032) (8, 0.035, 0.022)	(2, 0.025, 0.011) (4, 0.018, 0.008) (8, 0.056, 0.036)	(2, 0.033, 0.015) (4, 0.035, 0.022) (8, 0.065, 0.047)	(2, 0.053, 0.025) (4, 0.061, 0.035) (8, 0.247, 0.200)	(2, 0.095, 0.046) (4, 0.094, 0.047) (8, 0.324, 0.224)
64	(2, 0.040, 0.014) (4, 0.035, 0.019) (8, 0.098, 0.085)	(2, 0.046, 0.018) (4, 0.068, 0.048) (8, 0.117, 0.092)	(2, 0.065, 0.029) (4, 0.079, 0.049) (8, 0.164, 0.132)	(2, 0.099, 0.049) (4, 0.106, 0.052) (8, 0.362, 0.301)	(2, 0.174, 0.093) (4, 0.178, 0.085) (8, 0.346, 0.225)
100	(2, 0.057, 0.021) (4, 0.058, 0.034) (8, 0.820, 0.800)	(2, 0.067, 0.026) (4, 0.061, 0.030) (8, 0.471, 0.436)	(2, 0.097, 0.044) (4, 0.092, 0.044) (8, 0.511, 0.449)	(2, 0.151, 0.074) (4, 0.182, 0.112) (8, 0.250, 0.137)	(2, 0.262, 0.142) (4, 0.279, 0.138) (8, 0.497, 0.309)
144	(2, 0.081, 0.030) (4, 0.069, 0.033) (8, 0.564, 0.531)	(2, 0.096, 0.036) (4, 0.086, 0.040) (8, 0.164, 0.114)	(2, 0.137, 0.059) (4, 0.168, 0.101) (8, 0.679, 0.583)	(2, 0.219, 0.107) (4, 0.249, 0.142) (8, 0.460, 0.289)	(2, 0.372, 0.193) (4, 0.401, 0.216) (8, 0.729, 0.401)
196	(2, 0.094, 0.035) (4, 0.137, 0.054) (8, 0.894, 0.847)	(2, 0.133, 0.062) (4, 0.113, 0.053) (8, 0.539, 0.466)	(2, 0.208, 0.113) (4, 0.182, 0.095) (8, 1.151, 1.032)	(2, 0.364, 0.220) (4, 0.333, 0.153) (8, 0.490, 0.273)	(2, 0.636, 0.400) (4, 0.545, 0.292) (8, 0.925, 0.588)
256	(2, 0.114, 0.040) (4, 0.109, 0.040) (8, 0.196, 0.136)	(2, 0.168, 0.067) (4, 0.158, 0.072) (8, 0.331, 0.242)	(2, 0.308, 0.117) (4, 0.274, 0.153) (8, 0.473, 0.330)	(2, 0.473, 0.213) (4, 0.394, 0.201) (8, 0.752, 0.511)	(2, 0.865, 0.399) (4, 0.720, 0.379) (8, 1.034, 0.556)
324	(2, 0.183, 0.058) (4, 0.189, 0.070) (8, 0.283, 0.204)	(2, 0.231, 0.107) (4, 0.208, 0.108) (8, 0.440, 0.336)	(2, 0.395, 0.199) (4, 0.401, 0.218) (8, 0.455, 0.277)	(2, 0.706, 0.368) (4, 0.595, 0.357) (8, 0.813, 0.523)	(2, 1.350, 0.702) (4, 1.076, 0.665) (8, 1.419, 0.852)
400	(2, 0.180, 0.067) (4, 0.235, 0.099) (8, 0.505, 0.422)	(2, 0.288, 0.115) (4, 0.270, 0.121) (8, 0.486, 0.357)	(2, 0.496, 0.205) (4, 0.451, 0.212) (8, 0.639, 0.411)	(2, 0.897, 0.368) (4, 0.793, 0.375) (8, 2.111, 1.740)	(2, 1.717, 0.717) (4, 1.446, 0.678) (8, 1.611, 0.952)
484	(2, 0.233, 0.097) (4, 0.208, 0.086) (8, 0.492, 0.372)	(2, 0.381, 0.167) (4, 0.337, 0.155) (8, 1.107, 0.939)	(2, 0.671, 0.300) (4, 0.592, 0.296) (8, 1.215, 0.953)	(2, 1.295, 0.611) (4, 0.972, 0.432) (8, 1.245, 0.771)	(2, 2.483, 1.184) (4, 1.877, 0.887) (8, 1.910, 1.071)
576	(2, 0.279, 0.097) (4, 0.239, 0.095) (8, 3.014, 2.822)	(2, 0.465, 0.170) (4, 0.446, 0.237) (8, 53.021, 47.760)	(2, 0.873, 0.329) (4, 0.678, 0.312) (8, 65.350, 60.106)	(2, 1.617, 0.588) (4, 1.219, 0.571) (8, 243.530, 234.624)	(2, 3.156, 1.178) (4, 2.287, 1.067) (8, 114.705, 113.348)

3) The elapsed time of the repetitions of application execution exceeds the maximum time allowed (*maxT* in seconds) (Lines 18-20).

So, for each data point, the function *MeanUsingTtest* is invoked and the sample mean *mean* is returned at the end of invocation. The function *Measure* measures the execution time or the dynamic energy consumption using the library [57] based on the input, *TIME* or *ENERGY*. The input minimum and maximum number of repetitions, *minReps* and *maxReps*, differ based on the problem size solved. For small problem sizes ($32 \leq n \leq 1024$), these values are 10000 and 100000. For medium problem sizes ($1024 < n \leq 5120$), these values are 100 and 1000. For large problem sizes ($n > 5120$), these values are 5 and 500. The values of *maxT*, *cl*, and *eps* are 3600 seconds, 0.95, and 0.025. If the precision of measurement is not achieved before the maximum number of repeats is exceeded, we increase the number of repetitions and also the maximum elapsed time allowed.

APPENDIX E
PARALEPH: PARALLEL DATA PARTITIONING ALGORITHM USING DYNAMIC PROGRAMMING

Subroutines called in *PARALEPH* are presented here.

TABLE 15. Execution times of *PARALEPH* for DGEMM solving *EOPT* for varying values of t in seconds.

$p \downarrow, \frac{n}{p} \rightarrow$	8	16	32	64	128
2	(2, 0.000, 0.000) (4, 0.000, 0.000) (8, 0.000, 0.000)	(2, 0.000, 0.000) (4, 0.000, 0.000) (8, 0.000, 0.000)	(2, 0.000, 0.000) (4, 0.000, 0.000) (8, 0.000, 0.000)	(2, 0.000, 0.000) (4, 0.000, 0.000) (8, 0.000, 0.000)	(2, 0.001, 0.000) (4, 0.001, 0.000) (8, 0.001, 0.000)
4	(2, 0.001, 0.000) (4, 0.000, 0.000) (8, 0.000, 0.000)	(2, 0.001, 0.000) (4, 0.000, 0.000) (8, 0.000, 0.000)	(2, 0.002, 0.000) (4, 0.016, 0.000) (8, 0.001, 0.000)	(2, 0.002, 0.001) (4, 0.020, 0.000) (8, 0.001, 0.000)	(2, 0.004, 0.002) (4, 0.002, 0.000) (8, 0.002, 0.000)
16	(2, 0.009, 0.003) (4, 0.005, 0.002) (8, 0.051, 0.048)	(2, 0.011, 0.005) (4, 0.026, 0.022) (8, 0.061, 0.056)	(2, 0.014, 0.006) (4, 0.010, 0.005) (8, 0.036, 0.026)	(2, 0.022, 0.012) (4, 0.018, 0.009) (8, 0.052, 0.018)	(2, 0.039, 0.020) (4, 0.044, 0.024) (8, 0.075, 0.043)
36	(2, 0.021, 0.008) (4, 0.024, 0.016) (8, 0.120, 0.115)	(2, 0.024, 0.010) (4, 0.018, 0.009) (8, 0.054, 0.038)	(2, 0.033, 0.015) (4, 0.033, 0.020) (8, 0.061, 0.038)	(2, 0.052, 0.024) (4, 0.088, 0.063) (8, 0.100, 0.053)	(2, 0.093, 0.045) (4, 0.096, 0.045) (8, 0.175, 0.071)
64	(2, 0.039, 0.013) (4, 0.037, 0.022) (8, 0.060, 0.046)	(2, 0.046, 0.019) (4, 0.040, 0.020) (8, 0.071, 0.047)	(2, 0.064, 0.028) (4, 0.056, 0.026) (8, 0.188, 0.153)	(2, 0.101, 0.049) (4, 0.117, 0.066) (8, 0.181, 0.119)	(2, 0.170, 0.084) (4, 0.174, 0.082) (8, 0.245, 0.108)
100	(2, 0.056, 0.020) (4, 0.045, 0.021) (8, 0.396, 0.308)	(2, 0.068, 0.027) (4, 0.055, 0.024) (8, 0.354, 0.321)	(2, 0.118, 0.064) (4, 0.108, 0.064) (8, 0.429, 0.365)	(2, 0.152, 0.071) (4, 0.174, 0.097) (8, 0.435, 0.339)	(2, 0.259, 0.127) (4, 0.317, 0.177) (8, 0.445, 0.240)
144	(2, 0.083, 0.031) (4, 0.079, 0.043) (8, 0.161, 0.123)	(2, 0.099, 0.038) (4, 0.083, 0.037) (8, 0.255, 0.203)	(2, 0.136, 0.056) (4, 0.132, 0.064) (8, 0.290, 0.204)	(2, 0.217, 0.099) (4, 0.246, 0.135) (8, 1.311, 1.191)	(2, 0.377, 0.191) (4, 0.423, 0.226) (8, 0.863, 0.541)
196	(2, 0.093, 0.035) (4, 0.097, 0.050) (8, 0.152, 0.110)	(2, 0.138, 0.066) (4, 0.110, 0.050) (8, 0.267, 0.186)	(2, 0.208, 0.110) (4, 0.183, 0.093) (8, 0.742, 0.624)	(2, 0.363, 0.218) (4, 0.336, 0.147) (8, 0.850, 0.582)	(2, 0.665, 0.414) (4, 0.540, 0.275) (8, 1.089, 0.764)
256	(2, 0.113, 0.037) (4, 0.140, 0.071) (8, 0.762, 0.695)	(2, 0.165, 0.064) (4, 0.172, 0.081) (8, 0.721, 0.637)	(2, 0.275, 0.118) (4, 0.260, 0.136) (8, 0.376, 0.220)	(2, 0.468, 0.204) (4, 0.435, 0.234) (8, 1.048, 0.796)	(2, 0.857, 0.340) (4, 0.745, 0.392) (8, 1.019, 0.541)
324	(2, 0.148, 0.059) (4, 0.183, 0.103) (8, 0.401, 0.322)	(2, 0.234, 0.103) (4, 0.258, 0.121) (8, 0.292, 0.181)	(2, 0.397, 0.198) (4, 0.400, 0.249) (8, 0.578, 0.395)	(2, 0.712, 0.365) (4, 0.593, 0.345) (8, 0.983, 0.674)	(2, 1.357, 0.684) (4, 1.093, 0.654) (8, 1.364, 0.733)
400	(2, 0.182, 0.064) (4, 0.188, 0.082) (8, 0.289, 0.196)	(2, 0.297, 0.112) (4, 0.320, 0.137) (8, 0.493, 0.359)	(2, 1.434, 1.120) (4, 0.460, 0.202) (8, 0.735, 0.499)	(2, 0.977, 0.353) (4, 0.816, 0.363) (8, 1.190, 0.775)	(2, 1.846, 0.780) (4, 1.525, 0.688) (8, 2.807, 2.064)
484	(2, 0.232, 0.091) (4, 0.208, 0.084) (8, 0.385, 0.261)	(2, 0.386, 0.159) (4, 0.335, 0.145) (8, 0.649, 0.470)	(2, 0.682, 0.297) (4, 0.559, 0.246) (8, 0.811, 0.523)	(2, 1.260, 0.525) (4, 1.008, 0.447) (8, 1.677, 1.218)	(2, 2.614, 1.208) (4, 1.902, 0.832) (8, 1.972, 1.118)
576	(2, 0.289, 0.102) (4, 0.243, 0.098) (8, 14.149, 11.554)	(2, 0.467, 0.144) (4, 0.414, 0.192) (8, 12.460, 7.163)	(2, 0.831, 0.246) (4, 0.707, 0.323) (8, 16.501, 13.077)	(2, 1.713, 0.600) (4, 1.252, 0.577) (8, 20.946, 16.597)	(2, 2.952, 0.816) (4, 2.397, 1.108) (8, 317.244, 311.997)

A. COMPUTATION OF THE CELL (V, H) IN THE ANTI-DIAGONAL L

The algorithm 6 computes the cell (v, h) in the anti-diagonal L. This is a local routine.

B. NUMBER OF CELLS IN AN ANTI-DIAGONAL L

The algorithm 7 returns the number of cells in an anti-diagonal L. This is a local routine.

C. NUMBER OF CELLS IN AN ANTI-DIAGONAL L OWNED BY PROCESS ME

The algorithm 8 returns the number of cells, *myncells*, in an anti-diagonal L belonging to process *me* and the number of cells, *ncellsbeforeme*, that precede the cells belonging to process *me*. The input *ncells* is the total number of cells in an anti-diagonal that are distributed between the q processes. This is also a local routine.

D. TRACEBACK

This section explains the recursive traceback algorithm 10. All the parallel processes participate in the execution of this routine. The traceback array entry for *dpt* cell, (v, h), contains four values, (n_1, p_1, n_2, p_2) where $v = n_1 + n_2, h = p_1 + p_2$.

The process *root* owns the traceback entries for the table cell (v, h). It broadcasts this cell to the other processes. The processes owning (n_1, p_1) and (n_2, p_2) become roots for the subsequent recursive broadcasts. The recursion terminates when three conditions are met: a). The traceback values,

TABLE 16. Execution times of *PARALEPH* for FFTW solving *POPT* for varying values of t in seconds.

$p\downarrow, \frac{n}{p}\rightarrow$	8	16	32	64	128
2	(2, 0.000, 0.000) (4, 0.000, 0.000) (8, 0.000, 0.000)	(2, 0.000, 0.000) (4, 0.000, 0.000) (8, 0.000, 0.000)	(2, 0.000, 0.000) (4, 0.000, 0.000) (8, 0.000, 0.000)	(2, 0.000, 0.000) (4, 0.000, 0.000) (8, 0.000, 0.000)	(2, 0.001, 0.000) (4, 0.001, 0.000) (8, 0.001, 0.000)
4	(2, 0.001, 0.000) (4, 0.000, 0.000) (8, 0.000, 0.000)	(2, 0.002, 0.000) (4, 0.000, 0.000) (8, 0.000, 0.000)	(2, 0.002, 0.000) (4, 0.001, 0.000) (8, 0.001, 0.000)	(2, 0.003, 0.001) (4, 0.001, 0.000) (8, 0.001, 0.000)	(2, 0.005, 0.002) (4, 0.003, 0.000) (8, 0.003, 0.000)
16	(2, 0.009, 0.003) (4, 0.005, 0.002) (8, 0.030, 0.026)	(2, 0.010, 0.004) (4, 0.007, 0.003) (8, 0.024, 0.006)	(2, 0.013, 0.006) (4, 0.028, 0.022) (8, 0.015, 0.010)	(2, 0.021, 0.010) (4, 0.018, 0.010) (8, 0.120, 0.064)	(2, 0.035, 0.020) (4, 0.033, 0.017) (8, 0.057, 0.018)
36	(2, 0.020, 0.007) (4, 0.012, 0.004) (8, 0.103, 0.090)	(2, 0.023, 0.009) (4, 0.017, 0.007) (8, 0.071, 0.058)	(2, 0.032, 0.014) (4, 0.025, 0.012) (8, 0.068, 0.032)	(2, 0.048, 0.024) (4, 0.055, 0.034) (8, 0.076, 0.038)	(2, 0.080, 0.045) (4, 0.107, 0.068) (8, 0.138, 0.067)
64	(2, 0.037, 0.012) (4, 0.044, 0.027) (8, 0.205, 0.191)	(2, 0.044, 0.016) (4, 0.043, 0.022) (8, 0.131, 0.096)	(2, 0.059, 0.026) (4, 0.064, 0.038) (8, 0.132, 0.086)	(2, 0.089, 0.048) (4, 0.149, 0.106) (8, 0.219, 0.152)	(2, 0.148, 0.083) (4, 0.205, 0.132) (8, 0.231, 0.155)
100	(2, 0.052, 0.016) (4, 0.061, 0.036) (8, 0.275, 0.256)	(2, 0.064, 0.024) (4, 0.073, 0.042) (8, 0.176, 0.148)	(2, 0.086, 0.039) (4, 0.102, 0.061) (8, 0.177, 0.138)	(2, 0.132, 0.068) (4, 0.155, 0.098) (8, 0.386, 0.302)	(2, 0.224, 0.127) (4, 0.238, 0.130) (8, 0.419, 0.286)
144	(2, 0.074, 0.023) (4, 0.074, 0.038) (8, 0.180, 0.146)	(2, 0.087, 0.031) (4, 0.092, 0.050) (8, 0.273, 0.228)	(2, 0.122, 0.053) (4, 0.131, 0.074) (8, 0.282, 0.197)	(2, 0.188, 0.097) (4, 0.208, 0.123) (8, 0.505, 0.365)	(2, 0.317, 0.180) (4, 0.347, 0.205) (8, 0.799, 0.603)
196	(2, 0.088, 0.030) (4, 0.072, 0.028) (8, 0.327, 0.276)	(2, 0.117, 0.051) (4, 0.109, 0.054) (8, 0.240, 0.171)	(2, 0.174, 0.092) (4, 0.186, 0.110) (8, 0.321, 0.212)	(2, 0.287, 0.171) (4, 0.299, 0.148) (8, 0.444, 0.289)	(2, 0.500, 0.323) (4, 0.462, 0.268) (8, 1.268, 0.986)
256	(2, 0.103, 0.034) (4, 0.115, 0.049) (8, 0.267, 0.212)	(2, 0.143, 0.057) (4, 0.141, 0.064) (8, 0.294, 0.212)	(2, 0.219, 0.100) (4, 0.228, 0.123) (8, 0.495, 0.369)	(2, 0.369, 0.184) (4, 0.337, 0.181) (8, 0.931, 0.736)	(2, 0.655, 0.339) (4, 0.597, 0.338) (8, 0.979, 0.612)
324	(2, 0.167, 0.049) (4, 0.167, 0.053) (8, 0.222, 0.152)	(2, 0.189, 0.085) (4, 0.176, 0.084) (8, 1.306, 1.203)	(2, 0.305, 0.155) (4, 0.274, 0.149) (8, 0.462, 0.311)	(2, 0.527, 0.288) (4, 0.463, 0.275) (8, 1.150, 0.865)	(2, 0.968, 0.549) (4, 0.836, 0.521) (8, 1.394, 0.900)
400	(2, 0.154, 0.055) (4, 0.162, 0.069) (8, 0.539, 0.446)	(2, 0.226, 0.092) (4, 0.216, 0.094) (8, 1.199, 1.086)	(2, 0.371, 0.164) (4, 0.351, 0.170) (8, 0.951, 0.762)	(2, 0.691, 0.307) (4, 0.622, 0.328) (8, 0.964, 0.637)	(2, 1.218, 0.582) (4, 1.129, 0.608) (8, 1.418, 0.870)
484	(2, 0.187, 0.072) (4, 0.184, 0.079) (8, 0.514, 0.399)	(2, 0.297, 0.135) (4, 0.262, 0.114) (8, 0.679, 0.509)	(2, 0.488, 0.230) (4, 0.431, 0.204) (8, 0.648, 0.407)	(2, 0.899, 0.453) (4, 0.748, 0.361) (8, 1.048, 0.637)	(2, 1.670, 0.845) (4, 1.348, 0.684) (8, 1.679, 0.938)
576	(2, 0.233, 0.089) (4, 0.213, 0.091) (8, 12.255, 10.689)	(2, 0.351, 0.135) (4, 0.368, 0.167) (8, 63.178, 56.298)	(2, 0.592, 0.232) (4, 0.511, 0.221) (8, 11.343, 8.347)	(2, 1.071, 0.425) (4, 0.889, 0.421) (8, 10.173, 6.668)	(2, 2.032, 0.822) (4, 1.608, 0.752) (8, 368.169, 363.260)

n_1 and n_2, are zero. b). The traceback values, p_1 and p_2, are both 1, and c). The traceback entries n_1 and n_2, are both 1.

The process 0 during the execution of this routine fills the optimal workload distribution, D_{opt}, in the routine, *fillD*.

APPENDIX F
CLASSES OF APPLICATIONS WHERE *PARALEPH* CAN BE EMPLOYED

From our experience, there are multiple classes of applications that benefit from our data partitioning algorithms. A dominant class contains applications where the speed of an application is a function of problem size, which is defined as a set of one, two or more parameters characterizing the amount and layout of data stored and processed during the execution of a computational task. Some classes are the following:

- Data-parallel applications, which involve dense matrix computations. For example: Matrix-vector multiplication, Matrix-matrix multiplication, QR decomposition, LU decomposition, Cholesky decomposition, etc.
- Image and signal processing applications, which involved a fast Fourier transform (FFT).
- Applications performing inexact matching of strings. For example: the core algorithm in gene sequencing applications, Smith-Waterman, which computes the alignment of two sequences. The speed here is a function of problem size, which is represented by the lengths of the two sequences.
- Dense or regular stencil kernels where the speed is a function of the size of the dimensions representing the computational domain of the stencil. For example:

TABLE 17. Execution times of *PARALEPH* for FFTW solving *EOPT* for varying values of t in seconds.

$p\downarrow, \frac{n}{p}\rightarrow$	8	16	32	64	128
2	(2, 0.000, 0.000) (4, 0.000, 0.000) (8, 0.000, 0.000)	(2, 0.000, 0.000) (4, 0.000, 0.000) (8, 0.000, 0.000)	(2, 0.000, 0.000) (4, 0.000, 0.000) (8, 0.000, 0.000)	(2, 0.000, 0.000) (4, 0.000, 0.000) (8, 0.000, 0.000)	(2, 0.001, 0.000) (4, 0.001, 0.000) (8, 0.001, 0.000)
4	(2, 0.001, 0.000) (4, 0.000, 0.000) (8, 0.000, 0.000)	(2, 0.001, 0.000) (4, 0.000, 0.000) (8, 0.000, 0.000)	(2, 0.002, 0.000) (4, 0.001, 0.000) (8, 0.001, 0.000)	(2, 0.003, 0.001) (4, 0.001, 0.000) (8, 0.001, 0.000)	(2, 0.005, 0.001) (4, 0.014, 0.000) (8, 0.003, 0.000)
16	(2, 0.009, 0.003) (4, 0.034, 0.031) (8, 0.041, 0.012)	(2, 0.010, 0.004) (4, 0.016, 0.012) (8, 0.016, 0.009)	(2, 0.014, 0.006) (4, 0.019, 0.014) (8, 0.150, 0.055)	(2, 0.021, 0.010) (4, 0.018, 0.010) (8, 0.030, 0.012)	(2, 0.036, 0.019) (4, 0.034, 0.017) (8, 0.049, 0.016)
36	(2, 0.020, 0.007) (4, 0.012, 0.005) (8, 0.033, 0.020)	(2, 0.023, 0.009) (4, 0.027, 0.017) (8, 0.132, 0.119)	(2, 0.031, 0.013) (4, 0.053, 0.039) (8, 0.115, 0.094)	(2, 0.048, 0.024) (4, 0.057, 0.036) (8, 0.091, 0.039)	(2, 0.083, 0.044) (4, 0.093, 0.052) (8, 0.197, 0.126)
64	(2, 0.038, 0.013) (4, 0.039, 0.022) (8, 0.101, 0.075)	(2, 0.045, 0.017) (4, 0.041, 0.021) (8, 0.131, 0.096)	(2, 0.060, 0.026) (4, 0.118, 0.090) (8, 0.097, 0.056)	(2, 0.091, 0.044) (4, 0.087, 0.043) (8, 0.248, 0.179)	(2, 0.151, 0.083) (4, 0.201, 0.126) (8, 0.342, 0.255)
100	(2, 0.055, 0.019) (4, 0.040, 0.017) (8, 0.104, 0.082)	(2, 0.065, 0.025) (4, 0.068, 0.037) (8, 0.145, 0.116)	(2, 0.088, 0.038) (4, 0.120, 0.076) (8, 0.142, 0.098)	(2, 0.136, 0.067) (4, 0.149, 0.085) (8, 0.300, 0.206)	(2, 0.232, 0.127) (4, 0.258, 0.139) (8, 0.472, 0.263)
144	(2, 0.075, 0.023) (4, 0.057, 0.020) (8, 0.115, 0.053)	(2, 0.089, 0.031) (4, 0.102, 0.058) (8, 0.299, 0.255)	(2, 0.123, 0.052) (4, 0.147, 0.090) (8, 0.272, 0.197)	(2, 0.196, 0.097) (4, 0.218, 0.128) (8, 1.101, 0.971)	(2, 0.338, 0.194) (4, 0.362, 0.200) (8, 0.556, 0.364)
196	(2, 0.088, 0.030) (4, 0.082, 0.038) (8, 0.469, 0.419)	(2, 0.120, 0.053) (4, 0.104, 0.049) (8, 0.580, 0.499)	(2, 0.177, 0.092) (4, 0.161, 0.085) (8, 0.360, 0.274)	(2, 0.293, 0.170) (4, 0.262, 0.141) (8, 0.729, 0.556)	(2, 0.652, 0.457) (4, 0.479, 0.274) (8, 0.877, 0.576)
256	(2, 0.107, 0.036) (4, 0.116, 0.049) (8, 0.609, 0.548)	(2, 0.146, 0.056) (4, 0.153, 0.072) (8, 0.408, 0.306)	(2, 0.222, 0.095) (4, 0.215, 0.105) (8, 1.087, 0.960)	(2, 0.378, 0.177) (4, 0.363, 0.197) (8, 0.826, 0.613)	(2, 0.735, 0.344) (4, 0.714, 0.466) (8, 1.035, 0.624)
324	(2, 0.130, 0.048) (4, 0.145, 0.068) (8, 0.431, 0.357)	(2, 0.192, 0.084) (4, 0.186, 0.093) (8, 0.374, 0.266)	(2, 0.313, 0.153) (4, 0.290, 0.160) (8, 0.552, 0.403)	(2, 0.551, 0.293) (4, 0.510, 0.313) (8, 0.808, 0.550)	(2, 1.008, 0.543) (4, 0.869, 0.530) (8, 1.213, 0.675)
400	(2, 0.155, 0.053) (4, 0.161, 0.068) (8, 0.516, 0.425)	(2, 0.234, 0.090) (4, 0.230, 0.103) (8, 0.355, 0.227)	(2, 0.391, 0.163) (4, 0.371, 0.177) (8, 0.676, 0.494)	(2, 0.690, 0.297) (4, 0.629, 0.307) (8, 0.939, 0.596)	(2, 1.308, 0.583) (4, 1.170, 0.578) (8, 1.579, 0.969)
484	(2, 0.192, 0.073) (4, 0.195, 0.088) (8, 0.564, 0.411)	(2, 0.310, 0.138) (4, 0.326, 0.140) (8, 0.775, 0.624)	(2, 0.533, 0.255) (4, 0.450, 0.212) (8, 0.790, 0.538)	(2, 0.958, 0.464) (4, 0.765, 0.348) (8, 1.216, 0.828)	(2, 1.856, 0.934) (4, 1.480, 0.730) (8, 1.699, 0.966)
576	(2, 0.228, 0.077) (4, 0.216, 0.088) (8, 9.716, 4.113)	(2, 0.367, 0.134) (4, 0.355, 0.141) (8, 6.505, 2.852)	(2, 0.657, 0.233) (4, 0.530, 0.242) (8, 11.405, 6.186)	(2, 1.150, 0.447) (4, 0.947, 0.421) (8, 253.813, 249.015)	(2, 2.300, 0.919) (4, 1.708, 0.743) (8, 9.188, 3.888)

consider the real-life scientific application, Multidimensional Positive Definite Advection Transport Algorithm (*MPDATA*). *MPDATA* is a core component of the *EULAG* (Eulerian/semi-Lagrangian fluid solver) geophysical model [58], which is an established computational model developed for simulating thermo-fluid flows across a wide range of scales and physical scenarios. Here, the speed is a function of sizes of the dimensions representing the computational domain, (l, m, n).

- We envisage use of our algorithms to speedup core data analytical algorithms. They are enumerated below and form active subjects in our current research investigation.
  - k-means and k-medoids, where the computational complexity is $O(N \times k)$ and therefore speed is a function of (N,k), which are the number of observations and the number of clusters.
  - Support Vector Machines (SVM), where the computational complexity is $O(m^3 + m \times N + N \times p \times m)$ and therefore speed is a function of (N, p, m) where (N, p, m) are the number of training cases, the number of predictors, and the number of support vectors.
  - General splines whose computational complexity is $O(N^3)$ and therefore speed is a function of N, the number of observations.
  - Finally, neural networks whose computational complexity is $O(N \times p \times M \times L)$ where speed is a function of N observations, p predictors, M hidden units, and L training epochs.

APPENDIX G
EXPERIMENTAL RESULTS FOR *PARALEPH* FOR VARYING (q, t)

Table 14 shows the results for *PARALEPH* solving *POPT* for DGEMM for multiple configurations of (q, t). The parameter t takes the values, {2,4,8} since each node has four physical cores and eight logical cores. For a given t, the values of q are set to $\frac{p}{t}$. Each cell in the table contains two tuples. Each tuple is represented by $(t, t_{PARALEPH}, t_{comm})$ where t is the number of threads used in the execution of $t_{PARALEPH}$, $t_{PARALEPH}$ is the execution time of *PARALEPH*, and t_{comm} is just the time of communications in *PARALEPH*. For the value of $t = 8$, we can see that there is serious performance degradation. While the configuration $(q, 4)$ performs better than $(q, 2)$, the optimal configuration is $(q, t) = (p, 1)$.

Table 15 shows the results for *PARALEPH* solving *EOPT* for DGEMM for different configurations of (q, t). Table 16 shows the results for *PARALEPH* solving *POPT* for FFTW for different configurations of (q, t). Table 17 shows the results for *PARALEPH* solving *EOPT* for FFTW for different configurations of (q, t).

REFERENCES

[1] R. Reddy and A. L. Lastovetsky. (2017). *PARALEPH: Parallel Data Partitioning Algorithms for Optimization of Data-Parallel Applications on Modern Extreme-Scale Multicore Platforms for Performance and Energy*. [Online]. Available: https://git.ucd.ie/manumachu/PARALEPH

[2] D. Clarke, A. Lastovetsky, and V. Rychkov, "Dynamic load balancing of parallel computational iterative routines on highly heterogeneous HPC platforms," *Parallel Process. Lett.*, vol. 21, no. 2, pp. 195–217, Jun. 2011.

[3] R. D. Williams, "Performance of dynamic load balancing algorithms for unstructured mesh calculations," *Concurrency, Pract. Exper.*, vol. 3, no. 5, pp. 457–481, 1991.

[4] C. Walshaw, M. Cross, and M. G. Everett, "Parallel dynamic graph partitioning for adaptive unstructured meshes," *J. Parallel Distrib. Comput.*, vol. 47, no. 2, pp. 102–108, Dec. 1997.

[5] A. Arulananthan *et al.*, "A generic strategy for dynamic load balancing of distributed memory parallel computational mechanics using unstructured meshed," *Parallel Computational Fluid Dynamics: Development and Applications of Parallel Technology*. 1998, pp. 43–50.

[6] B. Hendrickson and K. Devine, "Dynamic load balancing in computational mechanics," *Comput. Methods Appl. Mech. Eng.*, vol. 184, nos. 2–4, pp. 485–500, 2000.

[7] Z. Chen, J. Dongarra, P. Luszczek, and K. Roche, "Self-adapting software for numerical linear algebra and LAPACK for clusters," *Parallel Comput.*, vol. 29, nos. 11–12, pp. 1723–1743, Nov. 2003.

[8] F. Franchetti, M. Puschel, Y. Voronenko, S. Chellappa, and J. M. F. Moura, "Discrete Fourier transform on multicore," *IEEE Signal Process. Mag.*, vol. 26, no. 6, pp. 90–102, Nov. 2009.

[9] R. Reddy, A. Lastovetsky, and P. Alonso, "HeteroPBLAS: A set of parallel basic linear algebra subprograms optimized for heterogeneous computational clusters," *Scalable Comput., Pract. Exper.*, vol. 10, no. 6, pp. 201–216, Jun. 2009.

[10] C. D. Patel, C. E. Bash, R. Sharma, M. Beitelmal, and R. Friedrich, "Smart cooling of data centers," in *Proc. Int. Electron. Packag. Tech. Conf. Exhib.*, vol. 2, Jan. 2003, pp. 129–137.

[11] C. Bash and G. Forman, "Cool job allocation: Measuring the power savings of placing jobs at cooling-efficient locations in the data center," in *Proc. USENIX Annu. Tech. Conf.*, CA, USA: USENIX Association, 2007, pp. 29:1–29:6.

[12] L. Wang, G. von Laszewski, J. Dayal, X. He, A. J. Younge, and T. R. Furlani, "Towards thermal aware workload scheduling in a data center," in *Proc. 10th IEEE Int. Symp. Pervasive Syst., Algorithms, Netw. (ISPAN)*, Dec. 2009, pp. 116–122.

[13] O. Sarood, A. Gupta, and L. V. Kale, "Temperature aware load balancing for parallel applications: Preliminary work," in *Proc. IEEE Int. Symp. Parallel Distrib. Process. Workshops Phd Forum*, May 2011, pp. 796–803.

[14] M. Atif and P. Strazdins, "Adaptive parallel application resource remapping through the live migration of virtual machines," *Future Gener. Comput. Syst.*, vol. 37, pp. 148–161, Jul. 2014.

[15] OpenBLAS. (2016). *OpenBLAS: An Optimized BLAS Library*. [Online]. Available: http://www.openblas.net/

[16] FFTW. (2016). *FFTW: A Fast, Free C FFT Library*. [Online]. Available: http://www.fftw.org/

[17] PFFTW. (2016). *Parallel FFTW*.[Online]. Available: http://www.fftw.org/fftw2_doc/fftw_4.html

[18] A. Lastovetsky and R. R. Manumachu, "New model-based methods and algorithms for performance and energy optimization of data parallel applications on homogeneous multicore clusters," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 4, pp. 1119–1133, Apr. 2017.

[19] A. L. Lastovetsky and R. Reddy, "Data partitioning with a functional performance model of heterogeneous processors," *Int. J. High Perform. Comput. Appl.*, vol. 21, no. 1, pp. 76–90, 2007.

[20] A. Ilic, 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*, 2010, pp. 202–222.

[21] D. Clarke, A. L. Lastovetsky, and V. Rychkov, "Column-based matrix partitioning for parallel matrix multiplication on heterogeneous processors based on functional performance models," in *Proc. Eur. Conf. Parallel Process.*, in Lecture Notes in Computer Science, vol. 7155. New York, NY, USA: Springer-Verlag, 2012, pp. 450–459.

[22] X. Liu, Z. Zhong, and K. Xu, "A hybrid solution method for CFD applications on GPU-accelerated hybrid HPC platforms," *Future Gener. Comput. Syst.*, vol. 56, pp. 759–765, Mar. 2016.

[23] M. Radmanović, D. Gajić, and R. Stanković, "Efficient computation of galois field expressions on hybrid CPU-GPU platforms," *J. Multiple-Valued Logic Soft Comput.*, vol. 26, no. 3, 2016.

[24] A. Ilic and L. Sousa, "Simultaneous multi-level divisible load balancing for heterogeneous desktop systems," in *Proc. IEEE ISPA*, Jul. 2012, pp. 683–690.

[25] J. Colaço, A. Matoga, A. Ilic, N. Roma, and P. Tomas, 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.

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

[27] R. R. Manumachu and A. Lastovetsky, "Bi-objective optimization of data-parallel applications on homogeneous multicore clusters for performance and energy," *IEEE Trans. Comput.*, vol. 67, no. 2, pp. 160–177, Feb. 2018.

[28] Y. Ogata, T. Endo, N. Maruyama, and S. Matsuoka, "An efficient, model-based CPU-GPU heterogeneous FFT library," in *Proc. IEEE Int. Symp. Parallel Distrib. Process. (IPDPS)*, Apr. 2008, pp. 1–10.

[29] C. Yang *et al.*, "Adaptive optimization for petascale heterogeneous CPU/GPU computing," in *Proc. IEEE Int. Conf. Cluster Comput. (CLUSTER)*, Sep. 2010, pp. 19–28.

[30] M. D. Linderman, J. D. Collins, H. Wang, and T. H. Meng, "Merge: A programming model for heterogeneous multi-core systems," *ACM SIGOPS Oper. Syst. Rev.*, vol. 42, no. 2, pp. 287–296, 2008.

[31] 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," *ACM SIGPLAN Notices*, vol. 44, no. 4, pp. 121–130, Apr. 2009.

[32] C. Augonnet, S. Thibault, and R. Namyst, "Automatic calibration of performance models on heterogeneous multicore architectures," in *Proc. 3rd Workshop Highly Parallel Process. Chip (HPPC)*, Aug. 2009, pp. 56–65.

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

[34] J. M. Bahi, S. Contassot-Vivier, and R. Couturier, "Dynamic load balancing and efficient load estimators for asynchronous iterative algorithms," *IEEE Trans. Parallel Distrib. Syst.*, vol. 16, no. 4, pp. 289–299, Apr. 2005.

[35] A. Legrand, H. Renard, Y. Robert, and F. Vivien, "Mapping and load-balancing iterative computations," *IEEE Trans. Parallel Distrib. Syst.*, vol. 15, no. 6, pp. 546–558, Jun. 2004.

[36] R. L. Cariño and I. Banicescu, "Dynamic load balancing with adaptive factoring methods in scientific applications," *J. Supercomput.*, vol. 44, no. 1, pp. 41–63, 2008.

[37] 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, pp. 151–159, Nov. 2011.
[38] T. Gautier, X. Besseron, and L. Pigeon, "KAAPI: A thread scheduling runtime system for data flow computations on cluster of multi-processors," in *Proc. Int. Workshop Parallel Symbolic Comput.*, 2007, pp. 15–23.
[39] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "StarPU: A unified platform for task scheduling on heterogeneous multicore architectures," *Concurrency Comput., Pract. Exper.*, vol. 23, no. 2, pp. 187–198, Feb. 2011.
[40] G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, P. Lemarinier, and J. Dongarra, "DAGuE: A generic distributed DAG engine for high performance computing," in *Proc. IEEE IPDPSW*, May. 2011, pp. 1151–1158.
[41] H. Han and C.-W. Tseng, "Exploiting locality for irregular scientific codes," *IEEE Trans. Parallel Distrib. Syst.*, vol. 17, no. 7, pp. 606–618, Jul. 2006.
[42] Y. Jo and M. Kulkarni, "Enhancing locality for recursive traversals of recursive structures," *ACM SIGPLAN Notices*, vol. 46, no. 10, pp. 463–482, Oct. 2011.
[43] M. Ravishankar, J. Eisenlohr, L.-N. Pouchet, J. Ramanujam, A. Rountev, and P. Sadayappan, "Code generation for parallel execution of a class of irregular loops on distributed memory systems," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.* Salt Lake City, UT, USA: IEEE Computer Society Press, 2012, pp. 72:1–72:11.
[44] A. Venkat, M. Shantharam, M. Hall, and M. M. Strout, "Non-affine extensions to polyhedral code generation," in *Proc. Annu. IEEE/ACM Int. Symp. Code Gener. Optim.*, 2014, p. 185.
[45] A. Mahanti and D. L. Eager, "Adaptive data parallel computing on workstation clusters," *J. Parallel Distrib. Comput.*, vol. 64, no. 11, pp. 1241–1255, 2004.
[46] I. Galindo, F. Almeida, and J. M. Badía-Contelles, "Dynamic load balancing on dedicated heterogeneous systems," in *Proc. Eur. Parallel Virtual Mach./Message Passing Interface Users' Group Meeting*. Springer, 2008, pp. 64–74.
[47] J. A. Martínez, F. Almeida, E. M. Garzón, A. Acosta, and V. Blanco, "Adaptive load balancing of iterative computation on heterogeneous nondedicated systems," *J. Supercomput.*, vol. 58, no. 3, pp. 385–393, 2011.
[48] J. Sanjuan-Estrada, L. G. Casado, and I. García, "Adaptive parallel interval branch and bound algorithms based on their performance for multicore architectures," *J. Supercomput.*, vol. 58, no. 3, pp. 376–384, 2011.
[49] W.-J. Wang, Y.-S. Chang, C.-H. Wu, and W.-X. Kang, "A self-adaptive computing framework for parallel maximum likelihood evaluation," *J. Supercomput.*, vol. 61, no. 1, pp. 67–83, 2012.
[50] A. Acosta, V. Blanco, and F. Almeida, "Towards the dynamic load balancing on heterogeneous multi-GPU systems," in *Proc. 10th IEEE ISPA*, Jul. 2012, pp. 646–653.
[51] W. Zhang *et al.*, "VarCatcher: A framework for tackling performance variability of parallel workloads on multi-core," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 4, pp. 1215–1228, Apr. 2017.
[52] A.-J. Rico-Gallego, A. Lastovetsky, and J.-C. Díaz-Martín, "Model-based estimation of the communication cost of hybrid data-parallel applications on heterogeneous clusters," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 11, pp. 3215–3228, Nov. 2017.
[53] R. A. Van De Geijn and J. Watts, "SUMMA: Scalable universal matrix multiplication algorithm," *Concurrency, Pract. Exper.*, vol. 9, no. 4, pp. 255–274, 1997.
[54] K. Hasanov, J.-N. Quintin, and A. Lastovetsky, "Hierarchical approach to optimization of parallel matrix multiplication on large-scale platforms," *J. Supercomput.*, vol. 71, no. 11, pp. 3991–4014, Nov. 2015.
[55] A. Lastovetsky and R. Reddy, "On performance analysis of heterogeneous parallel algorithms," *Parallel Comput.*, vol. 30, no. 11, pp. 1195–1216, Nov. 2004.
[56] M. Kondo, "Report on exascale architecture roadmap in Japan," Tech. Rep., 2012.
[57] R. Reddy and A. L. Lastovetsky. (2016). *HCLWattsUp: API for Power and Energy Measurements Using WattsUp Pro Meter*. [Online]. Available: http://git.ucd.ie/hcl/hclwattsup
[58] P. K. Smolarkiewicz and W. W. Grabowski, "The multidimensional positive definite advection transport algorithm: Nonoscillatory option," *J. Comput. Phys.*, vol. 86, no. 2, pp. 355–375, Feb. 1990.

RAVI REDDY MANUMACHU received the B.Tech. degree from IIT Madras in 1997 and the Ph.D. degree from the School of Computer Science and Informatics, University College Dublin, in 2005. His main research interests include high-performance heterogeneous computing, parallel computational fluid dynamics, complexity theory, and cryptography.

ALEXEY LASTOVETSKY received the Ph.D. degree from the Moscow Aviation Institute in 1986 and the D.Sc. degree from the Russian Academy of Sciences in 1997. He has published over 100 technical papers in refereed journals, edited books, and international conferences. He has authored the monographs *Parallel Computing on Heterogeneous Networks* (Wiley, 2003) and *High Performance Heterogeneous Computing* (Wiley, 2009). His main research interests include algorithms, models, and programming tools for high-performance heterogeneous computing.