Chapter 6

Energy-Aware High Performance Computing

Martin Wlotzka, Vincent Heuveline, Manuel F. Dolz,
M. Reza Heidari, Thomas Ludwig,
A. Cristiano I. Malossi and Enrique S. Quintana-Orti

Additional information is available at the end of the chapter

http://dx.doi.org/10.5772/

Abstract

High performance computing centres consume substantial amounts of energy to power large-scale supercomputers and the necessary building and cooling infrastructure. Recently, considerable performance gains resulted predominantly from developments in multi-core, many-core and accelerator technology. Computing centres rapidly adopted this hardware to serve the increasing demand for computational power. However, further performance increases in large-scale computing systems are limited by the aggregate energy budget required to operate them. Power consumption has become a major cost factor for computing centres. Furthermore, energy consumption results in carbon dioxide emissions, a hazard for the environment and public health; and heat, which reduces the reliability and lifetime of hardware components. Energy efficiency is therefore crucial in high performance computing.

This chapter addresses key issues of energy-aware high performance computing. We outline some numerical methods which are often used in scientific applications, and present an energy profiling and tracing technique suitable to analyse the power consumption of applications. The next section is devoted to the performance and energy characterization of the sparse matrix-vector product, a basic numerical building block. Finally, we discuss opportunities for saving energy in computations by means of two examples. First, we present energy-aware runtimes on shared memory multi-core platforms for the Conjugate Gradient method. Second, we present energy-efficient techniques for multigrid methods on distributed memory clusters.

Keywords: high performance computing, energy-aware numerics, energy profiling, energy-aware runtimes, energy-efficient multigrid

1. Introduction

Numerical simulations play a key role in scientific discovery. Modeling and simulation of physical processes is an enabling technology for investigations beyond the scope of theoretical analysis and experiments. Numerical simulations may yield insight in situations where technical, ethical, or financial issues prevent from experiments. This chapter presents an overview of basic methodologies in high performance computing (HPC) used for numerical simulations of physical processes. We discuss energy-aware parallelization techniques both for shared memory multicore platforms, as well as for distributed memory clusters.

We outline a class of discretization methods that can be used to deal with physical models which are described by partial differential equations (PDEs) in Section 2. We demonstrate that solving algebraic systems of equations, and in particular solving linear systems, is at the heart of many simulations. In Section 3, we present techniques for power tracing and analysis of scientific applications. Section 4 is devoted to the energy characterization of the sparse matrix-vector multiplication. This is one of the fundamental numerical building blocks used in many solver methods. In Section 5, we focus on energy-aware runtimes for numerical linear algebra on multicore processors. Finally, we present a distributed memory parallelization of linear algebra operations and energy-efficient techniques for multigrid methods in Section 6.

2. Numerical simulation of physical processes modelled by partial differential equations

We depict the discretization methods and parallelization techniques by means of two prototypical partial differential equations, namely the Poisson's equation and the continuity equation. Poisson's equation can be used to model a steady state temperature distribution in a continuum, where the solution represents the equilibrium state. The time-dependent continuity equation can be used to model the temporal evolution of a conserved physical quantity. In our examples, we restrict to the case of two spatial dimensions. We outline the basic discretization methodology of the finite difference method (FDM) and of the finite element method (FEM) for Poisson's equation, and of the finite volume method (FVM) for the continuity equation. The goal is to show that these three methods have in common the translation of the underlying model equations into a finite-dimensional system of algebraic equations, either linear or nonlinear. The solution to the algebraic equations represents the numerical solution of the model. Thus, performing numerical simulations often requires to solve algebraic systems of equations. Therefore, the linear system solver is the crucial part of many simulations. This is also true for nonlinear systems, since these are often solved by means of Newton-type methods that use a sequence of linear systems to approximate the nonlinear solution.

2.1. Poisson's equation

The spatial domain where the model is defined is denoted by Ω. For our two-dimensional example, we have $\Omega \subset R^2$. Poisson's equation reads

http://dx.doi.org/10.5772/

$$-\Delta u = f \quad \text{in } \Omega, \tag{1}$$

where the unknown solution u represents the equilibrium temperature distribution in Ω, f is a heat source term, and $\Delta = \partial^2/\partial x^2 + \partial^2/\partial y^2$ denotes the two-dimensional Laplace operator. Poisson's equation is usually accompanied by boundary conditions. One can use Dirichlet-type boundary conditions to model given environmental temperatures, or Neumann-type boundary conditions to model given heat fluxes. Theory on existence and uniqueness of solutions can be found in textbooks, e.g., Refs. [1, 2].

2.2. Finite difference discretization of Poisson's equation

The finite difference method approximates derivatives by means of difference quotients, which are evaluated at certain points in the domain Ω. In the simplest case, the method is based on a rectangular grid Ω_h, covering the domain with squares of side length h which lie parallel to the coordinate axes. The grid points are denoted as $x_{i,j}$, where the indices i and j establish a lexicographic enumeration of points in the grid. In this sense, $x_{i-1,j}$ is the left neighbor of $x_{i,j}$ at a distance h and $x_{i,j+1}$ is the upper neighbor at a distance h. **Figure 1** shows an example domain with a grid. An approximation of the Laplace operator can be defined as

$$\Delta u(x_{i,j}) \approx \frac{1}{h^2}\left[u(x_{i-1,j}) + u(x_{i+1,j}) + u(x_{i,j-1}) + u(x_{i,j+1}) - 4u(x_{i,j})\right]. \tag{2}$$

Eq. (2) defines a discrete Laplace operator by means of the five-point stencil, which uses the function values at the point $x_{i,j}$ and its four neighbor points in direction of the coordinate axes.

Writing $u_{i,j} = u(x_{i,j})$ and $f_{i,j} = f(x_{i,j})$, this discretization of the Poisson equation results in the linear system of equations

$$4u_{i,j} - u_{i-1,j} - u_{i+1,j} - u_{i,j-1} - u_{i,j+1} = h^2 f_{i,j}. \tag{3}$$

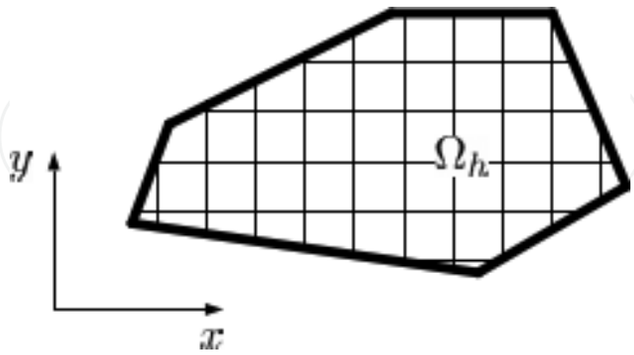

Figure 1. Simple computational grid Ω_h of a polygonal domain Ω.

Reusing the notation, the discretized system can be formulated with a matrix A and vectors u and b as $Au = b$. Here, the components of u and b represent the values of the discrete solution and of the source term at the grid points, respectively. The matrix A has the coefficients from

the left-hand side of Eq. (3) as its entries. Thus, solving Poisson's equation by means of the finite difference method amounts to solving the linear system in Eq. (3) represented as $Au = b$. For error and stability analysis, and for convergence theory, see Ref. [3].

2.3. Finite element discretization of Poisson's equation

The finite element method uses a modified formulation of Poisson's equation. Multiplying Eq. (1) with a test function v and using Green's first identity, one obtains the variational formulation

$$a(u, v) = (f, v) \quad \forall \ v \tag{4}$$

with the bi-linear form $a(u, v) = \int_\Omega \nabla_v \cdot \nabla_u \, dx$, which admits a unique solution in the Sobolev space $H_0^1(\Omega)$ of weakly differentiable functions. Again, a rigorous derivation of the variational formulation, treatment of boundary conditions, and analysis can be found in Refs. [1, 2].

To approximate the solution of Eq. (4), the finite element method uses finite-dimensional function spaces of piecewise polynomials based on a grid Ω_h where we denote the vertices x_i. A simple case of such finite element function spaces of Lagrange-type is the space of piecewise first-order polynomials $V_h = \left\{ v_h \in H_0^1(\Omega) : v_h \Big|_C \in P_1 \ \text{ for all cells } \ C \in \Omega_h \right\}$ resulting in the problem formulation

$$u_h \in V_h : a(u_h, v_h) = (f, v_h) \quad \forall \ v_h \in V_h. \tag{5}$$

The Lagrange finite element basis functions have the property $\phi_i(x_j) = \delta_{ij}$ using the Kronecker symbol, meaning that $\phi_i(x_i) = 1$ and $\phi_i(x_j) = 0$ for $i \neq j$. Therefore, the support of any such basis function ϕ_i, i.e., the region where $\phi_i \neq 0$ comprises only the adjacent cells of the vertex x_i. Expanding the discrete solution $u_h = \sum_{i=1}^n u_i \phi_i$ in terms of an n-dimensional finite element basis with coefficients u_i, Eq. (5) is equivalent to the linear system of equations

$$\sum_{i=1}^{n} u_i a\left(\phi_i, \phi_j\right) = \left(f, \phi_j\right) \ \ (j = 1, ..., \ n). \tag{6}$$

This can again be written as $Au = b$, where $A_{ij} = a(\phi_j, \phi_i)$ and $b_i = (f, \phi_i)$. For an overview of finite element theory, see Ref. [4].

2.4. Continuity equation

The prototype continuity equation reads $\partial_t \rho + \nabla \cdot j = f$, where ρ denotes the density of some conserved physical quantity, j is the flux, and f is a source term. The divergence operator applied to the flux is $\nabla \cdot j = \partial j / \partial_x + \partial j / \partial_y$. As an example for a conserved physical quantity, we consider the amount of some chemical substance which is dissolved in water with a concentration denoted as c. This concentration may vary in time and space. The flux j of the substance transported with a given water flow $\mathbf{v}$ is $j = c\mathbf{v}$. Thus, the equation of continuity for this example reads

$$\partial_t c + \nabla \cdot (c\mathbf{v}) = f \quad \text{in } \Omega \times (0, T). \tag{7}$$

The time interval under consideration is (0, T), and Ω again denotes the physical domain where the model is defined. Note that we omit the discussion of initial and boundary conditions for simplicity here, although they are important for practical applications.

2.5. Finite volume discretization of the continuity equation

The finite volume method is well-suited to treat partial differential equations like the continuity equation [5]. Like the finite difference and the finite element method, it also uses a grid, Ω_h, covering the model domain. In this paragraph, we outline a simple cell-centered finite volume method. To this end, the grid cells $C_i \in \Omega_h$ are enumerated with the index i. Integrating Eq. (7) over a cell and using the divergence theorem yields $\partial_t \int_{C_i} c \ dx + \int_{\partial C_i} c\mathbf{v} \cdot \mathbf{n} \ ds = \int_{C_i} f \ dx$, where $\mathbf{n}$ denotes the outer unit normal vector field on the cell boundary. Denoting the average concentration of the solute in cell C_i as $u_i = \frac{1}{m_i} \int_{C_i} c \ dx$ with cell volume m_i = vol(C_i), the boundary integral term can be approximated as

$$\int_{\partial C_i} cv \cdot \mathbf{n} \ ds = \sum_{j \neq i} A_{i,j} v_{i,j} u_{i,j}. \tag{8}$$

The summation index j runs over all cells adjacent to cell C_i, $v_{i,j}$ is the water velocity from cell C_i to cell C_j, $A_{i,j}$ is the cross-sectional area of the water flow between the two cells, and $u_{i,j}$ is the average concentration of the solute in the cell where the flow originates from. This leads to the spatially discretized system

$$\partial_t (Mu) + Au = b, \tag{9}$$

where the matrix M is a diagonal matrix with $M_{ii} = m_i$, the entries of the matrix A are determined through Eq. (8), and $b_i = \int_{C_i} f \ dx$. The resulting system of ordinary differential equation for the temporal evolution is often treated by means of numerical integrators such as one-step or multistep methods [6, 7]. When using implicit integrators, which is often the case for stiff problems, the computation of the time steps requires to solve linear systems of equations.

2.6. Commonalities

This brief outline on the basic methodology for discretizing partial differential equations by means of the finite difference, finite element, and finite volume method shows important commonalities. All three methods are based on a computational grid which covers the model domain. They transform the infinite-dimensional model problem into a finite-dimensional system of algebraic equations. Our linear example models directly yield linear systems. But also the solution of nonlinear equations is often approximated in Newton-type iteration with a sequence of linear systems. Thus, methods for solving linear systems of equations play the key role in many simulations. The structure of linear problems is induced by the association of the discrete variables with the grid cells or vertices. Couplings between the variables occur only

locally between the neighboring grid cells or vertices. Therefore, the resulting discrete operators and matrices are sparse, i.e., there is only a small number of nonzero entries per row.

2.7. Numerical methods for solving linear systems of equations

As shown above, solving linear systems of equations is at the heart of many scientific applications. Such linear systems may directly arise from the discretization of partial differential equations, or when using a Newton-type iteration to approximate the solution of nonlinear systems. The numerical methods for solving linear systems can be classified in terms of being "direct" or "iterative". Direct linear solvers yield the solution after an a priori known number of computational steps. Prominent members of this class are the LU-, QR-, and Cholesky-factorization. In general, this class includes all methods derived from Gaussian elimination. Intermediate states of the solution vector in direct solver algorithms may be affected with arbitrarily large errors. Therefore, direct solvers must usually be executed entirely until the designated end of the algorithm. The computational complexity of direct methods is $O(n^3)$.

Iterative linear solvers compute a sequence of approximations $x^{(0)} \rightarrow x^{(1)} \rightarrow x^{(2)} \rightarrow \ldots$ which converges to the solution of the linear system. Iterative solvers are usually stopped after a certain number of steps and the recent iteration is taken as approximation to the solution. A typical stopping criterion is when the residual vector norm falls below a given tolerance.

Examples of iterative methods are the classic relaxation schemes such as Jacobi, Gauss-Seidel and successive over-relaxation (SOR), and derived relaxation schemes like asynchronous iteration. Widely used standard iterative solvers are the Krylov subspace methods such as conjugate gradient (CG), general minimal residual (GMRES) and their variants. In practice, Krylov subspace methods are often used with a preconditioner that accelerates the convergence of the sequence of approximations to the solution of the linear system. The computational complexity of Krylov subspace methods is $O(n^2)$ for sparse systems. Another type of iterative linear solver are multigrid methods. These are the most efficient state of art methods with a computational complexity of $O(n)$ for sparse systems. Iterative linear solver algorithms are built on a limited number of basic linear algebra operations. All iterative methods mentioned above use only vector scaling and addition, scalar product, vector norm computation, and matrix-vector multiplication. Therefore, any optimization of a linear algebra routine has a direct effect on the entire linear solver.

The applicability of any particular method depends on the mathematical properties of the system matrix. The matrix properties result from the underlying model equations and the discretization. Some methods such as the LU- and QR-decomposition or the GMRES method are applicable for general nonsingular matrices. Other methods require stronger properties like diagonal dominance or being symmetric positive definite. The latter is the case for the CG method. Textbooks on linear algebra and solvers include Refs. [8, 9].

3. Energy tracing and analysis for parallel scientific applications

The development of exascale systems has exposed the fact that the current technologies, programming practices, and performance metrics are not adequate. Existing tools for HPC

systems mainly focus on monitoring and evaluation of performance metrics. Newer hardware has a wide range of sensors and measurement devices related to the power consumption, with varying granularity and informative value. Recently, new tools have been developed or have been adapted from other research areas (e.g., mobile computing) for analyzing the power consumption. Most of these tools focus only on the power consumption and disregard the performance aspects. Therefore, it is crucial to identify adequate sensors, hardware counters, and measurement devices to gain detailed insights about the power consumption and the performance of the hardware.

In order to optimize energy consumption of scientific applications, enhanced profiling and tracing frameworks combining both power and performance metrics are needed. Moreover, to gain a better understanding of energy usage, performance metrics, such as performance counters or routine events, should be correlated with the power traces. Only with analyzing these measurements, energy inefficiencies in software codes can be localized and optimized. In this chapter, we propose an integrated framework with a modular design to study power and energy profiles/traces of HPC scientific applications. This framework provides support for analyzing the power and performance of different types of parallel applications that run on distributed and shared memory platforms.

3.1. Framework environment

This section describes the software tools of the built-in framework that were developed for performance and energy profiling, and tracing of applications [10, 11]. This approach is based on the postmortem offline analysis as the recorded data is accessed after the application execution. The main advantage of this methodology is that the data can be analyzed many times and compared with other data.

The tracing mechanism normally proceeds out by collecting and analyzing data in order to characterize the application execution and system behavior. The approach to perform this analysis is realized in terms of statistics storage, comprising, e.g., absolute values for the number of invoked routines, the execution time of routines, and the hardware counters. Profiling tools that output these statistics are very useful to analyze the application behavior. Tracing tools are, as well, important to analyze the different phases and behavior of the application over time. Their extension to the power analysis also drives us to include data from the power measurement devices, with the aim of correlating them with the application traces. We use a combination of all these methods.

As shown in **Figure 3**, our proposed performance and energy/power analysis framework is composed of several components, including a power measurement library, a power measurement device, and a set of visualization tools. The scientific application runs on a high performance computing system (HPC), such as a cluster of multicore architecture or a hybrid system that benefits from offload computing parts such as GPGPUs. The application is instrumented with our power measurement library (PMLib), which allows for measuring the power consumption of the machine running the application. The second component is a measurement device which is attached to the target platform. This device is usually either an internal DC or an external AC wattmeter that steadily samples power and sends the output to a tracing

server. The instrumented application makes calls to some API functions from the power measurement library for performing different and complementary tasks such as:

- Instructing the tracing server to start/stop collecting data captured by wattmeters.
- Dump the samples into a disk file (power trace) in a particular format.
- Querying various features of the measurement device, etc.

The last component of the framework is a visualization tool, which is used to analyze the power traces generated by PMLib, once the application finishes its computation. One can combine the power traces with the application performance traces produced by the Extrae tool to make the results more meaningful and visualize the final work with Paraver. Due to the flexibility of the framework and PMLib, it is also possible to take advantage of other tracing tools such as TAU [12], VampirTrace [13], etc.

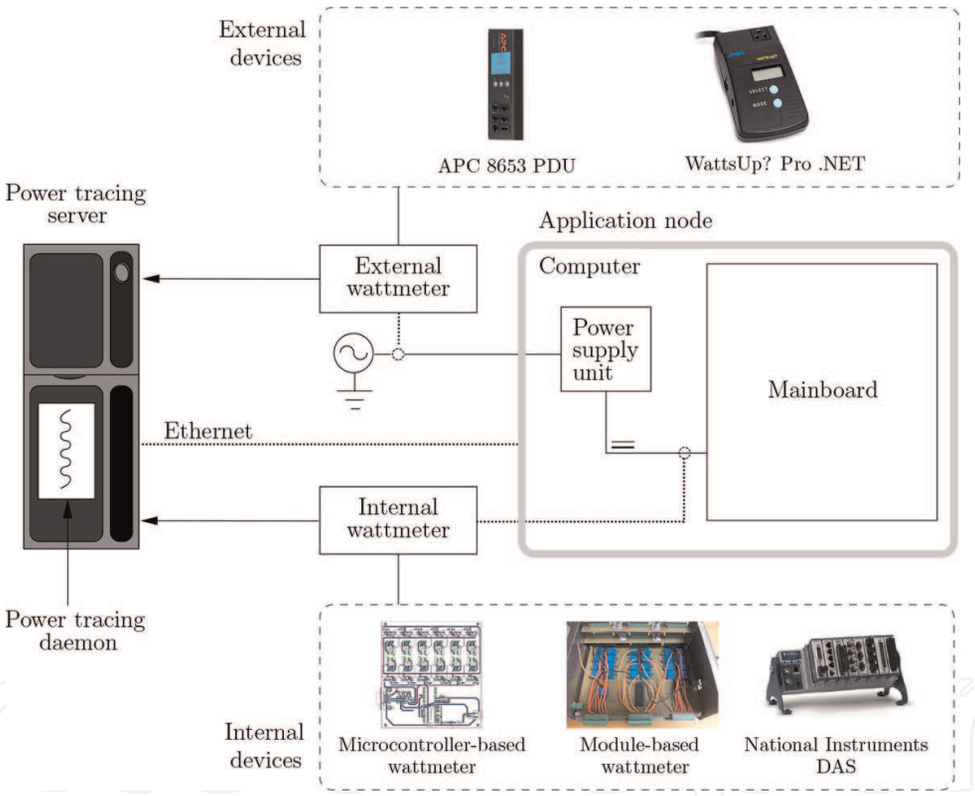

Figure 2. Single-node application system and sampling points for external and internal wattmeters.

3.2. The power measurement library: PMLib

The power measurement library (PMLib) has been designed to analyze the power consumption of HPC applications. The library supports various power measurement devices such as:

- AC wattmeters that connect to the input lines of power supply units of the computing systems and measure the AC input power. These devices include general power distribution units or professional wattmeters such as ZES Zimmer LMG450 or WattsUp?.

- DC wattmeters that can connect to the output DC lines of the power supply units inside the computing systems and measure the DC output power.
- Data acquisition cards such as National Instruments (NI) DAQs.
- Built-in power measurement components such as IPMI, Intel RAPL, and NVIDIA NVML.

The PMLib also offers a set of API interface to allow applications to measure their power consumption. The system and sampling points for external and internal wattmeters are illustrated in **Figure 2**.

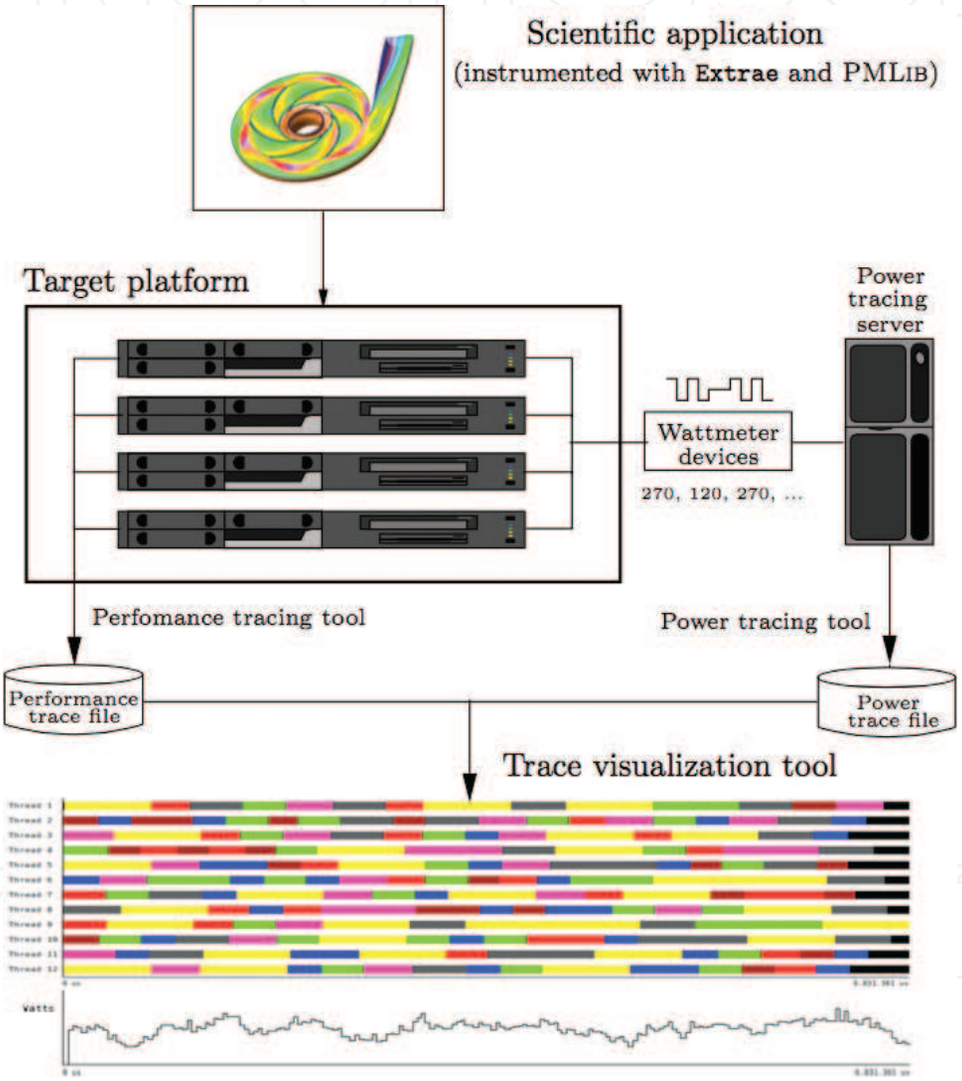

Figure 3. Collecting traces at runtime and visualisation of power-performance data.

3.3. Experimental results

In this section, we provide a detailed power and performance analysis of a dense linear algebra code to demonstrate the use of our performance-power framework on a multicore technology platform. This study offers a vision of the power drawn by the system during the execution of

the LAPACK LU factorization [14]. This factorization is the key to the solution of dense linear systems. In order to collect and illustrate this information, we bind an execution trace of the algorithm obtained with our proposed framework Extrae + Paraver, with our own power evaluation setup, using PMLib and an internal DC wattmeter.

The following experiments were carried out using IEEE double-precision arithmetic on a two 8-core AMD Opteron 6128 processors, running at 2.00 GHz, with 48 GB of DDR3 RAM memory. This experiment benefits from the Intel MKL (v10.3.4) implementation of BLAS. Tracing and visualization were obtained with Extrae (v2.2.0) and Paraver (v4.1.0). In our evaluation, the power readings are collected from an internal DC wattmeter at a sampling rate of 28 Sa/s. The wattmeter is directly attached to the 12 V lines that connect the PSU to the motherboard (chipset plus processors) of the test platform. Therefore, the results are not affected by inefficiencies of the PSU, or the "noise" due to the operation of other hardware components such as fans, disks, network interfaces, etc.

The LAPACK implementation for the LU factorization with partial pivoting (dgetrf) was evaluated during our experiments. In this case, the parallelism is exploited within the invocations to the multithreaded MKL BLAS. The block size in LAPACK routines was set to 128 as it always delivers performance figures close to the optimal. **Figure 4** depicts

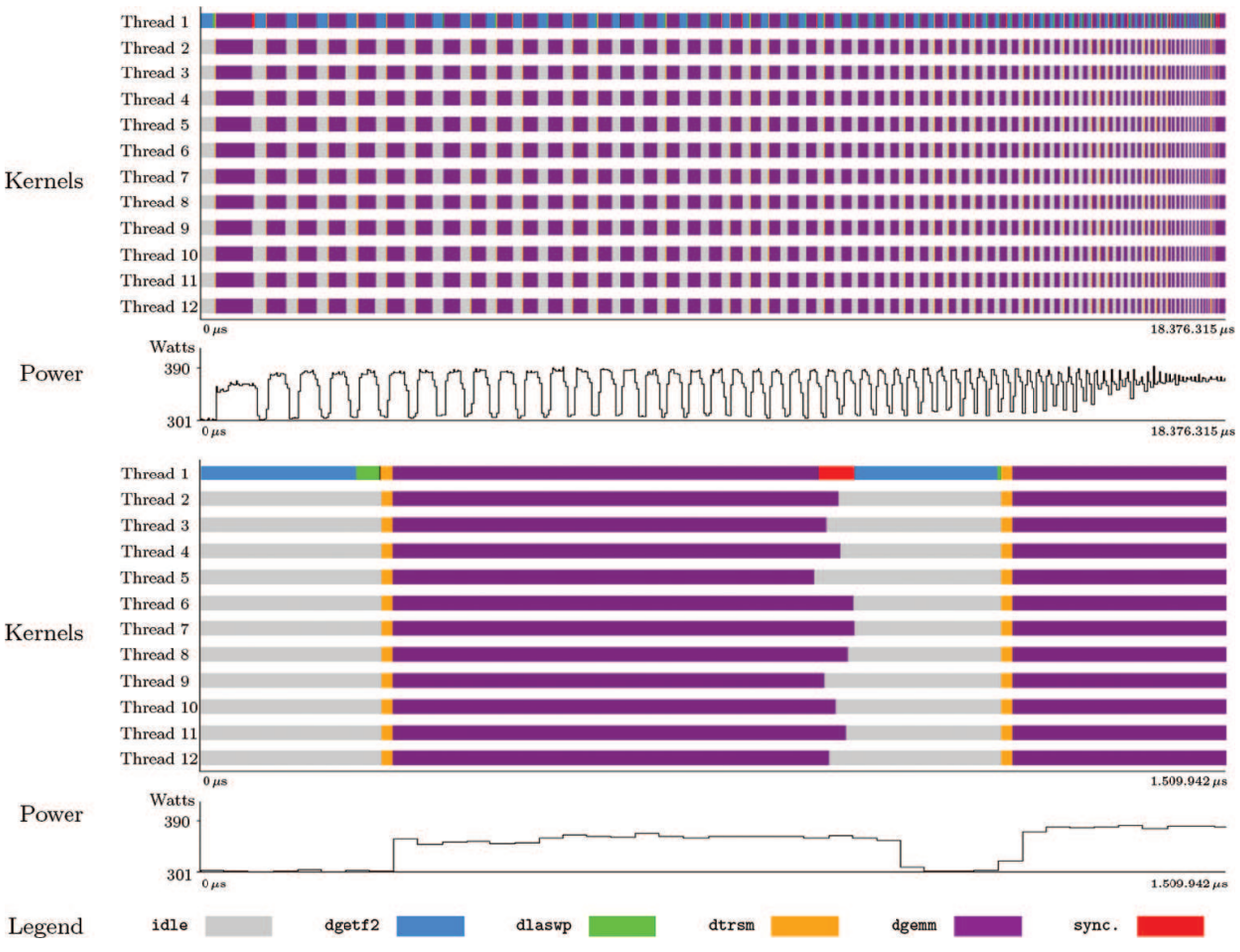

Figure 4. Trace of LAPACK dgetrf. Top: full. Bottom: first two iterations.

the activities of the cores and the power consumption traces during the execution of this routine, responsible for computing the LU factorization. In the trace, the colors identify the different kernels called by the main dgetrf routine: dgetf2 (factorization of the current panel), dlaswp (interchanging rows), dtrsm (triangular system solve), dgemm (matrix-matrix multiplications to calculate the trailing submatrix). The top trace in the figure shows the complete execution of dgetrf, while the bottom trace indicates the first two main iterations of the routine. As can be seen in the traces, the combination of this LAPACK routine along with the multithreaded MKL BLAS leads to interlaced sequential and parallel regions. It is also remarkable that the kernels, dgetf2 and dgemm, mostly monopolize the execution time of the routine. The zoomed trace in the bottom part reveals unbalanced loads among the execution units for the dgemm kernels, leading therefore to the need for the synchronization of threads. A closer look at the power traces highlights the fact that the recurring sequential and concurrent phases cause the power consumption to toggle repeatedly between 301 and 390W. For more details, refer to Ref. [15].

4. HPC energy metric and performance characterization of the sparse matrix-vector product

HPC metrics are supposed to drive system architects in the development of new hardware and supercomputers. With this respect, High performance Linpack (HPL) Linpack and the Top500 [16] have done a great job over the last 20 years, however with the advent of the power-wall and the imminent end of Moore's law, things started to change quite radically. Energy and power consumption are considered today as primary design parameters, together with FLOP/S and pure performance. For this reason, in 2007, the Green500 [17] ranking has been introduced, where FLOP/S have been normalized over power consumption. The new metric has finally moved some attention towards the power and energy consumption, however, the picture that it provides is still quite far from reality. This is visible in **Figure 5**, where FLOPS/W performance of the first system in Top500 and Green500 are compared between 2007 and today's. From the picture, it is clear that the 3.5x progress in last 5 years of Green500 has not propagated at all to the top systems in Top500. The motivation behind this behavior is that Green500 does not account for scalability and problem size. In other words, a small nonscalable system that barely enters Top500 could, in principle, rank first in Green500 due to a favorable MFLOPS/W ratio. Indeed, all top 10 systems in Green500 are rather small systems, with a consumption of O (100 kW), as shown in **Figure 6**.

The above scenario tells us that more has to be done at the level of the metrics to correctly capture real computational patterns. We need metrics that promote scalability rather than FLOPS. In other words, we need metrics that measure time- and energy-to-solution, with respect to problem size. This must also be accompanied by an increased typology of benchmarks, to capture all the variety of computations performed in real world applications.

To provide an example of the complexity and variety of the problem, in this chapter, we analyze the sparse matrix-vector product (SpMV). Sparse matrices appears in a lot of

applications, with strong link to finite element models for partial differential equations, numerical methods for boundary value problems and also in economic modeling, ranking search methodologies for the web, and information retrieval. The sparse matrix-vector product (SpMV) has a central role in many of these applications [18], and is a key ingredient to address large-scale sparse linear systems and eigenvalue problems via iterative methods [8, 9]. Due to the relevance of SpMV in scientific computing, we pursue the accurate characterization of this operation on multithreaded architectures, from the point of view of three performance metrics, i.e., time, power, and energy. In this section, we present the most important aspects of Ref. [19].

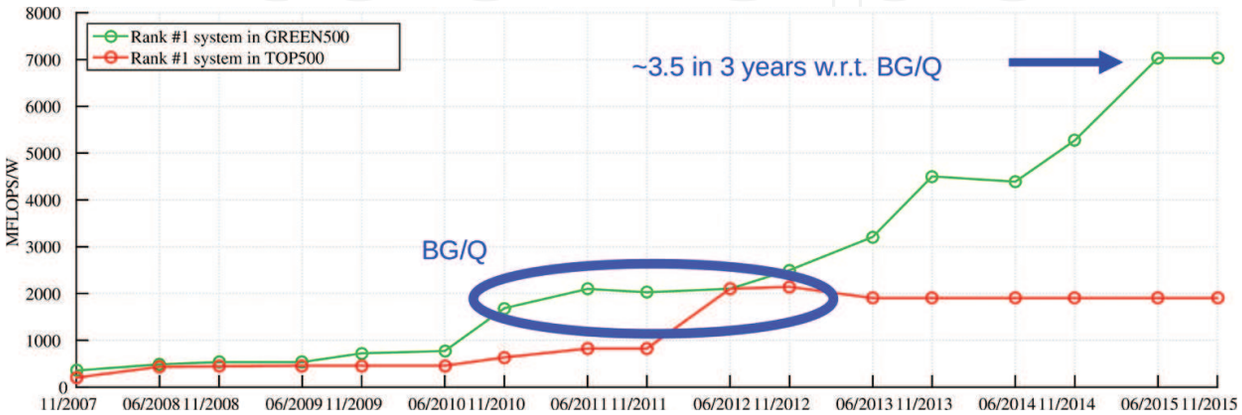

Figure 5. Comparison of the FLOPS/W performance of the no. 1 systems of the Top500 and the Green500 list.

Green500 Rank	MFLOPS/W	Site*	Computer*	Total Power (kW)
1	7,031.58	Institute of Physical and Chemical Research (RIKEN)	Shoubu - ExaScaler-1.4 80Brick, Xeon E5-2618Lv3 8C 2.3GHz, Infiniband FDR, PEZY-SC	50.32
2	5,331.79	GSIC Center, Tokyo Institute of Technology	TSUBAME-KFC/DL - LX 1U-4GPU/104Re-1G Cluster, Intel Xeon E5-2620v2 6C 2.1GHz, Infiniband FDR, NVIDIA Tesla K80	51.13
3	5,271.81	GSI Helmholtz Center	ASUS ESC4000 FDR/G2S, Intel Xeon E5-2690v2 10C 3GHz, Infiniban FDR, AMD FirePro S9150	57.15
4	4,778.46	Institute of Modern Physics (IMP), Chinese Academy of Sciences	Sugon Cluster W780I, Xeon E5-2640v3 8C 2.6GHz, Infiniband QDR, NVIDIA Tesla K80	65.00
5	4,112.11	Stanford Research Computing Center	XStream - Cray CS-Storm, Intel Xeon E5-2680v2 10C 2.8GHz, Infiniband FDR, Nvidia K80	190.00
6	3,856.90	IT Company	Inspur TS10000 HPC Server, Xeon E5-2620v3 6C 2.4GHz, 10G Ethernet, NVIDIA Tesla K40	58.00
7	3,775.45	Internet Service	Inspur TS10000 HPC Server, Intel Xeon E5-2620v2 6C 2.1GHz, 10G Ethernet, NVIDIA Tesla K40	110.00
8	3,775.45	Internet Service	Inspur TS10000 HPC Server, Intel Xeon E5-2620v2 6C 2.1GHz, 10G Ethernet, NVIDIA Tesla K40	110.00
9	3,775.45	Internet Service	Inspur TS10000 HPC Server, Intel Xeon E5-2620v2 6C 2.1GHz, 10G Ethernet, NVIDIA Tesla K40	110.00
10	3,775.45	Internet Service	Inspur TS10000 HPC Server, Intel Xeon E5-2620v2 6C 2.1GHz, 10G Ethernet, NVIDIA Tesla K40	110.00

Figure 6. Snapshot of the Green500 list.

4.1. Sparse matrices parameterisation

Let us consider the SpMV operation $y = Ax$, where A is an $n \times n$ sparse square matrix, with nz nonzero real elements, and x, y are both real dense vectors of dimension n. We assume that A is stored in compressed sparse row (CSR) format, using two integer vectors of dimension nz and $n + 1$ for the indices, and a real vector of size nz for the matrix entries [8, 9].

We do not limit our analysis to a specialized type of matrices (e.g., arrow, banded, tridiagonal, etc.) nor to a specific class of problems (e.g., matrices arising in electrical problems, structural problems, computational chemistry, …). Our first goal is to analyze the problem and to identify a small set of parameters that capture the main "sparsity properties" of SpMV. A couple of these parameters, n and nz, are immediate and already appeared during the initial description of the problem. They are relevant because they determine the starting location of the matrix and vectors in the memory hierarchy that in turn has a big impact on the performance of SpMV. However, these quantities are not sufficient to describe in detail the sparsity pattern and they do not lead to an effective parameterization for our model. We thus introduce three complementary parameters (two of them normalized or nondimensional) to distinguish between matrices with similar dimension and number of nonzeros, but different sparsity patterns can later end in distinct characterization groups. In particular, for our CSR-based implementation of SpMV, we distinguish:

1. Block size: In many applications the nonzeros are clustered into a few compact dense blocks. The block size bs specifies the number of columns in these blocks. This parameter is important, because it captures the number of elements in vector x that are accessed with unit stride, which generally renders a better exploitation of data prefetching and the cache memory.

2. Block density: $bd = bs/nzr \in [0, 1]$ is the fraction of the nonzeros per row, nzr, occupied by a single block. This parameter is relevant because, with bs fixed, it characterizes the reuse factor in the access to vector y, i.e., for an average number of floating-point arithmetic operations performed each time, an element of vector y is loaded into a register (specifically, nzr FLOP).

3. Row density: $rd = nzr/n \in [0, 1]$ is the number of nonzeros per row relative to the row size. This ratio is orientational of the level in the memory hierarchy where the accesses to vector x occur. With bs, bd (and, consequently, nzr) fixed, increasing the row density rd necessarily implies a reduction in the problem dimension (and vice versa).

All these parameters will vary row-wise (bd and rd), and block-wise (bs). Several averaging techniques can be used to extract a single triplet (bs, bd, rd) from an entire nonuniform (irregular) sparse matrix.

4.2. Classification of matrices

We establish here a simple classification for SpMV (and thus for sparse matrices) with respect to the parameters previously defined, as well as according to performance metrics. We define a reference training set made of sparse matrices, with a uniform (though sparse) nonzero

structure. In detail, the matrices in this set have a constant number of nonzeros per row *nzr* for all the rows, a fixed block size *bs* for all the blocks, and therefore the same number of blocks per row bd^{-1}, even though the position of the blocks inside each row is different and randomly assigned, with the only constraint that two blocks must be separated by at least one null entry.

To cover the full range of cases up to the memory capacity, while limiting the number of matrices in the set, we distribute the matrix instances equally in the $\log_2$ space:

- $bs = 2^0, 2^2, 2^4, \ldots$ up to 2^{12} on BG/Q and 2^{14} on P775,
- $bd = 2^0, 2^{-2}, 2^{-4}, \ldots$ down to 2^{-12} on BG/Q and 2^{-14} on P775,
- $rd = 2^0, 2^{-2}, 2^{-4}, \ldots$ down to 2^{-24} on BG/Q and 2^{-28} on P775.

Given that $bs \leq nzr \leq n$, we have a total of 118 samples on the IBM BG/Q and 175 samples on the IBM P755. **Figures 7** and **8** illustrate a compact 3-D representation of the training set, where each matrix is identified by a different point (*bs*, *bd*, *rd*) in the 3-D space.

This coarse training set gives enough variability to characterize sparse matrices from real applications. Nevertheless, we recognize that there exists a clear balance between training cost and accuracy.

We now classify the matrix instances of the training set into four groups, discriminating the training samples where the data fits into last level cache (LLC) from those that have to rely on DDR memory, while keeping the executions with one and four threads per core separated. Using a *k*-means clustering algorithm [20] (in the following we use $k = 2$), we establish a classification into *k* clusters per group for time, power, and energy. All measures are normalized a priori with the standard deviation because the algorithm relies on Euclidean distances.

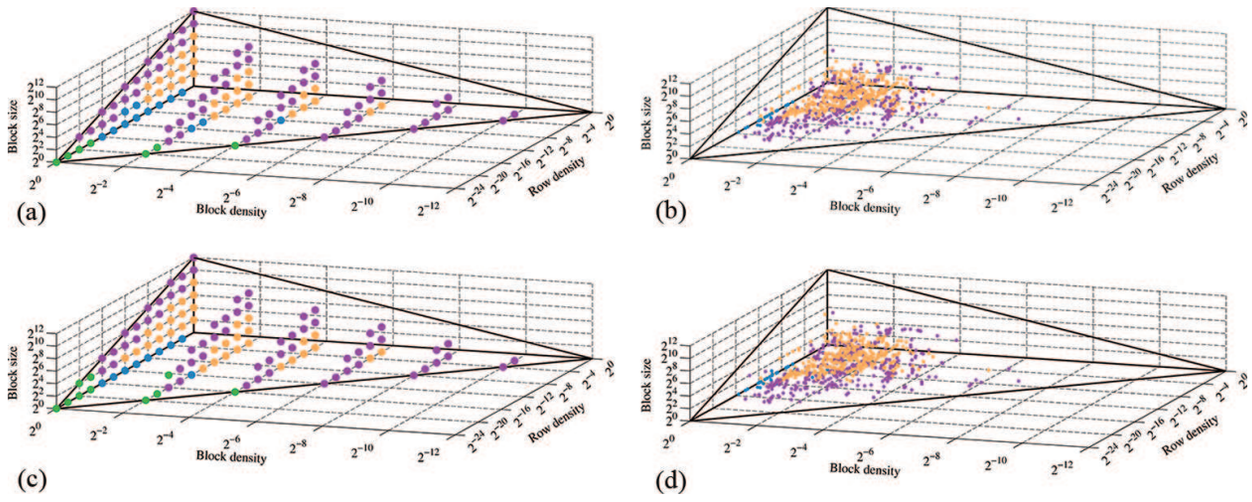

Figure 7. Matrix classification with respect to the triplet coordinated (*bs*, *bd*, *rd*) on the IBM BG/Q.

The left-hand side graphs in **Figures 9** and **11** illustrate the correlation between time and net power/energy. The resulting classification, summarized in **Table 1**, demonstrates that the *k*-means method is able to identify behavioral patterns in the time-power-energy triangle. At the

same time, **Figures 7a** and **8a** show that the four classes are clustered into precise regions in the 3-D space defined by the triplet coordinates (*bs*, *bd*, *rd*). As we can see the classification holds with respect to the parameterization introduced in Section II and, in consequence, it can be leveraged to obtain a fast, qualitative prediction of low/medium/high time-power-energy behavior for any sparse matrix, as a function of few pattern information. In addition, we note that the same classification holds independently with respect to the number of threads per core.

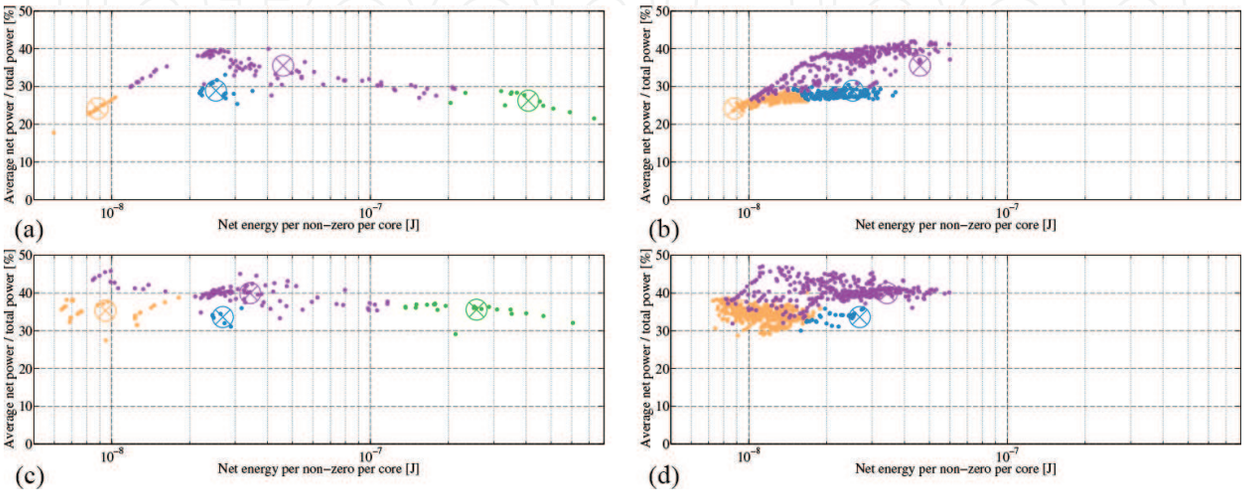

Figure 8. Matrix classification based on time, power, and energy measures on the IBM BG/Q. The position of the centroid of each is marked with a big crossed circle.

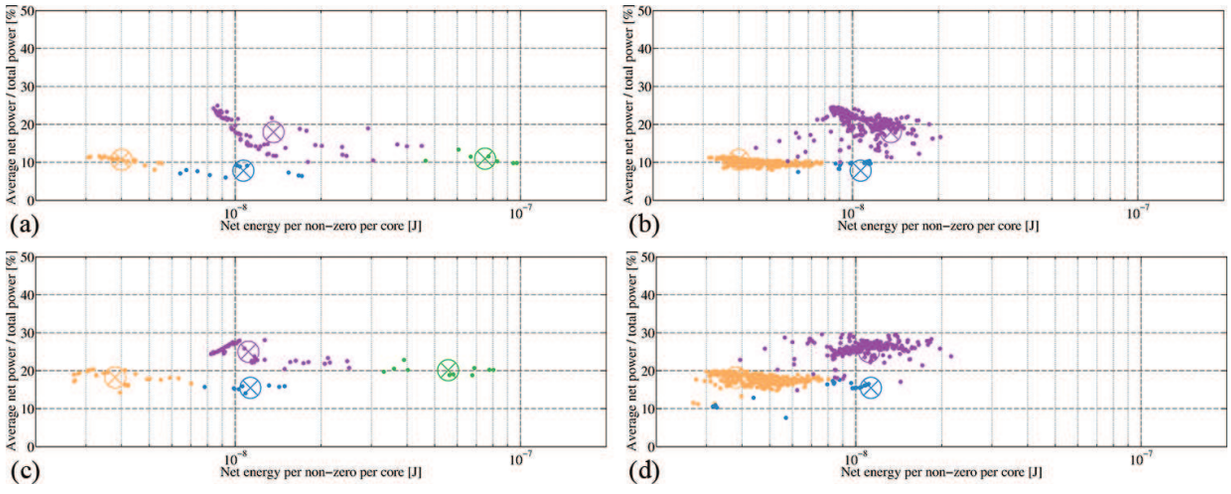

Figure 9. Matrix classification with respect to the triplet coordinated (*bs*, *bd*, *rd*) on the IBM P755.

We can also observe that from the point of view of performance: There is a vertical separation (both in power and energy) between the classes, depending on whether the problem data fits into the LLC or not. This is due to the additional energy required to move data to/from the DDR. The use of 4 threads per core increases the power but reduces the energy (mainly for the DDR cases). Energy increases linearly (in a log scale) with respect to time, while power generally decreases linearly. Inspecting **Figures 7a** and **8a**, we notice that, although real

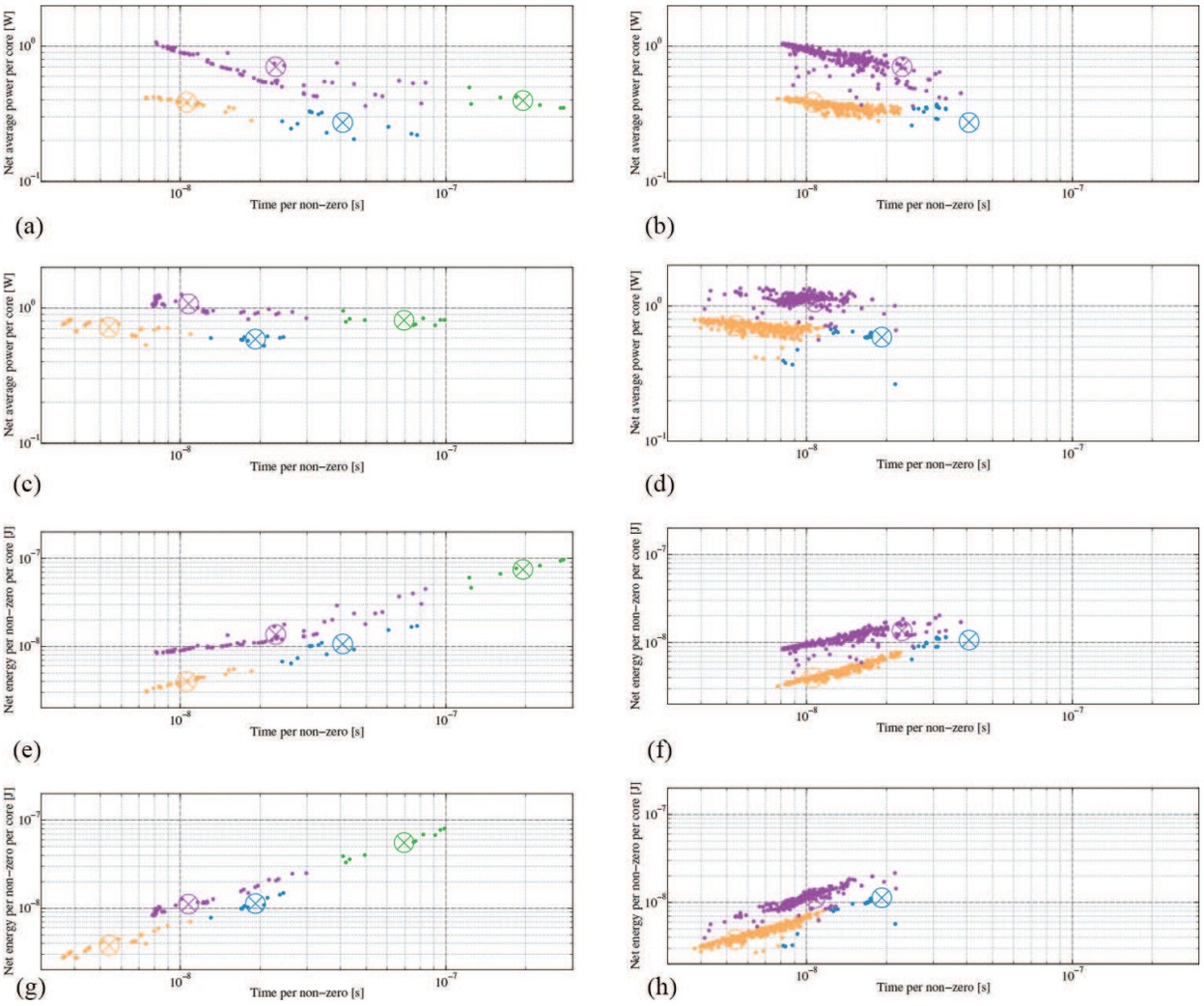

Figure 10. Percent of average net power consumption vs. net energy on the IBM BG/Q. Colors and centroids according to the matrix classification in **Figure 10**.

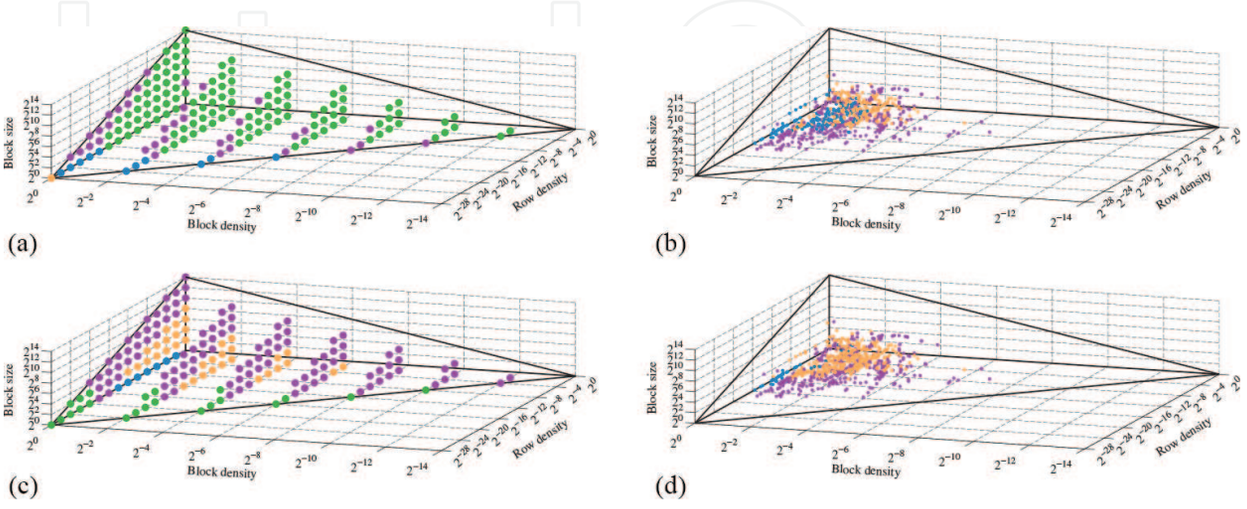

Figure 11. Percent of average net power consumption vs. net energy on the IBM P755. Colors and centroids according to the matrix classification in **Figure 12**.

matrices have a similar power vs. time behavior, energy performance is rather different. This hints to the fact that the FLOPS/W metric used in the Green500 with Linpack, is actually inappropriate to represent energy efficiency of general algorithms [21, 22]. Finally, **Figures 10** and **12** show a direct view over the power vs. energy consumption relation; here, the importance of the net power (especially on the P755) with respect to the overall power consumption is highlighted.

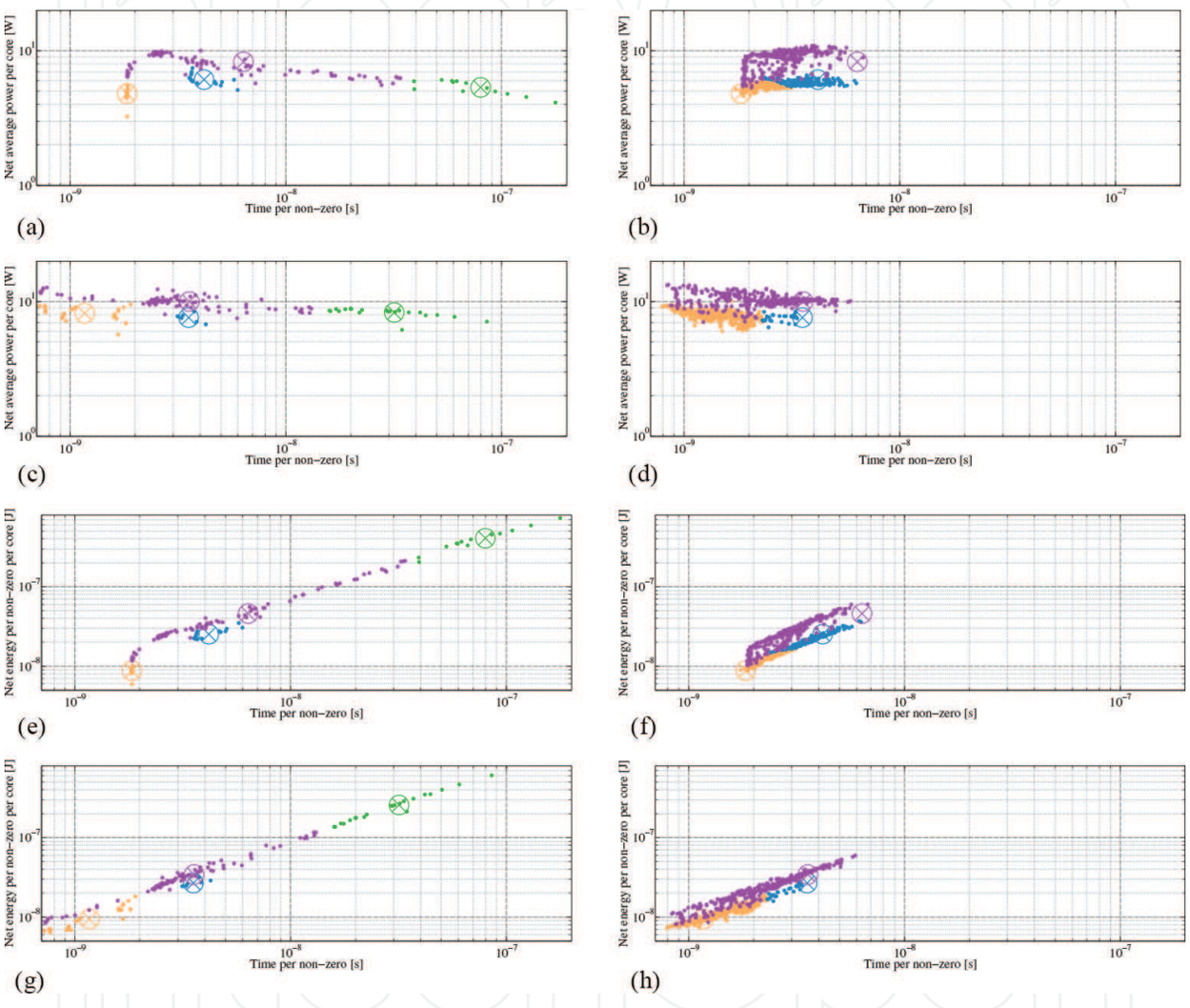

Figure 12. Matrix classification based on time, power, and energy measures on the IBM P755. The position of the centroid of each class is marked with a big crossed circle.

4.3. Validation

We evaluate our classification using the entire University of Florida Sparse Matrix Collection [23]. From this benchmark of real applications, we exclude complex and nonsquare matrices, as well as matrices with empty rows and the cases that do not fit in the target architectures due to DDR capacity restrictions. The resulting number of matrices available for validation consists of 1193 and 1202 samples for IBM BG/Q and IBM P755, respectively.

Class	Color	Memory	Time	Power	Energy
IBM BG/Q					
1		Cache	Low	Medium	Low
2		Cache	Medium	Low-medium	Medium
3		DDR	Low-medium	Medium-high	Medium-high
4		DDR	High	Medium	High
IBM P755					
1		Cache	Low	Low-medium	Low
2		Cache	Medium	Medium	Medium
3		DDR	Low-medium	Medium-high	Low-medium
4		DDR	High	Low-medium	High

Table 1. Qualitative behavior of each class of matrices.

For both architectures, we have measured time, power, and energy of the entire collection. This is displayed in the right-hand side plots of **Figures 10** and **12** (see also **Figures 7b** and **8b** for a 3-D representation), where colors (classes) have been assigned using the same *k*-means classification applied to the training sets (indeed, note that the centroids are the same in the left- and right-hand side images). The cross comparison of colors (i.e., classes) and values (i.e., time-power-energy measures) between the left and right graphs demonstrates that our classification strategy (based on coarse regular training sets) accurately captures a broad range of matrices from real applications for both architectures under investigation. In other words, the classification in **Table 1** offers a cheap, reliable, fast, and simple strategy to qualitatively determine time, power, and energy consumption for SpMV.

4.4. Conclusions

HPC supercomputers, and particularly cloud computing centers, can tremendously benefit from the existence of automatic tools to administer and optimize execution performance and cost of continuous and large stream of parallel tasks (from many different users). We have devised a systematic machine learning algorithm to classify and predict the performance costs of the SpMV kernel. The validity, accuracy, and robustness of our strategy have been demonstrated over a wide database of real matrices. More importantly, behind all the technical details, this approach actually represents a first concrete step towards the development of a global tool, which is able to characterize and capture the features of any sparse linear algebra operation (with a straight-forward extension to the dense case), thus covering a significant percentage of existing scientific computing kernels.

5. General-purpose multi-core servers: energy-aware runtimes for (sparse) linear algebra

Linear algebra operations and, in particular, sparse linear systems are a fundamental building block in many scientific and engineering applications. It is not surprising, therefore, that

considerable effort has been spent by the scientific and high performance community towards the design and optimization of numerical methods for the solutions of these sparse problems, which can exert significant impact on the efficient solution of the applications that are built upon them.

The conjugate gradient (CG) method is among the most effective Krylov subspace solvers for sparse symmetric positive definite (SPD) linear systems [8, 9]. Furthermore, when the problem features only a few right-hand side vectors, the method has been also proven to be highly competitive for the solution of dense linear systems [21]. In this section, we illustrate the positive effects of integrating energy efficiency techniques into numerical software for linear algebra by considering the solution of sparse linear systems via ILUPACK (incomplete LU package) [24] as a workhorse. ILUPACK is a numerical software based on Krylov-based iterative methods that implements multilevel ILU preconditioners for general, symmetric indefinite, and Hermitian positive definite systems, in combination with inverse-based ILUs and Krylov subspace solvers. In other words, ILUPACK provides an implementation of the CG method (for sparse SPD linear systems) furnished with a very sophisticated preconditioner.

5.1. Energy-efficient processor technology

Current processors have adopted tools and technologies, originally designed for embedded systems and the mobile market in order to improve the energy efficiency. As of today, most processors adhere to the advanced configuration and power interface (ACPI) standard [25], which allows to configure the system state depending on the workload, and thus offers a tool to tune the power usage to the actual needs. Concretely, the CPU defines two types of energy-related states, which are reviewed next.

P-states. The ACPI standard defines a collection of operating voltage-frequency pairs for the processor, referred to as the *performance states* (P-states). The number of P-states and their granularity (processor vs. core) depends on the specific processor architecture. For example, for some architectures, the P-states can only be set for all cores in the socket (e.g., Intel Xeon E5504), but in others each core can be set to a distinct P-state (AMD 6128).

Table 2 shows the voltage-frequency pairs associated to the different states available in two recent multi-core processors. As a general rule, increasing the frequency of compute-bound operations reduces the execution time. For memory-bound operations, tampering with the frequency could be expected to have no effect on the execution time. However, in some processors (e.g., the AMD 6128), the memory bandwidth is connected with the processor frequency, and increasing/reducing the frequency can actually have an analogous effect in the execution time.

C-states. The ACPI standard also defines the *processor* or *CPU power states* (C-states). This energy-saving mechanism determines the conditions to turn off certain parts of the processor. State C0 corresponds to a processor that is in normal mode of operation. Compared with this, in all other states (C1, C1E, C2…), power is shut down for certain components such as the lower levels of the memory hierarchy. A deeper C-state will surely save power, but also increases the latency to transition back to the active C0 state. The programmer can only set

the appropriate conditions in the application so that, when the processor is idle, the operating system promotes it to an energy-efficient C-state.

For the examples in **Table 2**, the AMD 6128 presents three C-states: C0, C1, and C1E; the Intel Xeon E5504 has four C-states: C0, C1, C3, and C6.

	Intel E5504 (4 cores)		**AMD 6128 (8 cores)**	
	Voltage (V)	**Frequency (GHz)**	**Voltage (V)**	**Frequency (GHz)**
P0	1.04	2.00	1.23	2.00
P1	1.01	1.87	1.17	1.50
P2	0.98	1.73	1.12	1.20
P3	0.95	1.60	1.09	1.00
P4	n.a.	n.a.	1.06	0.80

Table 2. P-states of two recent processors, from Intel and AMD.

5.2. Task-parallel runtimes for linear algebra operations and general applications

In recent years, a number of runtimes have been proposed to alleviate the burden of programming multi and many threaded platforms. Concretely, OmpSs, StarPU, Mentat, Kaapi, and Harmony, among others, have followed the approach pioneered by Cilk, offering implicit parallel programming models with dependence analysis. When applied to dense linear algebra (DLA) operations, SMPSs (a precursor of OmpSs), StarPU, Quark, and SuperMatrix have demonstrated the advantage of extracting task-parallelism using this "runtime approach". The idea underlying these runtimes to leverage the task-parallelism of a DLA operation consists in decomposing the computations into a collection of tasks connected via data dependencies. In a subsequent stage, the runtime issue these tasks for execution following an out-of-order schedule that maximizes concurrency, while taking into account the dependencies; for example, see Ref. [26] for details. These ideas have been proven useful and the new OpenMP standard has integrated some preliminary primitives in order to integrate task parallelism [27].

5.3. Parallelizing ILUPACK on multicore processors

A potential approach to tackle the solver underlying ILUPACK consists in exploiting task parallelism via a runtime, yielding a dynamic schedule of the work to the cores, with numeric properties similar to those of the sequential ILUPACK. This approach is similar to those employed in DLA, but considerably more difficult due to the irregular data structures that are involved in an iterative sparse linear system solver.

Figure 13 reports the algorithm for the preconditioned conjugate gradient (PCG) method. There, the computation of the preconditioner, M, is the initial step of the solver (O0). The iteration, after this first step, is composed of a sparse matrix-vector product (SpMV, O1), the application of the preconditioner (O5), and several vector operations (dot products, axpy-like updates, 2-norm; in O2–O4 and O6–O8). For simplicity, we regard both the computation of the

preconditioner and its application as "black boxes". In practice, these are by far the most difficult operations.

$A \rightarrow M$	O0. Preconditioner computation
Initialize $x_0, r_0, z_0, d_0, \beta_0, \tau_0; k := 0$	
while $(\tau_k > \tau_{\max})$	Loop for iterative PCG solver
$w_k := Ad_k$	O1. SpMV
$\rho_k := \beta_k / d_k^T w_k$	O2. DOT product
$x_{k+1} := x_k + \rho_k d_k$	O3. AXPY
$r_{k+1} := r_k - \rho_k w_k$	O4. AXPY
$z_{k+1} := M^{-1} r_{k+1}$	O5. Apply preconditioner
$\beta_{k+1} := r_{k+1}^T z_{k+1}$	O6. DOT product
$d_{k+1} := z_{k+1} + (\beta_{k+1}/\beta_k) d_k$	O7. AXPY-like
$\tau_{k+1} := \| r_{k+1} \|_2$	O8. vector 2-norm
$k := k + 1$	
endwhile	

Figure 13. Algorithmic formulation of the preconditioned CG method.

The concurrency implicit in the iterative solver underlying ILUPACK, can be explicitly exposed by applying nested dissection to the adjacency graph associated with the sparse coefficient matrix of the linear system. Concretely, by recursively applying a divide-and-conquer strategy to this graph, we can obtain a hierarchy of subgraphs that reveals the concurrency of the operation. The parallelism enabled by this splitting is in practice captured as a (directed) task-dependency tree, with nodes representing tasks and arcs specifying the dependencies between pairs of them, as shown in **Figure 14**.

The multithreaded execution of the task tree implicit in ILUPACK, on a multicore processor, can be then left in the hands of a runtime which dynamically maps tasks to threads (cores), taking into consideration the dependencies. The runtime can aim to improve different criteria, including, e.g., balancing the workload distribution during the computation of the ILU preconditioner and the subsequent solution of the triangular linear systems involved in the PCG (see **Figure 13**). The runtime keeps track of the ready tasks (i.e., tasks with all their dependencies fulfilled), which initially contain those tasks corresponding to the independent subgraphs (leaves of the tree). The threads update this information as they complete the execution of tasks allocated to them. From the implementation perspective, this information is maintained in shared data structures, and the concurrent access is carefully controlled via lightweight synchronization system calls.

In addition to the concurrency intrinsic to the calculation and application of the preconditioner, the operations that appear in the iterative PCG solve define a partial order

which enforces a strict execution order. In the actual implementation, one can eliminate the explicit barriers between these operations by instead relying on a runtime which can accommodate the nested parallelism exhibited by the task/subtask dependencies. In such scenario, the nested variant defines O1 + O2, O3 + O4, O5, O7, and O6 + O8 as five coarsegrain tasks, and offloads the complete detection and control of the dependencies to the runtime. In addition, these five macrooperations can be further divided into fine grain subtasks, and merge pairs of them as described above.

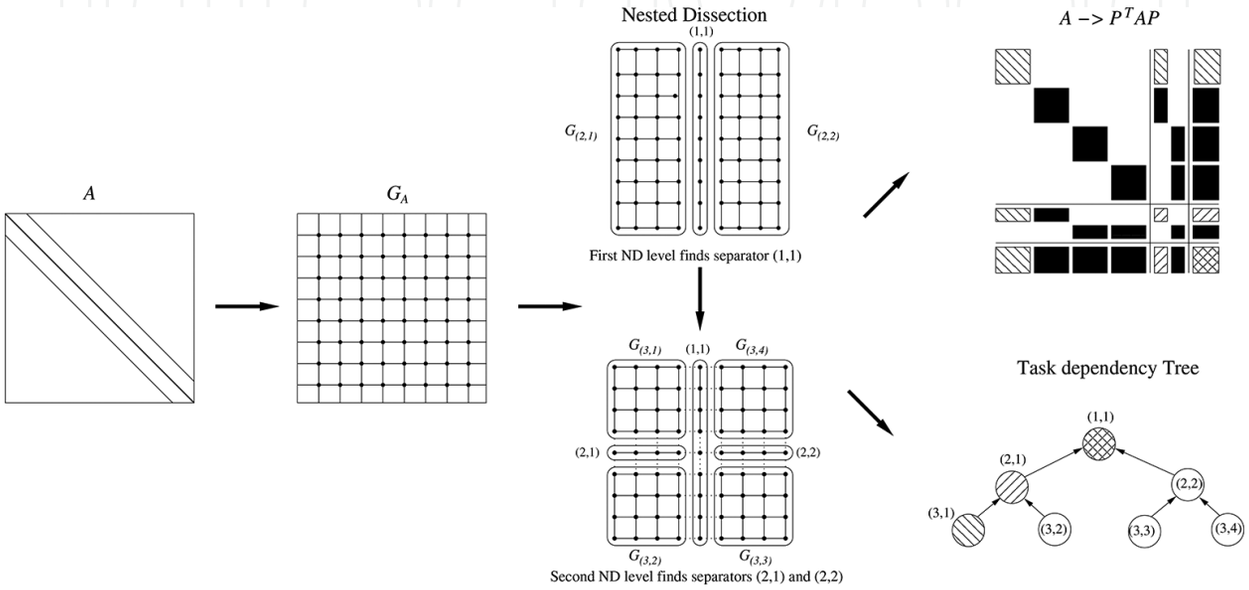

Figure 14. Nested dissection applied to the adjacency graph associated with a sparse matrix and corresponding task dependency tree.

For current NUMA architectures, the application records where (i.e., the socket) each task was executed on during the initial calculation of the preconditioner to achieve that tasks which operate on the same data that was generated/accessed during the preconditioner calculation and the PCG solve are mapped to the same socket [28]. This strategy ensures that, during the PCG iteration, a task is always executed on (any core of) the same socket that computed the corresponding task during the computation of the preconditioner.

For manycore processors, such as the Intel Xeon Phi, a critical aspect is how to bind the application threads to the hardware threads/cores of the systems, in order to attain a balanced distribution of the workload.

5.4. Optimizing energy consumption for linear algebra operations via the runtime

In general, optimizing energy consumption strongly requires the optimization of performance as a first step. This requisite particularly holds for DLA operations and especially the solution of sparse linear systems. For ILUPACK, though, it is possible to improve the energy efficiency of a runtime-based parallelization, with negligible impact on performance, by tuning the runtime itself to be aware of the ACPI C-states and P-states.

In a runtime-based parallelization, an idle runtime thread can rely on either polling or blocking policies upon encountering no task ready to be executed. For example, the prototype of SuperMatrix enforced a "busy-wait" for idle threads, till a new task is available. Unfortunately, this option impedes the transition of the corresponding core to a power-saving C-state because the thread is active (though doing useless work). From the positive point of view, this approach favors an immediate reaction of the thread to the creation of a new task. Alternatively, a power-aware version of the runtime can enforce an "idle-wait" (blocking) for idle threads.

The exploitation of the P-stages can also yield some potential energy savings if correctly adjusted from the runtime. Unfortunately, in some platforms, the use of the P-states is constrained by the operation of this mechanism at the socket level (i.e., it is not possible to change the P-state of a single core; instead, the change must be applied simultaneously for all cores of the socket). A second limiting factor for the P-states is the interplay between the frequency and memory bandwidth for some processor architectures, which implies that a reduction of the former negatively affects the latter as well. In consequence, exploiting this mechanism is far more delicate than doing so with the C-states. In particular, as energy consumption equals the product of time and power dissipation, the use of a lower P-state that reduces the frequency, and in principle, also the power draft, can result in a longer execution time that blurs the benefits from the point of view of energy consumption. In other words, reducing of power consumption is beneficial only if it does not increase the execution time that destroys the positive effects on energy consumption.

6. Energy-efficient techniques for multigrid methods on distributed memory platforms

Multigrid solvers belong to the most efficient numerical methods for solving symmetric positive definite linear systems. The computational complexity is $O(n)$ for sparse systems with n unknowns. To introduce the multigrid methodology, we revisit the definition of an iterative linear solver. Let $x^{(k)}$ be the approximation to the solution x resulting from the k-th iteration of the solver. The (unknown) error is denoted as $e^{(k)} = x - x^{(k)}$ and the residual vector is defined as $r^{(k)} = b - Ax^{(k)}$. Note that $e^{(k)}$ solves the error equation $Ae^{(k)} = r^{(k)}$ and $x^{(k)} = x \Leftrightarrow e^{(k)} = 0 \Leftrightarrow r^{(k)} = 0$. The abstract solution scheme stated in Algorithm 1 is known as iterative refinement. This scheme offers full flexibility for choosing a method to solve the error equation in step 4. Convergence of the approximation to the solution is guaranteed if the correction is sufficiently accurate since $r^{(k+1)} = r^{(k)} - Ac^{(k)}$. The idea of multigrid methods is to compute the error correction in step 4 of the iterative refinement scheme, not in the same space where the final solution is sought, but in a space of smaller dimension. The theoretical basis of multigrid methods is formulated in terms of "subspace correction" methods, see Refs. [29, 30]. Two prototypical variants of multigrid methods can be distinguished: the algebraic multigrid (AMG) variant is based on a purely algebraic problem formulation by means of a linear system of equations [31]. In contrast, the geometric multigrid (GMG) variant is based on the discretization of the underlying model equations on several grid refinement levels [32]. Both

multigrid variants use a hierarchy of grid levels, either derived from the algebraic problem formulation or resulting from the discretization of the model equations on different computational grids. The process of constructing the hierarchy from the finest level is called "coarsening". The transition between two grid levels is defined by means of grid transfer operators, namely restriction and prolongation. Smoothing methods are used in order to make the unknown error from a fine grid representable on a coarser grid, where the correction shall be computed. The smoother is applied to the approximation on the fine grid and has the implicit effect of removing high frequency contributions from the error. The multigrid cycle is stated in Algorithm 2.

Algorithm 1 Iterative refinement

1: Set initial solution $x^{(0)}$, tolerance $\delta > 0$, iteration counter $k = 0$.
2: Compute initial residual $r^{(0)} = b - Ax^{(0)}$.
3: **while** $\|r^{(k)}\| > \delta$ **do**
4: Solve error equation $Ae^{(k)} = r^{(k)}$ approximately by means of a correction $c^{(k)} \approx A^{-1}r^{(k)}$.
5: Update solution $x^{(k+1)} = x^{(k)} + c^{(k)}$.
6: Compute residual $r^{(k+1)} = b - Ax^{(k+1)}$.
7: $k \leftarrow k + 1$.
8: **end while**

Algorithm 2 Cycle (A_h, x_h, b_h, γ)

1: **if** $h = H$ **then**
2: $x_h \leftarrow A_h^{-1} b_h$ (coarse grid solution)
3: **else**
4: $x_h \leftarrow S_h(A_h, x_h, b_h)$ (presmoothing)
5: $r_h \leftarrow b_h - A_h x_h$ (residual computation)
6: $b_{2h} \leftarrow R_{2h}^h r_h$ (restriction)
7: $x_{2h} \leftarrow 0$
8: **for** $k = 1, 2, \ldots, \gamma$ **do**
9: Cycle$(A_{2h}, x_{2h}, b_{2h}, \gamma)$ (recursion)
10: **end for**
11: $c_h \leftarrow I_{2h}^h x_{2h}$ (prolongation)
12: $x_h \leftarrow x_h + c_h$ (correction)
13: $x_h \leftarrow S_h(A_h, x_h, b_h)$ (postsmoothing)
14: **end if**

It is a recursive algorithm with its recursion basis on the coarsest grid level, where the error equation is solved with high accuracy in step 2. On all finer levels, only smoothers, residual

computation, and grid transfer operators are used. The typical choices for the recursion parameter $\gamma = 1$ or $\gamma = 2$ lead to the V- or W-cycle, respectively, depicted in **Figure 15**. The solution is sought on the finest grid level where one expects the highest accuracy. The problem size in terms of number of variables is the largest on the finest grid level, and becomes subsequently smaller on the coarser levels. Standard smoothing methods include classical relaxation schemes like Jacobi, (symmetric) Gauss-Seidel or (symmetric) successive over-relaxation, and many other smoothing methods have been developed for specific fields of application. The smoother and grid transfer operator computations on fine grid levels usually only employ vector operations, sparse matrix-vector multiplications or element-wise operations in the grid. Only on the coarsest level, a direct or iterative method is used to solve the error correction equation with high accuracy. In the following paragraphs, we briefly introduce distributed memory platforms, the domain decomposition parallelization, which is often used on such platforms, and some implications on the numerical linear algebra routines.

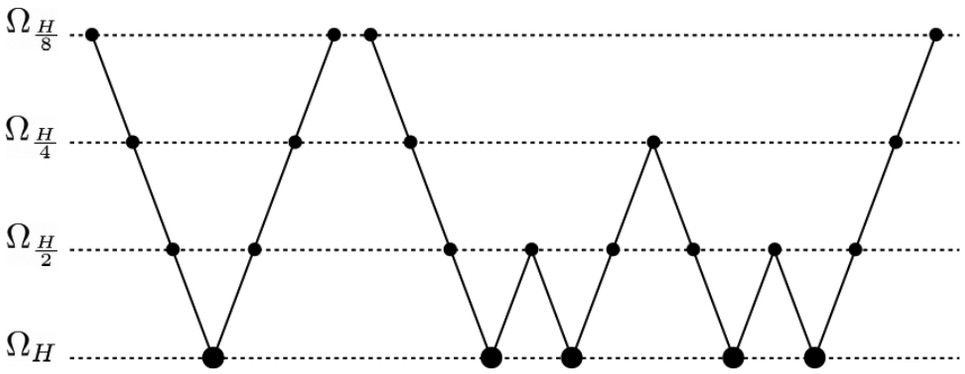

Figure 15. V-cycle (left) and W-cycle (right) across four grid levels. Small dots indicate the execution of the smoother, residual computation and grid transfer operators, while large dots indicate the solution of the coarse grid error equation.

6.1. Distributed memory platforms

Distributed memory means the separation of the available memory in distinct address spaces. This may be inherent to the computer platform, e.g., the computer units, or nodes, in interconnected clusters often have their own memory address space which is not accessible from other nodes. A distributed memory setup may also be induced by the coexistence of several host processes with individual private address spaces within the same computer. This is a fundamentally different situation compared to shared memory platforms, where all host processes or threads can access the same memory using a common address space. Note that there exist techniques which create a shared memory view of actually distributed memory platforms, but this is not discussed here. Parallelism on distributed memory platforms can be exploited by means of the "single instruction multiple data (SIMD)" or "multiple instruction multiple data (MIMD)" paradigm. It means that the problem data is split into pieces and distributed, while program replica (SIMD) or individual programs (MIMD) work on the data pieces in parallel. The processes which constitute the kind of parallel scientific application which we consider, may run on separate compute nodes in an interconnected cluster, possibly with several processes per node. Unless the application is "embarrassingly parallel", which denotes a situation without any dependency between the processes, a technique for making

data from one process available to other processes is needed. The widely used standard technique in HPC is explicit data transfer between processes using the message passing interface (MPI) [33].

6.2. Domain decomposition parallelization and its implications on linear algebra routines

The distributed memory parallelization technique used for many numerical simulations of physical processes is based on domain decomposition. Using a number p of processes, the computational grid Ω_h is divided into a corresponding number of subdomains $S_q \subset \Omega_h$, $\bigcup_{q=1}^{p} S_q = \Omega_h$. In practice, graph partitioner tools can be used to achieve a balanced decomposition. **Figure 16** shows an example of a domain decomposition of a flow channel with an obstacle into eight subdomains.

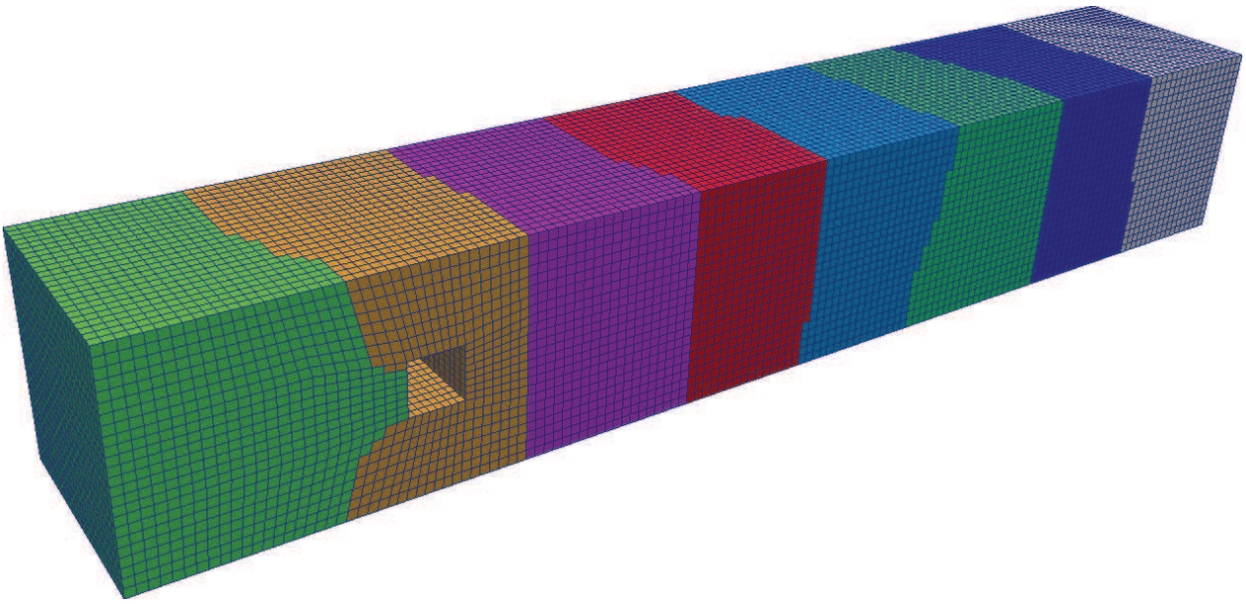

Figure 16. Domain decomposition of the computational grid of a flow channel with an obstacle into eight subdomains.

The domain decomposition implies a distribution of the discrete variables since these are associated to certain locations in the grid. Usually one uses a process-wise enumeration of the variables, so that the vectors and matrices of the discrete system are distributed in contiguous blocks among the processes. The usual technique to account for couplings of variables across subdomain boundaries is the addition of a "ghost layer" or "halo" of grid cells. The ghost layer replicates cells with coupling variables from the subdomains of other processes. The replicated variables are also called "ghost variables", and they are used in read-only mode for contributions of remote processes to the computations of the local process. Therefore, if the computation of a process relies on recent contributions from other processes, all affected ghost variables need to be updated by means of a data transfer. The communication pattern for a ghost update is determined by the neighborhood relation of the subdomains and the couplings of the variables. The local nature of the couplings, resulting from the discretization techniques described above, maintains the locality of the

communication pattern of each individual process, i.e., it comprises only the subdomain neighbor processes. To illustrate the effect of the domain decomposition parallelization on the actual computations using distributed matrices and vectors, we introduce some dedicated notation. The local part of a vector v that comprises the variables associated with the subdomain of the process q is denoted as v_q^{loc}, and the ghost variables of process q form the ghost vector part v_q^{ghost}. Accordingly, the distribution of a matrix A yields a matrix block on the diagonal A_q^{diag}, whose entries represent couplings among the local variables of process q, and an off-diagonal block $A_q^{\text{off-diag}}$, which comprises couplings of the local variables of process q with variables from other processes, i.e., with ghost variables. **Figure 17** shows a distributed matrix with diagonal and off-diagonal blocks, and a distributed vector with local parts. The ghost vector parts are not shown in this graphic.

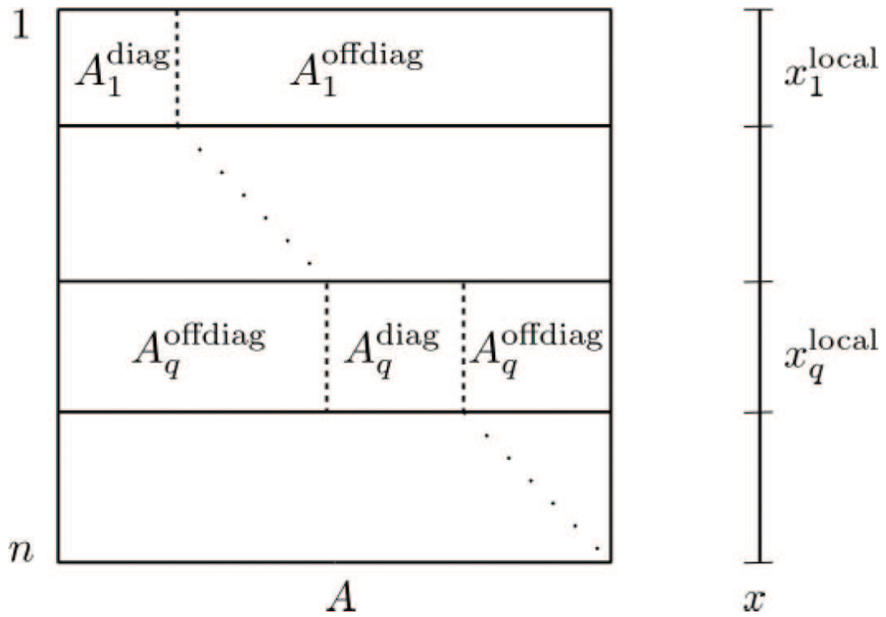

Figure 17. Distributed matrix with diagonal and off-diagonal blocks, and distributed vector local parts.

6.3. Parallel setup of some linear algebra routines

We can now formulate the distributed memory parallel version of some of the most important linear algebra routines, which are often used in numerical solvers. The scaling and addition of vectors can be done independently by each process for its local vector part, without using the ghost part:

$$\forall\ 1 \le q \le p : (\alpha v + w)_q^{\text{loc}} = \alpha v_q^{\text{loc}} + w_q^{\text{loc}}, \tag{10}$$

where α is a scalar, and v and w are distributed vectors.

The matrix-vector multiplication uses the ghost vector part and therefore needs a ghost update:

$$\forall\ 1 \le q \le p : (Av)_q^{\text{loc}} = A_q^{\text{diag}} v_q^{\text{loc}} + A_q^{\text{off-diag}} v_q^{\text{ghost}}. \tag{11}$$

The ghost update needs to be completed before the off-diagonal matrix part is multiplied with the ghost vector part. However, the contribution from the diagonal matrix block multiplied with the local vector part is independent from other processes. This offers the opportunity to use a nonblocking communication mechanism for overlapping communication and computation, according to the following scheme:

1. Start nonblocking communication for the ghost vector update.
2. Compute $w_q^{\text{loc}} \leftarrow A_q^{\text{diag}} v_q^{\text{loc}}$.
3. Wait for completion of ghost vector update.
4. Compute $w_q^{\text{loc}} \leftarrow w_q^{\text{loc}} + A_q^{\text{off-diag}} v_q^{\text{ghost}}$.

In this scheme, the computation for step 2 potentially overlaps with the communication for the ghost update. This is advantageous since computation and communication may happen concurrently, thus accelerating the routine. However, it depends on the compute node and network hardware and system software to what degree the overlapping actually happens. If the node architecture is able to delegate the communication to the system, and meanwhile continue with the computation for the local contribution, a benefit for the overall performance can be expected. Ideally, the data transfer would happen entirely in the background, thus being completely hidden behind the computation overlap, without any communication overhead degrading the parallel efficiency. Of course, this can only be realized if the ghost update is completed before the computation for the local contribution finishes, so that there is actually no waiting necessary in step 3 of the scheme.

Without involving the ghost vector part, but nevertheless different from scaling and vector addition, and also from matrix-vector multiplication is the computation of scalar products and norms:

$$v \cdot w = \sum_{q=1}^{p} v_q^{\text{loc}} \cdot w_q^{\text{loc}}, \quad \|v\|_k = \left(\sum_{q=1}^{p} \|v_q^{\text{loc}}\|_k^k \right)^{1/k} \text{ for } 1 \le k < \infty, \tag{12}$$

$$\|v\|_\infty = \max_{1 \le q \le p} \|v_q^{\text{loc}}\|_\infty \tag{13}$$

The difference from scaling, vector addition, and matrix-vector multiplication is that the result has a global nature. Scalar product and norms require a global reduction operation such as global sum or global maximum. All processes must contribute to the result, and this result must be available on all processes. Therefore, scalar product and norm computations imply a global synchronization of all processes. Communication libraries such as MPI often provide routines with an optimized routing strategy, e.g., using tree-based routing algorithms of logarithmic complexity with respect to the number of processes, to implement global reduction operations.

6.4. Energy-efficient techniques for multigrid methods

Opportunities for making multigrid solvers more energy-efficient can be sought in all building blocks of the method including the smoother, grid transfer operators, and coarse grid solver. However, the individual parts must usually be addressed by individual measures for optimization due to their different roles in the multigrid algorithm.

As explained above, the purpose of the smoother is to remove high frequency contributions from the unknown error. To achieve this smoothing effect, the smoother does usually not need to yield an accurate approximation of the solution. Moreover, it does not always need to converge at all, as long as it has the desired smoothing properties. This gives space for the choice and optimization of smoothers. We depict one example in the following:

Traditional smoother choices include the classical Jacobi or Gauss-Seidel iteration and their damped variants. For a system matrix A with nonzero diagonal elements, the Jacobi iteration reads in component-wise form

$$x_i^{k+1} = x^k + \frac{1}{a_{ii}}\left[b_i - \sum_{j \neq i} a_{ij} x_j^k\right] \quad (i = 1, ..., n), \tag{14}$$

where k is the iteration index. The new iterate x^{k+1} relies on the previous iteration x^k, thus imposing a sequential order for the computation of the iterates. Moreover, in parallel setups, computing iterates usually involve contributions from other processes. This requires a synchronization of the processes after each iteration to make sure all needed values are updated from the last iteration.

For smoothers, it might, however, be acceptable to relax this strictly synchronized scheme by allowing to use also older or newer iterations in the computation. This leads to asynchronous iterations, which can benefit from massively parallel architectures like manycore devices or graphics processing units (GPUs). On the mathematical level, this is achieved by introducing a shift function, s, and an update function, u, in the algorithm:

$$x_i^{k+1} = \begin{cases} x_i^k + \frac{1}{a_{ii}}\left[b_i - \sum_{j \neq i} a_{ij} x_j^{k-s(j)}\right] & \text{if } i = u(k), \\ x_i^k & \text{if } i \neq u(k). \end{cases} \tag{15}$$

This abstract scheme can be adapted to fit the hardware at hand, e.g., by aggregating the components into blocks and mapping them to the cores of a multicore processor or to the thread blocks of a CUDA GPU. Thus, the adaption of the classic, synchronized relaxation scheme allows to efficiently exploit the parallelism of modern hardware, and in particular, it offers an opportunity to benefit from the superior FLOPS per Watt characteristics of GPUs.

In contrast, the residual and the grid transfer usually need to be computed accurately, because otherwise the overall convergence of the multigrid solver cannot be guaranteed. Nevertheless, performing residual and grid transfer computations on coprocessors such as manycore accelerators or GPUs might still prove beneficial, but care must be taken to ensure consistency in

distributed systems. The coarse grid error correction solver is also expected to provide an accurate result. It nonetheless offers space for optimization, both in the choice of the method and its implementation. One usually employs Krylov subspace methods such as CG or GMRES methods, or direct methods such as LU or QR decompositions. The coarse grid solver can itself be subject to energy and performance optimization, and the overall multigrid solver would then inherit the benefits.

Another direction for energy and performance optimization, which is particularly relevant for distributed memory platforms, affects the parallel setup of the grid levels in the multigrid hierarchy. The problem sizes in the hierarchy often differ in several orders of magnitude from the largest problem size on the finest level to the smallest problem size on the coarsest level. A simple parallelization, where all grid levels are distributed to all available processors, may turn out to scale poorly for a large number of processors. This is due to the communication overhead becoming significant and diminishing the efficiency of the computations on coarse levels with small problem sizes. Instead, parallel setups, where coarser levels use only a subset of the available processors, can be beneficial. Balanced setups can maintain the overall performance of the parallelization, while reducing the communication, such that the overall efficiency is conserved. A fraction of the available processors can be temporarily deactivated, while the multigrid algorithm operates on coarse levels, and activated again to use the full computing power on finer levels. It is crucial to keep communication patterns local between neighboring subdomains, both within each grid level as well as between grid levels for the grid transfer. Well-configured parallel setups in the multigrid hierarchy can yield substantial energy savings by deactivating processes and reducing communication, while maintaining the overall performance.

Author details

Martin Wlotzka[1]*, Vincent Heuveline[2], Manuel F. Dolz[3], M. Reza Heidari[4], Thomas Ludwig[4], A. Cristiano I. Malossi[5] and Enrique S. Quintana-Orti[6]

*Address all correspondence to: martin.wlotzka@uni-heidelberg.de

1 Interdisciplinary Center for Scientific Computing (IWR), Heidelberg University, Germany

2 Computing Center, Heidelberg University, Germany

3 Computer Science Department, University Carlos III of Madrid, Spain

4 Department of Informatics, Universität Hamburg, Germany

5 Cognitive Computing and Computational Sciences Department, IBM Research, Zurich, Switzerland

6 Departamento de Ingenieria y Ciencia de Computadores, Universidad Jaime I, Castellon, Spain

References

[1] N.S. Trudinger D. Gilbarg. *Elliptic Partial Differential Equations of Second Order. Classics in Mathematics*. Springer Berlin Heidelberg, 2001.

[2] L.C. Evans. *Partial Differential Equations*, volume 19 of *Graduate Studies in Mathematics*. American Mathematical Society, 2 edition, 2010.

[3] R.J. LeVeque. *Finite Difference Methods for Ordinary and Partial Differential Equations*. Society for Industrial and Applied Mathematics, Classics in Applied Mathematics edition, 2007.

[4] J.L. Guermond A. Ern. *Theory and Practice of Finite Elements*. Springer New York, 2010.

[5] R. Herbin R. Eymard, T. Gallouete. *Finite Volume Methods. In: Handbook of Numerical Analysis, Vol. 7*. Elsevier. 2000.

[6] G. Wanner E. Hairer. *Solving Ordinary Differential Equations II: Stiff and Differential Algebraic Problems*. Springer Berlin Heidelberg, 2010.

[7] G. Wanner E. Hairer, S.P. Norsett. *Solving Ordinary Differential Equations I: Nonstiff Problems*. Springer Berlin Heidelberg, 2008.

[8] C.F. van Loan G.H. Golub. *Matrix Computations*. Johns Hopkins University Press Baltimore London, 4 edition, 2013.

[9] Y. Saad. *Iterative Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics, 2 edition, 2003.

[10] Towards supercomputers: Eu project Exa2Green improves energy efficiency in high performance computing. *ICT-Energy Lett.*, (10), 2015.

[11] S. Catalan M.F. Dolz G. Fabregat R. Mayo E.S. Quintana-Orti S. Barrachina M. Barreda. An integrated framework for power-performance analysis of parallel scientific workloads. In *Proceedings of the 3rd International Conference on Smart Grids, Green Communications and IT Energy-Aware Technologies (ENERGY)*, pages 114–119, 2013.

[12] A.D. Malony S.S. Shende. The tau parallel performance system. *Int. J. High Perform. Comput. Appl.*, 20, 2006.

[13] M. Jurenz M. Lieber H. Brunst H. Mix W.E. Nagel M.S. Müller, A. Knüpfer. Developing scalable applications with vampir, vampirserver and vampirtrace. In *Proceedings of the Parallel Computing: Architectures, Algorithms and Applications Conference (PARCO)*, 2007.

[14] *Lapack project homepage*.

[15] R. Mayo E.S. Quintana-Orti R. Reyes M. Barreda, M.F. Dolz. Binding performance and power of dense linear algebra operations. In *Proceedings of the IEEE 10th International Symposium on Parallel and Distributed Processing with Applications*, pages 63–70, 2012.

[16] *The Top500 List*, June 2014.

[17] *The Green500 List*, June 2014.

[18] B.C. Catanzaro J.J. Gebis O. Husbands. K. Kreutzer D.A. Patterson W.L. Plishker J. Shalf S.W. Williams K.A. Yelick K. Asanovic R. Bodik. *The Landscape of Parallel Computing Research: A View from Berkeley*. Technical report, Electrical Engineering and Computer Sciences, University of California, Berkeley, Tech. Rep. UCB/EECS-2006-183, 2006.

[19] C. Bekas A. Curioni E.S. Quintana-Orti A.C.I. Malossi, Y. Ineichen. Performance and energy-aware characterization of the sparse matrix-vector multiplication on multithreaded architectures. In *Proceedings of the 43rd International Conference on Parallel Processing Workshops*, 2014.

[20] S. Lloyd. Least squares quantization in pcm. *IEEE Trans. Inf. Theory*, 28(2):129–137, 1982.

[21] A. Curioni C. Bekas. A new energy aware performance metric. *Comput. Sci. Res. Dev.*, 25 (3–4):187–195, 2010.

[22] P. Klavik, A.C.I. Malossi C. Bekas A. Curioni. Changing computing paradigms towards power efficiency. *Philos. Trans. Math. Phys. Eng. Sci.*, 372(2018), 2014.

[23] *The University of Florida Sparse Matrix Collection*, April 2014.

[24] *Ilupack*, July 2015.

[25] *Acpi*, July 2015.

[26] *Ompss*, July 2015.

[27] *The Openmp Api Specification for Parallel Programming*, November 2015.

[28] M. Barreda M. Bollhöfer E.S. Quintana-Orti J.I. Aliaga, R. Badia. Leveraging task-parallelism with ompss in ILUPACK's preconditioned CG method. In *Proceedings of the 26th International Symposium on Computer Architecture and High Performance Computing*, pages 262–269, 2014.

[29] J. Xu. Iterative methods by space decomposition and subspace correction. *SIAM Rev.*, 34 (4):581–613, 1992.

[30] H. Yserantant. Old and new convergence proofs for multigrid methods. *Acta Numer.*, 2:285–326, 1993.

[31] K. Stueben. A review of algebraic multigrid. *J. Comput. Appl. Math.*, 128:281–309, 2001.

[32] C.W. Oosterlee P. Wesseling. Geometric multigrid with applications to computational fluid dynamics. *J. Comput. Appl. Math.*, 128:311–334, 2001.

[33] *Message Passing Interface Forum. MPI: A Message-Passing Interface Standard*, Version 3.0. University of Tennessee, Knoxville, Tennessee, 2012.