Energy-aware Strategies for Task-parallel Sparse Linear System Solvers

José I. Aliaga, María Barreda, Asunción Castaño

aliaga,mvaya,castano@uji.es

Depto. de Ingeniería y Ciencia de Computadores,

Universidad Jaume I (UJI), 12071–Castellón, Spain

Abstract

We present some energy-aware strategies to improve the energy efficiency of a task-parallel preconditioned Conjugate Gradient (PCG) iterative solver on a Haswell-EP Intel Xeon. These techniques leverage the power-saving states of the processor, promoting the hardware into a more energy-efficient C-state and modifying the CPU frequency (P-states of the processors) of some operations of the PCG. We demonstrate that the application of these strategies during the main operations of the iterative solver can reduce its energy consumption considerably.

1 Introduction

Large sparse systems of linear equations are omnipresent problems in diverse scientific and engineering applications and big-data analytics. The interest of these applications and the fact that the solution of the linear system is usually a significant time-consuming stage has motivated, over the past decades, the development of highly tuned algorithms and libraries to efficiently tackle sparse instances of these linear algebra problems in general-purpose processors, following the evolution of computer architectures.

High Performance Computing architectures enable the solution of complex applications by aggregating a number of multicore processors. As a consequence, developers face the challenge of implementing parallel algorithms that efficiently exploit the concurrency of the hardware. Furthermore, the advances in the number of transistors that can be integrated in a circuit have not enjoyed a proportional reduction of the power dissipated by the CMOS technology, turning the power wall into a crucial challenge that the High Performance Computing community needs to address [1, 2, 3]. Unfortunately, in an era where power has become the key factor that constrains both the design and performance of current computer architectures, few software developers take it into account in their implementations. Therefore, much remains to be done in terms of energy efficiency to render Exascale systems feasible by 2020 [3, 4].

ILUPACK (Incomplete LU decomposition PACKage) [1] offers an assorted variety of Krylov subspace-based methods, enhanced with a sophisticated ILU-type preconditioner, for the iterative solution of sparse linear systems. The computational cost of computing and applying ILUPACK's preconditioner has sparked several recent efforts to develop parallel versions of this solver, for multicore processors, graphics accelerators, and clusters of computer nodes [5, 6, 7].

Task-parallel versions of ILUPACK have also been used as a case study to explore the energy consumption and optimization of iterative solvers. Concretely, in [8], the impact of P-States and C-States on ILUPACK is analyzed on two distinct platforms based on multicore technology from AMD and Intel. On both architectures, the *race-to-halt/race-to-idle* strategy, on which the C-states are exploited, was the key to develop energy-aware implementations of ILUPACK. Moreover, the authors conclude that the use of static Voltage Frequency Scaling (VFS) based approach is only useful in one of these platforms. More recently, in [9], a preliminary algorithmic-based energy-saving technique is introduced to improve the energy efficiency, in which the P-states of some preconditioned Conjugate Gradient (PCG) operations are dynamically changed. In this paper, we extend that work, exploring different new approaches (using both P-states and C-states) to save energy in the task-parallel version of ILUPACK's PCG on an Intel Xeon Haswell-EP processor. To yield an *energy-efficient execution* we have analyzed the following strategies:

- We explore the benefits of *race-to-halt/race-to-idle* strategy, which reduces the energy-consumption of ILUPACK transforming the busy-wait behaviour of idle threads to idle-wait.

- In addition, we leverage the iterative nature of the method to progressively adjust the frequency of the processor cores in order to reduce idle periods and harvest energy, implementing a *dynamic VFS-based approach (DVFS)*. In rough detail, this strategy detects the threads which produce more idle periods, and adjusts the best P-state for the execution of each task in order to reduce the waiting time of these threads.

- We also implement a *memory-bound aware* methodology that adjusts the P-state of all threads taking into account the energy consumption instead of the execution time. This technique can use several alternatives to achieve the objective of minimizing energy consumption.

All these techniques have been combined to analyze their impact on the implementation of task-parallel versions of ILUPACK, showing many energy gains on the Haswell-EP processor.

The rest of the paper is structured as follows. In section 2 we introduce the multilevel preconditioned iterative solver in ILUPACK, and its task-parallel variant. In section 3 we explain the different energy-aware strategies implemented to improve the energy efficiency in ILUPACK. We evaluate the impact

[1] http://ilupack.tu-bs.de

$A \to M$	O0. PrCo
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. PrAp
$\beta_{k+1} := r_{k+1}^T z_{k+1}$	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 1: Algorithmic formulation of the PCG method. Here, $\tau_{\max}$ is an upper bound on the relative residual for the computed approximation to the solution.

on the application of these strategies in section 4. Finally, we close the paper with a few concluding remarks in section 5.

2 Overview of ILUPACK

2.1 Sequential ILUPACK

Consider the linear system $Ax = b$, where $A \in \mathbb{R}^{n \times n}$ is a sparse coefficient matrix, $b \in \mathbb{R}^n$ is the right-hand side vector, and $x \in \mathbb{R}^n$ is the sought-after solution. The solution of these kind of systems can be performed using ILUPACK, a software library, written in C and Fortran, for the iterative solution of large sparse linear systems. For s.p.d linear systems, ILUPACK's implementation of the PCG method integrates an "inverse-based approach" into the ILU factorization of matrix A, in order to obtain an efficient preconditioner ($A \approx LU = LL^T = M$) [10]. This method applies dropping combined with pivoting to bound the norm of the inverse triangular factor L, obtaining a numerical multilevel hierarchy of partial inverse-based approximations [11, 12].

The algorithmic description of the PCG method implemented in ILUPACK is presented in Figure 1. The first step of the solver is the computation of the preconditioner M (PrCo in O0). The following iteration comprises a matrix-vector product (SpMV in O1), the preconditioner application (PrAp in O5), and various vector operations (dot products, axpy-like updates, 2-norm; in O2–O4 and O6–O8). The computation and application of the preconditioner centralize most of the computational cost of the solver. For this reason, in the remainder of this section, we mainly focus on these operations.

Computation of the preconditioner. ILUPACK relies on the commonly named *inverse-based approach* to compute the preconditioner. This approach enhances the robustness of classical Incomplete LDL^T factorizations by restricting the growth of the entries in the inverses of the triangular factors. To verify

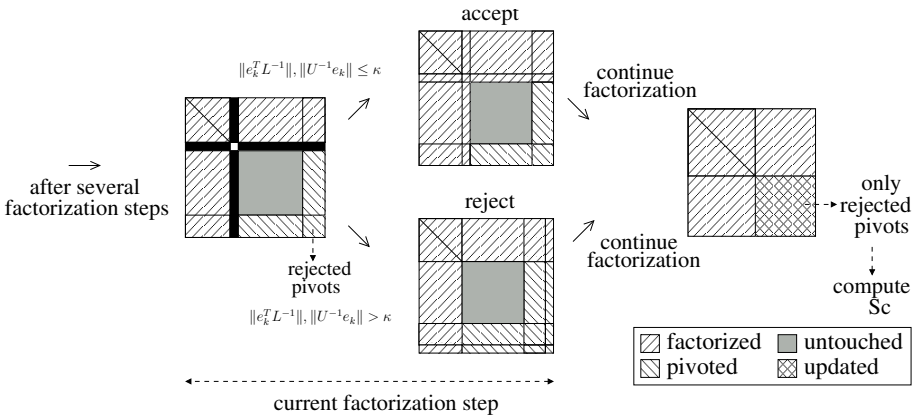

Figure 2: A step of the Crout variant of the preconditioner computation in ILUPACK.

this, consider the factorization:

$$A = \tilde{L}\tilde{D}\tilde{L}^T + R \;, \tag{1}$$

where $\tilde{L}$ is unit lower triangular matrix, $\tilde{D}$ is diagonal, and R is the error matrix which accumulates those entries dropped during the factorization. The preconditioned matrix is obtained by applying the preconditioner $M = \tilde{L}\tilde{D}\tilde{L}^T$ on the original matrix:

$$\tilde{L}^{-1} A \tilde{L}^{-T} = \tilde{D} + \tilde{L}^{-1} R \tilde{L}^{-T} \;. \tag{2}$$

The fact that $\tilde{L}^{-1}$ exhibits large norms may impact the convergence rate of the preconditioned iterative solver, because the size of the entries in R will be significantly amplified in the preconditioner application. Thus, to solve this issue, ILUPACK accommodates pivoting during the factorization to bound the norm of the inverse triangular factors, creating a multi-level hierarchy of partial inverse-based approximations (see Figure 2).

When the multi-level method is employed over multiple levels, a cascade of factors are usually acquired. Besides, the computed multi-level factorization is adapted to the structure of the subjacent system. Consequently, in this case, the multi-level preconditioner can be formulated recursively, at a given level l, as

$$M_l \approx \tilde{D}^{-1}\tilde{P}P \quad , \quad \begin{bmatrix} \tilde{L}_B & 0 \\ \tilde{L}_F & I \end{bmatrix} \begin{bmatrix} \tilde{D}_B & 0 \\ 0 & M_{l+1} \end{bmatrix} \begin{bmatrix} \tilde{L}_B^T & \tilde{L}_F^T \\ 0 & I \end{bmatrix} P^T \tilde{P}^T \tilde{D}^{-1} \;, \tag{3}$$

where $\tilde{L}_B$, $\tilde{L}_F$ and $\tilde{D}_B$ are blocks of the factors of the multi-level $\tilde{L}\tilde{D}\tilde{L}^T$ preconditioner (with $\tilde{L}_B$ unit lower triangular and $\tilde{D}_B$ diagonal); and M_{l+1} is the preconditioner computed at level $l+1$.

Application of the preconditioner. Tha applicaton of the preconditioner in level l (i.e., computing $z := M_l^{-1} r$) requires solving this system of linear

equations:

$$\begin{bmatrix} \tilde{L}_B & 0 \\ \tilde{L}_E & I \end{bmatrix} \begin{bmatrix} \tilde{D}_B & 0 \\ 0 & M_{l+1} \end{bmatrix} \begin{bmatrix} \tilde{L}_B^T & \tilde{L}_F^T \\ 0 & I \end{bmatrix} P^T \tilde{P}^T \tilde{D}^{-1} z = P^T \tilde{P}^T \tilde{D}^{-1} r \ . \tag{4}$$

After applying several transformations to the initial system ($r' := Dr$ and $\hat{r} := P^T \tilde{P}^T r'$), we obtain this new system [13]:

$$\begin{bmatrix} \tilde{L}_B & 0 \\ \tilde{L}_E & I \end{bmatrix} \begin{bmatrix} \tilde{D}_B & 0 \\ 0 & M_{l+1} \end{bmatrix} \begin{bmatrix} \tilde{U}_B & \tilde{U}_F \\ 0 & I \end{bmatrix} w = \hat{r},$$

which is solved for $w(= P^T \tilde{P}^T \tilde{D}^{-1} z)$ in three steps:

$$\begin{bmatrix} \tilde{L}_B & 0 \\ \tilde{L}_E & I \end{bmatrix} \begin{bmatrix} y_B \\ y_C \end{bmatrix} = \begin{bmatrix} \hat{r}_B \\ \hat{r}_C \end{bmatrix} \Rightarrow \tilde{L}_B y_B = \hat{r}_B \ ; \ y_C := \hat{r}_C - \tilde{L}_E y_B$$

$$\begin{bmatrix} \tilde{D}_B & 0 \\ 0 & M_{l+1} \end{bmatrix} \begin{bmatrix} x_B \\ x_C \end{bmatrix} = \begin{bmatrix} y_B \\ y_C \end{bmatrix} \Rightarrow x_B := D_B^{-1} y_B \ ; \ x_C := M_{l+1}^{-1} y_C$$

$$\begin{bmatrix} \tilde{U}_B & \tilde{U}_F \\ 0 & I \end{bmatrix} \begin{bmatrix} w_B \\ w_C \end{bmatrix} = \begin{bmatrix} x_B \\ x_C \end{bmatrix} \Rightarrow w_C := x_C \ ; \ \tilde{U}_B w_B = x_B - \tilde{U}_F w_C$$

Overall, at each level of the iterative solver, ILUPACK operates two sparse matrix-vector multiplications and solves two linear systems of the form $\tilde{L}\tilde{D}\tilde{L}^T x = b$. Additionally, it performs three other types of operations: diagonal scaling, vector permutation, and vector updates of the form $x := a - b$.

2.2 Task-parallel ILUPACK

Nested dissection. The parallel version of ILUPACK can be exposed by means of nested dissection orderings, which reveal parallelism exploiting the connection between sparse matrices and adjacency graphs. In particular, nested dissection algorithm partitions the adjacency graph $G(A)$ associated to the approximate factorization of A into a hierarchy of vertex separators and independent subgraphs [14]. For example, in Figure 3, $G(A)$ is partitioned after two levels of recursion into four independent subraphs, $G_{(3,1)}$, $G_{(3,2)}$, $G_{(3,3)}$, and $G_{(3,4)}$, first by separator $S_{(1,1)}$ and then by separators $S_{(2,1)}$ and $S_{(2,2)}$. This hierarchy is constructed, using METIS software, to minimize the size of the vertex separators and balance the size of the independent subgraphs [15]. Therefore, relabeling the nodes of $G(A)$ according to the levels in the hierarchy produces a reordered matrix, $A \leftarrow P^T AP$, with a structure sensitive to efficient parallelization. Concretely, the leading diagonal blocks of $P^T AP$ associated with the independent subgraphs can be first computed independently; after that, $S_{(2,1)}$ and $S_{(2,2)}$ can be processed in parallel, and finally, the separator $S_{(1,1)}$ is calculated. This type of concurrency can be expressed as a binary task-dependency graph (TDG) (see Figure 3), where the nodes represent simultaneous tasks and the edges dependencies among them.

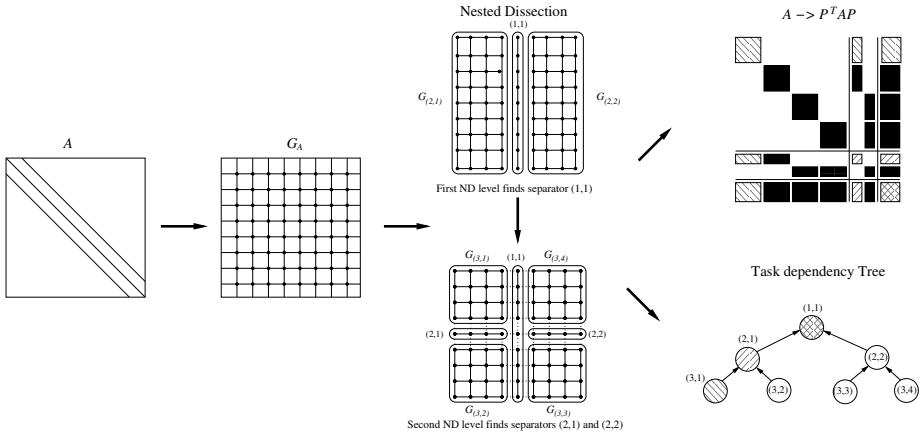

Figure 3: Nested dissection reordering. In this example $G(A)$ is partitioned into four independent subgraphs.

Computation of the preconditioner. In order to design a task-parallel version of ILUPACK, we decouple the computation of the preconditioner into tasks, identifying the dependencies among them, and mapping the tasks to the execution nodes. For that purpose, the task-parallel version exploits the connection between sparse matrices and adjacency graphs [16], extracting parallelism via nested dissection, as explained before. Consider, for example, a graph-based symmetric reordering, defined by a permutation $\bar{P} \in \mathbb{R}^{n \times n}$ [10], such that

$$\bar{P}^T A \bar{P} = \left[\begin{array}{cc|c} A_{00} & 0 & A_{02} \\ 0 & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right] . \tag{5}$$

Computing partial Incomplete Cholesky (IC) factorizations of the two leading blocks, A_{00} and A_{11}, generates the subsequent partial approximation of $\bar{P}^T A \bar{P}$

$$\left[\begin{array}{cc|c} L_{00} & 0 & 0 \\ 0 & L_{11} & 0 \\ \hline L_{20} & L_{21} & \mathrm{I} \end{array}\right] \left[\begin{array}{cc|c} D_{00} & 0 & 0 \\ 0 & D_{11} & 0 \\ \hline 0 & 0 & S_{22} \end{array}\right] \left[\begin{array}{cc|c} L_{00}^T & 0 & L_{20}^T \\ 0 & L_{11}^T & L_{12}^T \\ \hline 0 & 0 & \mathrm{I} \end{array}\right] + E_{01} ,$$

where

$$S_{22} = A_{22} - (L_{20} D_{00} L_{20}^T) - (L_{21} D_{11} L_{21}^T) + E_2 \tag{6}$$

is the approximate Schur complement. By recursively progressing with S_{22} in the same way, the IC factorization of $\bar{P}^T A \bar{P}$ is eventually finished.

The block structure in (5) exposes a coarse-grain parallelism during these computations. Concretely, the permuted matrix can be decomposed into two submatrices, in order to concurrently obtain the IC factorizations of the leading

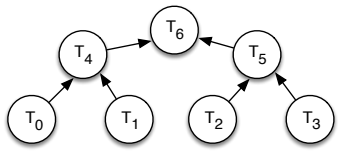

Figure 4: TDG of the diagonal blocks. Task T_j is associated with block A_{jj}.

block of both submatrices:

$$A_{22} = A_{22}^0 + A_{22}^1 \quad , \quad \begin{cases} \left[\begin{array}{c|c} A_{00} & A_{02} \\ \hline A_{20} & A_{22}^0 \end{array}\right] = \left[\begin{array}{c|c} L_{00} & 0 \\ \hline L_{20} & \mathrm{I} \end{array}\right] \left[\begin{array}{c|c} D_{00} & 0 \\ \hline 0 & S_{22}^0 \end{array}\right] \left[\begin{array}{c|c} L_{00}^T & L_{20}^T \\ \hline 0 & \mathrm{I} \end{array}\right] + E_0, \\ \left[\begin{array}{c|c} A_{11} & A_{12} \\ \hline A_{21} & A_{22}^1 \end{array}\right] = \left[\begin{array}{c|c} L_{11} & 0 \\ \hline L_{21} & \mathrm{I} \end{array}\right] \left[\begin{array}{c|c} D_{11} & 0 \\ \hline 0 & S_{22}^1 \end{array}\right] \left[\begin{array}{c|c} L_{11}^T & L_{21}^T \\ \hline 0 & \mathrm{I} \end{array}\right] + E_1. \end{cases} \tag{7}$$

Then, we can also compute in parallel the Schur complements related to both partial approximations

$$S_{22}^0 = A_{22}^0 - \left(L_{20} D_{00} L_{20}^T\right) + E_2^0 \quad , \quad S_{22}^1 = A_{22}^1 - \left(L_{21} D_{11} L_{21}^T\right) + E_2^1 \ .$$

However, the construction of (6) requires a synchronization before calculating the addition of these two blocks.

$$E_2 \approx E_2^0 + E_2^1 \quad , \quad S_{22} \approx S_{22}^0 + S_{22}^1. \tag{8}$$

To expose increasing amounts of task parallelism, we can identify a larger number of independent diagonal blocks, by applying permutations analogous to $\bar{P}$ on the two leading blocks. For example, we can obtain a block structure similar to (5), from which four submatrices can be disassembled, by reordering and renaming the blocks properly:

$$\left[\begin{array}{cccc|cc|c} A_{00} & 0 & 0 & 0 & A_{04} & 0 & A_{06} \\ 0 & A_{11} & 0 & 0 & A_{14} & 0 & A_{16} \\ 0 & 0 & A_{22} & 0 & 0 & A_{25} & A_{26} \\ 0 & 0 & 0 & A_{33} & 0 & A_{35} & A_{36} \\ \hline A_{40} & A_{41} & 0 & 0 & A_{44} & 0 & A_{46} \\ 0 & 0 & A_{52} & A_{53} & 0 & A_{55} & A_{56} \\ \hline A_{60} & A_{61} & A_{62} & A_{63} & A_{64} & A_{65} & A_{66} \end{array}\right] \rightarrow \begin{array}{l} \bar{A}_{00} = \left[\begin{array}{c|cc} A_{00} & A_{04} & A_{06} \\ \hline A_{40} & A_{44}^0 & A_{46}^0 \\ A_{60} & A_{64}^0 & A_{66}^0 \end{array}\right], \quad \bar{A}_{11} = \left[\begin{array}{c|cc} A_{11} & A_{14} & A_{16} \\ \hline A_{41} & A_{44}^1 & A_{46}^1 \\ A_{61} & A_{64}^1 & A_{66}^1 \end{array}\right] \\ \\ \bar{A}_{22} = \left[\begin{array}{c|cc} A_{22} & A_{25} & A_{26} \\ \hline A_{52} & A_{55}^2 & A_{56}^2 \\ A_{62} & A_{65}^2 & A_{66}^2 \end{array}\right], \quad \bar{A}_{33} = \left[\begin{array}{c|cc} A_{33} & A_{35} & A_{36} \\ \hline A_{53} & A_{55}^3 & A_{56}^3 \\ A_{63} & A_{65}^3 & A_{66}^3 \end{array}\right] \end{array} \tag{9}$$

Figure 4 illustrates the TDG for the factorization of the diagonal blocks in (9). The nodes that are located in the same level of the graph can be factorized in parallel, whereas the edges of the TDG define the dependencies between the diagonal blocks of the matrix (tasks), i.e., the order in which these blocks have to be processed.

The task parallel version of ILUPACK partitions the matrix A into a number of decoupled submatrices, and then performs a partial IC factorization during the computation of (7). The main change respect the sequential procedure is

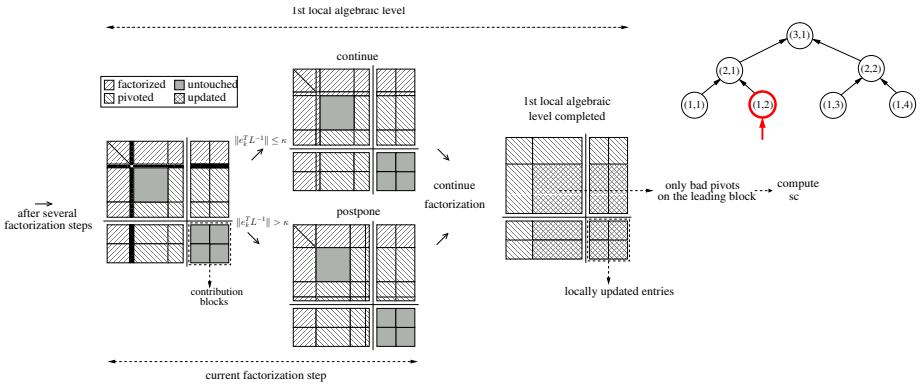

Figure 5: A step of the Crout variant of the parallel preconditioner computations.

that the computation here is restricted to the leading block, so that, the rejected pivots are moved to the bottom-right corner of the leading block; see Figure 5. Another modification is that preconditioning introduces structural levels in the recursive definition of IC preconditioners. Thus, the parallel IC preconditioner contains numerical and structural levels in its recursive definition. Therefore, distinct TDGs involve different recursion steps, obtainig distinct preconditioners. However, they present close numerical properties to that obtained with the sequential ILUPACK [17].

Application of the preconditioner. As we stated in the previous subsection, this operation in ILUPACK lacks the solution of two triangular systems (lower and upper triangular factors). The TDG has to be traversed two times per solve $z_{k+1} := M^1 r_{k+1}$ at each iteration of the PCG. Hence, although the TDG associated with the first triangular system (LwTrSv) presents the same structure and dependencies as that related to the preconditioner computation, in the latter triangular solve (UpTrSv), the dependencies are reversed (from the root to the leaves). For that reason, the amount of parallelism expands/reduces as we progress towards/away from the leaves.

Other kernels in the PCG iteration. The vectors involved in the PCG are partitioned conformally to matrix A, see (9). Accordingly, the SpMV and axpy-like kernels only operate with the data related to the leaves of the TDG. For example, for a matrix partitioned as in (9), the SpMV is decomposed into four matrix-vector products, so that, the processing of each one of these four leaves is totally independent from the others. The procedure in the axpy-like operations is the same, thus, this operation can be fully computed in parallel. On the other hand, the dot and the 2-norm require a reduction, after the computation in the leaves to obtain the result, which implies a synchronization point.

Mapping tasks to cores. In this paper, we rely on a data-flow version of ILUPACK using an ad-hoc runtime based on OpenMP [18], developed in [17], in which, each operation appearing in ILUPACK's PCG is decomposed into a number of tasks which should be mapped on the cores to be executed. The mapping is dynamically defined during the PrCo, through the use of a shared task queue, on which the active tasks are included and the cores access to demand new work. Initially, only the leaf nodes of the TGD are stored in the queue, whereas the intermediate nodes are included when their children have been completed. Later, the same mapping is used during the completion of the operations included in each PCG iteration. To do this, task queues are created in each core to manage the tasks which were mapped in the core during the PrCo. For the LwTrSv, the management of the tasks is the same as in the PrCo, but, for UpTrSv, the TDG is traversed in a reverse way, and, therefore, the task queues are initiated empty and the leaf nodes are added to the task queues when their parents are fulfilled. Although there are no intermediate nodes during the other PCG operations, task queues, initiated by the leaf nodes, are also managed to complete these operations, in order to use the same mapping in all the computations.

In practice, the number of tasks of each operation exceeds the number of cores, since this produces a more balanced workload during the execution. To improve it, the nodes are sorted in the shared task queue. In this way, for LwTrSv, the leaf nodes are processed before the intermediate nodes, and the nodes whose number of non-zeros is greater, are firstly processed. However, for UpTrSv, an opposite methodology is applied because the intermediate nodes should be computed as soon as possible. Although for LwTrSv and UpTrSv most of the computational work is concentrated into the leaf nodes of the TDG, the processing of the intermediate nodes introduces some overhead, and therefore the practical number of leaf nodes are limited. Moreover, the sizes of the operands in the SpMV and the vector operations also grow along with the number of levels in the TDG, and therefore, additional overhead appears. The take-away from this discussion is that, when deciding the number of levels/leaf nodes of the TDG, there is a trade-off between workload balancing and cost of processing the intermediate nodes of the TDG.

Figures 6 and 7 display, respectively, the `Extrae` [2] traces for the preconditioner computation and for one iteration of the PCG solver, executed on an Intel Xeon 8-core processor using 8 threads. The TDG in this example is composed of 32 leaves (4 leaf tasks per thread) and 6 levels, where the color of each area in the trace determines the operation (see legends). These traces show that, even with 4× more leaf nodes than threads, there still appear significant idle times and waiting times for implicit barrier at the end of parallel regions, for PrCo, SpMV, LwTrSv and UpTrSv. This facts motivate different approaches to save energy which are described in the next section.

[2] https://tools.bsc.es/extrae

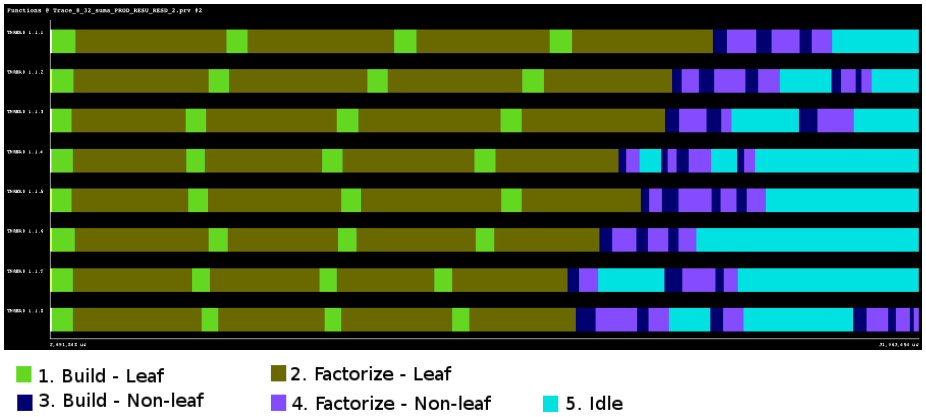

Figure 6: Execution trace of the preconditioner computation with ILUPACK for 8 threads.

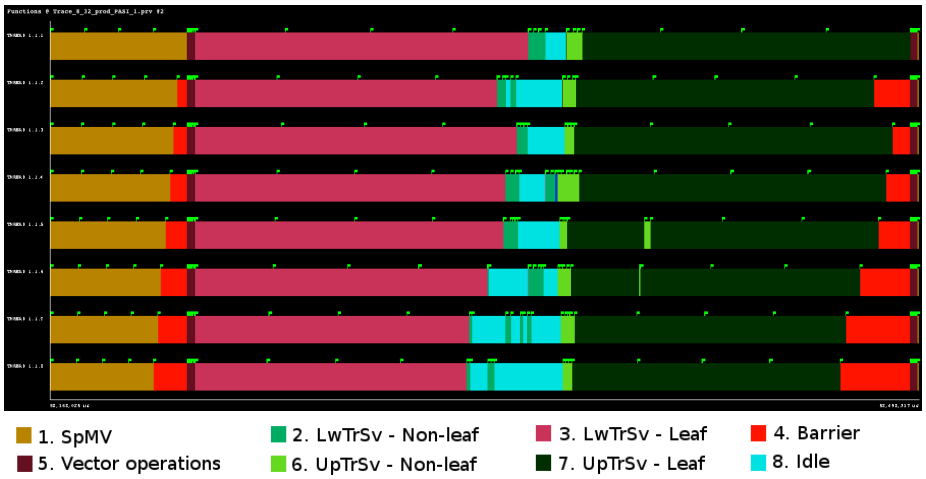

Figure 7: Execution trace of the PCG iterative solve preconditioned with ILUPACK for 8 threads.

3 Energy-aware techniques on ILUPACK

The reduction of the energy consumption in ILUPACK can be tackled by managing the C-states and P-states using different approaches. In this section we consider three different energy-aware strategies.

3.1 *Race-to-halt/race-to-idle* strategy

This strategy reduces the energy-consumption during the concurrent execution of ILUPACK transforming the "busy-wait" behaviour (threads are polling till a new task is available) to "idle-wait" (threads are blocked until a new task is ready). Benefits should be obtained because the operating system promotes the hardware into a more energy-efficient C-state.

The analysis of Figures 6 and 7 reveals two kinds of wait-time drain:

- Red areas indicate implicit OpenMP barrier at the end of parallel regions.
- Blue areas denote idle time related to task queues management.

The first ones can be transformed by assigning the value `PASSIVE` to the environment variable `OMP_WAIT_POLICY` before the code was executed. In this way, PCG operations, where the computation only affects the leaf nodes (SpMV and vector operations), can be optimized. For the second type, the solution is to synchronize the access to the task queues by using mutex and condition variables. This solution can be also applied to the operations on which the TDG has to be traversed, that is, PrCo and PrAp .

3.2 DVFS-based approach

The main objective of the DFVS approach is to apply a frequency-tuning policy, in order to reduce the waiting time as soon as possible. To complete this, the code incorporates some mechanisms to detect in which thread the waits are produced. Then, it adjusts the best P-state for the execution of each task in order to reduce the waiting time of that thread. Taking into account that the execution time of the tasks has to be measured several times to take the best decision, this technique can not be applied on the preconditioner computation, but it can be perfectly included in the PCG operations. The related overhead suggested to apply this technique only on the most expensive operations of the iterative solver: SpMV, LwTrSv and UpTrSv.

In Figure 7, it is easy to visually locate the slowest thread of the SpMV, but its calculation requires to measure the computational time of each task ($time(th)$), and, then, to obtain the thread whose execution time is the greatest one (th_{slw}). During this computation, the initial P-state of all the tasks is $P0$, and, after the thread th_{slw} is identified, the remaining threads increase the P-state of its last executed task. Afterwards, the execution time of each thread is measured again. If the measured value for a thread is greater than the current execution time for th_{slw}, the P-state of its last task is decreased, and the thread

```
/* The P-state of all tasks is initiated to P0 */
/* Initialization: Set_Threads={0, ..., maxTh − 1} , firstTime = true */
  // Execution of the PCG operation, getting time(i), i = 0, ..., maxTh − 1
  ...
  // Code included to implement the DFVS approach
  if (not empty(Set_Threads)) then
    if (firstTime)
      Identify the th_slw
      Remove th_slw from Set_Threads
      firstTime = false
    endif
    for th in Set_Threads do
      if (time(th) < time(th_slw)) then
        Increase P-State of the "last" task in th
      else
        Decrease P-State of the "last" task in th
        Remove th from Set_Threads
      endif
    endfor
  endif
```

Figure 8: Algorithmic formulation of the DVFS approach.

is removed from the procedure, otherwise, a new increment on the P-state of its last task is made. When the last task of a thread reaches the maximum P-state and a new increment should be applied, the previous task has to be modified; in fact, the detection of the last task of a thread always excludes the tasks whose P-state has arrived to the maximum. Figure 8 shows an algorithmic formulation of the described process.

The SpMVcomputation only involves leaf nodes, and, therefore, the implementation of the DVFS approach is direct. But, for LwTrSv and UpTrSv, intermediate nodes are also involved. Taking into account that the weight of these nodes is relatively small in the global computation, intermediate nodes related to both operations are not considered in this strategy. In LwTrSv, the implementation of the DVFS approach is easier because all the threads start the processing of the leaf nodes at the same time, whereas the technique assures that all the threads finalize their computation as close as possible reducing the idle time (blue area in the traces). On the other hand, the beginning of the leaf nodes in UpTrSv is unknown, because it depends on the execution time of the intermediate nodes, thence, it is more complex to adjust the finalization of the threads in this operation. Moreover, the execution of the leaf and intermediate nodes can be mixed during the execution of the PCG, generating more complex scenarios. Anyway, the DVFS approach is also useful in UpTrSv, reducing the waiting time related to the implicit barriers (red area in the traces). Figure 9 shows how the trace in Figure 7 changes when the DVFS approach is applied on PCG.

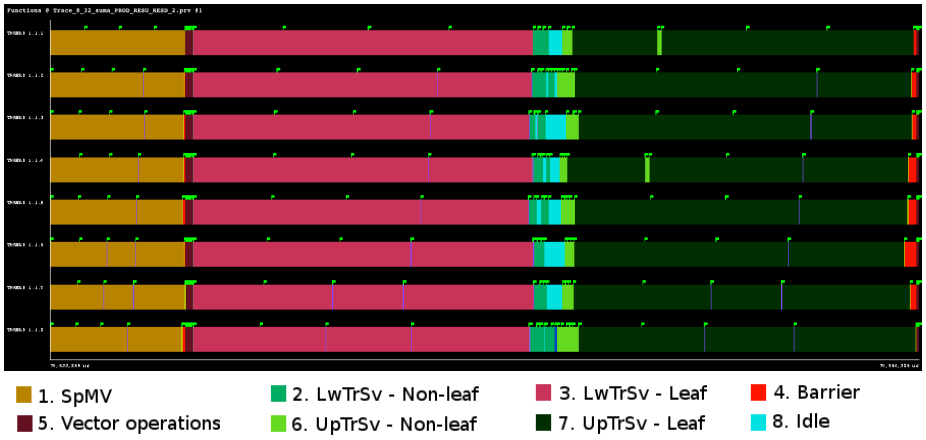

Figure 9: Execution traces of the PCG-DVFS iterative solve preconditioned with ILUPACK for 8 threads.

3.3 Memory-bound aware strategy

Many of the computations in SpMV, LwTrSv and UpTrSv are BLAS-2 kernels applied on sparse matrices. These ones are typical memory-bound operations, on which the best energy-aware implementation recommends to change the P-state related to the corresponding tasks, instead of using the highest frequency. However, the best frequency depends on the data (including its partition and its mapping), and on the machine architecture, and, therefore, it is unknown before the beginning of the method. A procedure similar to that described in Figure 8 can be defined to find this frequency, but, in this case, execution time is changed by energy consumption. The process is shown in Figure 10, where the variable *EnCon* refers to the energy consumption of all the tasks of the operation, avoiding to consider individual threads, and, consequently, the changes on the P-states affect to all threads. The parameter *ErrAllowed* allows to change the P-state of the threads although it produces a small loss of performance. In this way, the number of frequency changes in this phase is augmented and the energy efficiency is improved.

3.4 Combining energy-aware methodologies

The techniques described in the previous subsections are compatible, and therefore, they can be applied together at the same execution. In theory, the maximum benefit should be achieved when the three methodologies are applied, but this assertion will be confirmed by means of experimentation included in the next section. When the DVFS approach and the MBA strategy are combined, the algorithms have to be changed, because the application of the DVFS approach has to be made when the MBA strategy has finalized. In addition, the initialization at the beginning of Figure 8 has to be removed, because the initial frequency of the cores has been previously fixed in Figure 10.

```
/* The P-state of all tasks is initiated to P0 */
/* Initialization: firstTime = true , endProcess = false */
  // Execution of the PCG operation, getting EnCon
  ...
  // Code included to implement the MBA strategy
  if (not endProcess)
    if (not firstTime)
      if (prvEnCon < (EnCon * ErrAllowed)
        for th in {0, ..., maxTh − 1} do
          Increase P-State for all the "involved" task of th
        endfor
      else
        for th in {0, ..., maxTh − 1} do
          Decrease P-State for all the "involved" task of th
          endProcess = false
        endfor
      endif
    endif
    prvEnCon = EnCon
    firstTime = false
  endif
```

Figure 10: Algorithmic formulation of the MBA strategy.

4 Experimental Results

In this section, we present the impact of the energy-aware techniques on ILUPACK. With this aim, we compare the performance and energy efficiency of the implementation without any energy-aware technique, which is referred as Performance-Oriented (PO), and the implementations on which some of the energy-aware techniques have been included.

4.1 Hardware Setup

For the experiments included in this section, we employ a server equipped with two 8-core Intel Xeon(R) E5-2630 processors (Haswell-EP), running at 2.4 GHz, with 20 MBytes of L3 on-chip cache (LLC or last level of cache) each, and with 64 GBytes of DDR3 RAM. The operating system running in the server is Linux version 2.6.32-642.4.2.el6.centos.plus.x86_64, and the compiler is gcc 4.4.7. The *userspace* Linux governor allows the processor cores to operate at 13 possible frequencies ranging from 1.2 GHz to 2.4 GHz, with a stride of 0.1 GHz, being possible to fix each core to a different frequency.

For the generation of the first test matrix, *A200*, a large-scale linear system for the Laplacian equation $-\Delta u = f$ in a $3D$ unit cube $\Omega = [0, 1]^3$ with Dirichlet boundary conditions, $u = g$ on $\partial\Omega$, was generated, whose discretization is a sparse symmetric positive system. The second matrix in the experimentation corresponds to the sparse symmetric *audikw_1* example from the SuiteSparse

Matrix Collection [19], with close to 1,000,000 rows/columns. Table 1 shows the more important features of these two matrices.

Matrix	Dimension n	nnz	nnz/n	$numIter_{32}$
A200	8,000,000	31,880,000	3.99	107
audikw_1	943,695	39,297,771	41.64	815

Table 1: Matrices employed in the experimental evaluation.

Energy was measured using Intel's RAPL (Running Average Power Limit) interface [20], reflecting the estimated consumption of the core-uncore (*package*), DRAM and the total (core, uncore and DRAM) system. For the Haswell-EP, the isolated on-core consumption is not provided by RAPL. The idle energy was obtained by executing during 60 sec. the Linux sleep command in all cores. This value was then subtracted to the total energy in order to obtain the net energy. The experiments were executed after a warm up period of 120 sec. using a busy-wait loop, and each experiment was repeated 5 times, showing later the average values. The experiments analyze the performance and energy efficiency of the different implementations when they are executed on a single socket, and, therefore, we only show the results of the corresponding 8-core processor.

4.2 Experimental Setup

The right-hand side vector b in the iterative solvers was always initialized to the product $A(1, 1, \ldots, 1)^T$, and the PCG iteration was started with the initial guess $x_0 = 0$. The parameter that controls the convergence of the iterative process in ILUPACK, *restol*, was set to 10^{-6}, whereas the drop tolerance and the bound to the condition number of the inverse factors, which control ILUPACK's multilevel incomplete factorization process, where set to 0.01 and 5 respectively. The approximate number of PCG iterations required to solve the corresposding linear system by using these parameters, for a 32 leaf nodes TDG and employing IEEE754 real double-precision arithmetic, is also included in Table 1.

For each implementation, we have considered two different scenarios: balanced and unbalanced mapping. The first one is the default use of the library, in which the leaf nodes are sorted in the shared task queue regarding the number of nonzero elements, so that, during the preconditioner computation, the biggest nodes are factorized before. On the contrary, the second one has been manually build to generate an unbalance execution, so that it is possible to verify how the energy-aware strategies are adapted to these situations. For both scenarios, the same mapping tasks to cores is fixed for all the implementations so that, the different strategies can be properly compared.

In order to analyze the impact of each technique in the energy-consumption improvement, we have applied an incremental methodology, so that each new implementation incorporates the previously analized techniques. In this way, the DVFS implementations incorporate the RTH strategy, and the MBA implementations incorporate the two previous ones.

The DVFS approach and the MBA strategy require to measure how the operations are executed by the cores. To avoid the impact of the appearance of errors in the measurement, both strategies accumulate the results of several PCG iterations. In theory, higher number of iterations assure the correctness of the measures, but the adjustment period and its overhead are extended. In our experimentation, 2 or 4 iterations have been tried, and similar improvements were obtained, although the overhead was slower for the first case. Therefore, only the results for 2 iterations are shown in the paper.

The parametrization of the MBA strategies requires to fix the parameter *ErrAllowed*, which is set to 1.000 or 1.005, and to determine on which type of tasks the P-state is changed. Our experiments have shown that it is better to modify only the frequency of leaf nodes than to change the frequency for all the tasks.

4.3 Analysis of the Results

The performance and energy efficiency results are shown by means of tables, in which we expose the relative improvement of the corresponding variant with respect to the PO implementation. This computation is calculated as follows:

$$res(val_{IMP}) = \frac{ref_{PO}}{val_{IMP}} - 1 \quad ,$$

where ref_{PO} is the value of the PO implementation corresponding to the matrix which is used in the studied implementation (IMP). Note that, negative values for this expression reflect a decrease of the performance, whereas positive values reveal improvements. Later, we multiply this result by 100, in order to show the corresponding percentage value.

Tables 2 and 3 show the performance and energy efficiency of the different strategies for matrices A200 and `audikw_1`, respectively. Generally speaking, the first analysis of these tables can conclude that the energy improvements for `audikw_1` are always greater than those for A200, because the nonzero pattern of the first one is irregular, being more difficult to define a perfectly balanced mapping. In this way, the execution of `audikw_1` includes more idle periods and, therefore, it offers more options to apply energy-aware techniques. A similar conclusion is obtained when the two mappings are compared: these strategies are more profitable for unbalanced mapping, because there are more idle periods.

Race-to-Halt strategy. The first remark after applying the RTH methodology is that, for both matrices, kind of mappings, and stages (PrCo and PCG), the impact of this strategy on the execution time is practically negligible, because it is always less that 0.15%.

The same conclusion can be obtained for the DRAM values, since the use of the memory in this implementation has not changed respect to the PO case. Therefore, the DRAM changes are directly related to the changes in the column Time.

The analysis of the total and net values of energy reveals the main advantage of this implementation, that is, the reduction of the package values and, consequently, the global ones. This improvement is higher for the net energy and net power values.

For both matrix mappings, the previous assertions are fulfilled: this strategy reduces the energy-consumption and maintains the execution time. The relative variations of the net package energy for the PrCo are remarkable, because they are higher than 7,5% and 23%, respectively, for the unbalanced mapping of the two matrices. Furthermore, it is interesting the increment of the net package energy for the PCG stage, which is higher than 3,5% and 9%, respectively, for the unbalanced mapping of the two matrices. Moreover, the improvements are also appreciable for the balanced mappings.

Dynamic VFS-based approach. Following the incremental methodology mentioned before, we applied several DFVS implementations in the RTH case. Here, we consider three implementations:

- DVFS_1: The DVFS approach is only applied on SpMV.
- DVFS_2: The DVFS approach is applied on SpMVand LwTrSv.
- DVFS_3: The DVFS approach is applied on SpMV, LwTrSvand UpTrSv.

A new column (Add. steps) in Tables 2 and 3 is included, which collects the number of P-state increments (see Figure 8) with respect to the previous row in the table. Therefore, all the values in the DVFS_1 row correspond to the application of the DVFS approach on SpMV, the DVFS_2 row shows the impact of the strategy on LwTrSv, whereas the DFVS_3 row is focussed on UpTrSv.

In this approach, the impact on the execution time is still really small, since the descent is always less than 0.9%, revealing one of the strengths of the strategy. The growth of the execution time is basically due to the overhead of the code in Figure 8, and, therefore, it is higher when the number of P-state increments.

Unlike the RTH strategy, now there is not a direct relationship between DRAM values and execution time. One might expect that the memory would consume more energy because the threads are active more time, executing and demanding data from memory, but the tables expose that this assumption is not true in many cases. If the increment of the execution time is mainly related to the overhead of this strategy, it does not have any influence on the memory. Probably, the values of the tables show that the reduction of idle periods also decreases the overhead related to C-states transitions, and, therefore, the threads are less time executing tasks, diminishing the memory working-time. This conclusion is not true only for the case where more P-state increments are made (unbalanced mapping for `audikw_1`), presumably because, in that case, the additional active time of the threads is higher than the C-state savings.

Again, the improvement of the total and net values are the main features of this strategy. The tables exhibit the positive impact on the energy and power consumption of changing P-states, even though the transitions introduce additional overhead. Overall, the net package energy savings for the unbalanced mapping are close to 5% and 17% for matrices A200 and `audikw_1`, respectively, whereas they are still appreciated for the balanced mapping, improving 4,60% and 7,79%.

The comparison of the P-state increments in each operation allows to conclude that SpMV is the operation on which more changes are usually made. The reason could be the size of the corresponding tasks, because it is always easier to adjust the P-state for small tasks.

Memory-bound aware strategy. This strategy tries to minimize the energy consumption of the implementation, being possible to use several alternatives to achieve this objective. We have considered four different variants:

- PCK_0.0: On DVFS_3, the package energy is minimized, allowing no error.
- PCK_0.5: On DVFS_3, the package energy is minimized, allowing a 0.5% of error.
- GBL_0.0: On DVFS_3, the global energy is minimized, allowing no error.
- GBL_0.5: On DVFS_3, the global energy is minimized, allowing a 0.5% of error.

In this strategy, the meaning of the last column in the tables (Add. steps) is different from that in the DVFS approach, because, here, not only a single task changes but a P-state increment/decrement is applied on a set of tasks.

The augmentation of the execution time of the MBA strategy is really important, being close to 20% or 25%, respectively, for the unbalanced mapping of the two matrices allowing a 0.5% of error, whereas the augment is close to 15% and 13% with no error allowed. These values make sense because the objective of the strategy is to maximize the energy savings, and the increment of the execution time can be compensated by the power reduction. In any case, the performance loss is directly related to the number of P-state changes.

The increment of the execution time has a direct relationship with the DRAM consumption, although its figures are more moderate. The reason could be that, in fact, this strategy reduces the the DRAM net power. In this way, the pair (net energy, net power) for the unbalanced mapping of the two matrices are, respectively, close to (-7%,16%) and (-11%,17%) when the error is allowed and (-5%,10%) and (-6%,8%) if not.

The reduction of the energy consumption is really relevant for the package and global values. Minimizing global energy means to improve the package energy, sometimes even more that if only package energy is optimized. Moreover, allowing some errors improves the energy consumption in many cases. The saving figures are huge, being the improvements of the package net power greater

Balanced mapping											
	Total energy			Net energy			Time	Net power			Add.
	pack.	dram	global	pack.	dram	global		pack.	dram	global	steps
RTH(PRCO)	3,52	0,13	3,26	5,66	0,26	5,32	0,01	5,64	0,25	5,31	-
RTH(PCG)	2,40	0,03	2,15	3,63	0,06	3,25	-0,04	3,67	0,10	3,29	-
DVFS_1	2,64	0,02	2,36	4,11	0,16	3,69	-0,26	4,38	0,42	3,96	41
DVFS_2	2,78	0,16	2,50	4,26	0,30	3,84	-0,13	4,40	0,43	3,98	0
DVFS_3	2,96	-0,01	2,65	4,60	0,11	4,12	-0,24	4,85	0,35	4,37	5
PCK_0.0	9,00	-9,31	6,78	27,86	-5,49	23,37	-16,28	52,39	12,80	47,07	18
PCK_0.5	9,82	-11,81	7,12	34,46	-7,11	28,53	-20,13	68,34	16,30	60,92	22
GBL_0.0	9,40	-10,76	6,91	31,16	-6,56	25,91	-18,33	60,36	14,36	53,98	19
GBL_0.5	9,73	-11,99	7,01	34,54	-7,27	28,56	-20,34	68,87	16,39	61,37	22

Unbalanced mapping											
	Total energy			Net energy			Time	Net power			Add.
	pack.	dram	global	pack.	dram	global		pack.	dram	global	steps
RTH(PRCO)	4,72	0,02	4,35	7,62	-0,01	7,15	0,05	7,57	-0,06	7,10	-
RTH(PCG)	2,36	0,04	2,12	3,52	0,02	3,15	0,07	3,45	-0,05	3,08	-
DVFS_1	2,62	0,04	2,35	4,09	0,19	3,68	-0,26	4,36	0,45	3,94	50
DVFS_2	2,85	0,03	2,55	4,45	0,18	4,00	-0,28	4,74	0,46	4,28	5
DVFS_3	3,12	-0,16	2,77	4,99	0,00	4,46	-0,50	5,52	0,51	4,98	17
PCK_0.0	8,50	-8,54	6,47	24,97	-5,32	20,99	-14,51	45,99	10,70	41,35	17
PCK_0.5	9,96	-11,82	7,23	34,58	-7,21	28,62	-19,97	68,15	15,94	60,71	22
GBL_0.0	8,53	-8,59	6,48	25,00	-5,40	21,00	-14,50	45,56	10,48	40,96	15
GBL_0.5	9,71	-11,93	7,00	33,90	-7,44	28,02	-19,88	67,13	15,52	59,78	22

Table 2: Relative variation (in %) of the energy-aware variants with respect to PO, considering the balanced and unbalanced mappings of the A200 matrix when a 32-leaf TDG is processed by 8 cores.

that 67% and 90%, respectively, for the unbalanced mapping of the two matrices, and close to 68% and 84% for the balanced case. Although it could be expected that the improvement obtained with an error of 0.5% should enhance that computed without any error, it is not true for all the cases.

5 Conclusions

Several energy-aware strategies have been introduced in the paper to improve the efficiency of the task-parallel version of ILUPACK, focussing the study in the iterative solve of SPD sparse linear systems. The RTH and the DVFS strategies manage, respectively, the C-states and the P-states of the cores to reduce the energy consumption with a negligible impact on the execution time. Combining these two strategies, the improvements for the global energy and net global energy are, respectively, close to 9.5% and 15%. Additionally, the inclusion of the MBA strategy allows to achieve higher energy savings by consuming more execution time. In this case, the global energy and net global energy are, respectively, higher than 10% and 35%.

As part of future work, we would like to extend these techniques on a cluster, so that, the idle time related to communication operations can be reduced by

Balanced mapping											
	Total energy			Net energy			Time	Net power			Add.
	pack.	dram	global	pack.	dram	global		pack.	dram	global	steps
RTH(PRCO)	7,03	-0,09	6,63	11,61	0,05	11,32	-0,14	11,77	0,20	11,48	-
RTH(PCG)	2,87	-0,07	2,59	4,27	-0,07	3,88	-0,06	4,33	-0,01	3,94	-
DVFS_1	3,17	0,15	2,89	4,76	0,31	4,36	-0,14	4,90	0,45	4,50	41
DVFS_2	3,99	0,16	3,63	5,95	0,28	5,43	-0,05	6,00	0,32	5,48	22
DVFS_3	5,15	-0,09	4,65	7,79	-0,04	7,07	-0,18	7,99	0,14	7,26	19
PCK_0.0	8,05	-8,36	6,30	21,80	-5,39	18,84	-13,29	40,15	9,02	36,77	10
PCK_0.5	9,17	-16,76	6,14	38,89	-10,93	32,47	-25,56	86,44	19,62	77,83	19
GBL_0.0	8,31	-8,74	6,49	22,96	-5,54	19,83	-14,02	42,67	9,77	39,06	10
GBL_0.5	9,38	-16,36	6,38	38,67	-10,59	32,36	-25,11	84,82	19,31	76,44	19

Unbalanced mapping											
	Total energy			Net energy			Time	Net power			Add.
	pack.	dram	global	pack.	dram	global		pack.	dram	global	steps
RTH(PRCO)	13,51	-0,14	12,73	23,30	-0,10	22,76	-0,15	23,48	0,05	22,95	-
RTH(PCG)	5,93	-0,13	5,35	9,03	-0,15	8,18	-0,09	9,12	-0,06	8,28	-
DVFS_1	6,52	-0,22	5,87	9,97	-0,28	9,01	-0,13	10,10	-0,16	9,15	68
DVFS_2	8,36	-0,70	7,47	13,07	-0,85	11,74	-0,44	13,58	-0,41	12,24	70
DVFS_3	10,44	-1,36	9,25	16,71	-1,64	14,88	-0,86	17,72	-0,78	15,88	68
PCK_0.0	12,94	-9,00	10,52	31,70	-6,32	27,31	-13,28	51,77	8,01	46,72	11
PCK_0.5	12,01	-15,45	8,80	42,53	-10,36	35,72	-22,99	84,42	16,26	75,66	17
GBL_0.0	12,50	-7,88	10,29	29,06	-5,60	25,15	-11,58	45,79	6,73	41,39	8
GBL_0.5	12,03	-16,41	8,68	44,92	-10,96	37,58	-24,39	90,90	17,59	81,28	18

Table 3: Relative variation (in %) of the energy-aware variants with respect to PO, considering the balanced and unbalanced mappings of the `audikw_1` matrix when a 32-leaf TDG is processed by 8 cores.

applying RTH and DVFS strategies. Furthermore, we would like to analyze the impact of the MBA strategy to maximize the energy efficiency of different iterative solvers.

Acknowledgement

This work was supported by the CICYT project TIN2014-53495-R of the MINECO and FEDER, the H2020 EU FETHPC Project 671602 "INTERTWinE", and the project P1-1B2015-26 of the *Universitat Jaume I*.

References

[1] Duranton M, et al. The HiPEAC vision. http://www.hipeac.net/roadmap. [retrieved: July, 2017].

[2] Fuller SH, Millett LI. The future of computing performance: Game over or next level? *National Research Council of the National Academies*, 2011.

[3] Esmaeilzadeh H, Blem E, Amant RS, Sankaralingam K, Burger D. Dark silicon and the end of multicore scaling. *38th Annual International Symposium on Computer architecture - ISCA'11*, 2011; 365–376.

[4] Borkar S, Chien AA. The future of microprocessors. *Communications of the ACM* 2011; **54**(5):67–77.

[5] Aliaga JI, Badia RM, Barreda M, Bollhöfer M, Quintana-Ortí ES. Leveraging task-parallelism with OmpSs in ILUPACK's preconditioned cg method. *26th Int. Symp. on Computer Architecture and High Performance Computing (SBAC-PAD 2014)*, 2014; 262–269.

[6] Aliaga JI, Badia RM, Barreda M, Bollhöfer M, Dufrechou E, Ezzatti P, Quintana-Ortí ES. Exploiting task and data parallelism in ILUPACK's preconditioned CG solver on NUMA architectures and many-core accelerators. *Parallel Computing* 2016; **54**:97 – 107, doi:http://dx.doi.org/10.1016/j.parco.2015.12.004. URL http://www.sciencedirect.com/science/article/pii/S0167819115001581.

[7] Aliaga JI, Barreda M, Bollhöfer M, Quintana-Ortí ES. Exploiting task-parallelism in message-passing sparse linear system solvers using OmpSs. *Euro-Par 2016: Parallel Processing: 22nd Int. Conf. Parallel and Distributed Computing*, Springer, 2016; 631–643.

[8] Aliaga JI, Barreda M, Dolz MF, Martín AF, Mayo R, Quintana-Ortí ES. Assessing the impact of the CPU power-saving modes on the task-parallel solution of sparse linear systems. *Cluster Computing* 2014; **17**(4):1335–1348.

[9] Aliaga JI, Barreda M, Castaño A. Harvesting energy in ilupack via slack elimination. Zenodo, 2017, doi:10.5281/zenodo.814115. URL https://doi.org/10.5281/zenodo.814115.

[10] Aliaga JI, Dufrechou E, Ezzatti P, Quintana-Ortí ES. *Design of a Task-Parallel Version of ILUPACK for Graphics Processors*. Springer International Publishing: Cham, 2017; 91–103, doi:10.1007/978-3-319-57972-6_7. URL https://doi.org/10.1007/978-3-319-57972-6_7.

[11] Bollhöfer M, Grote MJ, Schenk O. Algebraic multilevel preconditioner for the Helmholtz equation in heterogeneous media. *SIAM J. Scientific Computing* 2009; **31**(5):3781–3805.

[12] Bollhöfer M, Saad Y. Multilevel preconditioners constructed from inverse–based ILUs. *SIAM J. Scientific Computing* 2006; **27**(5):1627–1650. Special issue on the 8–th Copper Mountain Conference on Iterative Methods.

[13] Aliaga JI, Bollhöfer M, Dufrechou E, Ezzatti P, Quintana-Ortí ES. Leveraging data-parallelism in ILUPACK using graphics processors. *13th Int. Symp. Parallel and Distributed Computing (ISPDC 2014)*, 2014; 119–126.

[14] Bollhöfer M, Aliaga JI, Martın AF, Quintana-Ortí ES. *Encyclopedia of parallel computing*, chap. ILUPACK. Springer US: Boston, MA, 2011; 917–926, doi:10.1007/978-0-387-09766-4_513. URL http://dx.doi.org/10.1007/978-0-387-09766-4_513.

[15] Karypis G, Kumar V. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing* 1999; **20**(1).

[16] Saad Y. *Iterative Methods for Sparse Linear Systems*. SIAM, 2003.

[17] Aliaga JI, Bollhöfer M, Martín AF, Quintana-Ortí ES. Exploiting thread-level parallelism in the iterative solution of sparse linear systems. *Parallel Comput.* 2011; **37**(3):183–202.

[18] The OpenMP API specification for parallel programming. http://www.openmp.org/specifications/.

[19] Davis TA, Hu Y. The university of florida sparse matrix collection. *ACM Trans. Math. Soft.* 2011; **38**(1):1–25, doi:10.1145/2049662.2049663.

[20] Intel Corp. *Intel 64 and IA-32 architectures software developer manual. Volume 3B: System programming guide, Part 2* 2015.