# Harvesting Energy in ILUPACK via Slack Elimination

José I. Aliaga[1] María Barreda[1] Asunción Castaño[1]

We develop a new energy-aware methodology to improve the energy consumption of a task-parallel preconditioned Conjugate Gradient iterative solver on a Haswell-EP Intel Xeon. This technique leverages the power-saving modes of the processor and the frequency range of the *userspace* Linux governor, modifying the CPU frequency for some operations. We demonstrate that its application during the main operations of the PCG solver can reduce its energy consumption.

## 1 Introduction

ILUPACK[2] (Incomplete LU decomposition PACKage) 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; see [2, 3] and the references therein.

Task-parallel versions of ILUPACK have also been used as a case study to explore the energy consumption and optimization of iterative solvers. Concretely, the authors of [4] investigated the benefits that an energy-aware implementation of the runtime in charge of the concurrent execution of ILUPACK produces on the time-energy balance of the application. The study in that paper reported energy savings between 7 and 13% (for Intel and AMD multicore processors from 2009-2010), with practically no penalty on performance.

In this paper we explore a new approach to save energy in the task-parallel version of ILUPACK's preconditioned Conjugate Gradient (PCG) method, leveraging the iterative nature of the method to progressively adjust the frequency of the processor cores in order to reduce idle periods and harvest energy. In rough detail, our algorithmic-based energy-saving (ABES) technique is applied to the major operations comprised by ILUPACK, namely the sparse matrix-vector product (SpMV) and the lower/upper triangular solves required for the application of the preconditioner (respectively denoted as LwTrSv and UpTrSv). Each one of these operations is divided into a collection of sub-operations, or *tasks*, to be executed in parallel. Then, by enforcing a deterministic mapping of these tasks to cores, we can detect and quantify idle periods during the first initial iterations, tuning the operating frequency of the cores to reduce these idle times.

## 2 Brief Overview of ILUPACK

ILUPACK provides C and Fortran routines for the numerical solution of sparse linear systems via Krylov subspace methods [8], combined with multilevel preconditioners that

[1]Dpto. de Ingeniería y Ciencia de Computadores, Universidad Jaume I, 12.071–Castellón, Spain, aliaga@uji.es, mvaya@uji.es, castano@uji.es

[2]`http://ilupack.tu-bs.de`

improve the numerical properties of the linear system, accelerating the convergence of the iterative solver. ILUPACK derives an efficient preconditioner from the ILU factorization of the system matrix, dropping the small entries of the factors, while relying on pivoting to bound the norm of the inverse triangular factors, to compute a numerical multilevel hierarchy of partial inverse-based approximations [5, 6].

**Exposing task-parallelism** In the remaining of this section, we focus on the parallelization strategy underlying the task-parallel versions of ILU-type iterative solvers in general, and ILUPACK PCG in particular; see, e.g., [2]. Basically, these methods exploit the connection between sparse matrices and adjacency graphs, recursively applying nested dissection to permute the sparse matrix. The goal of this re-organization of the matrix is to obtain a hierarchy of subgraphs and separators that fix the order in which the diagonal blocks have to be factorized. This process renders a task dependency graph (TDG) for the preconditioner calculation with the shape of a balanced binary tree, where the subgraphs occupy the leaves and the separators correspond to the internal nodes.

From the perspective of computational cost and complexity, the major operations in ILUPACK's PCG solve are SpMV, LwTrSv and UpTrSv, each ocurring once per iteration. With a proper organization of the data and distribution of the work, for a TDG with $l$ leaf nodes, the SpMV kernel can be decomposed into an equal number of *independent* tasks. The parallelization of the triangular solves is more complex. These kernels can both be decomposed into the same number of tasks as the TDG identified during the preconditioner calculation, maintaining the same task dependency. Thus, there exist dependencies in the binary-tree, pointing bottom-up for the lower triangular kernel and top-bottom for the upper triangular case. As a result, when these tasks are mapped to the cores, the information flows as in a reduction for LwTrSv or as in a broadcast for UpTrSv.

**Mapping tasks to cores** In practice, each operation appearing in ILUPACK's PCG is decomposed into a number of tasks that exceeds the number of cores as this produces a more balanced distribution of the workload during the execution. For LwTrSv and UpTrSv, most of the computational work is concentrated into the leaf nodes of the TDG. However, as the number of levels in the TDG is increased, the processing of the separators (non-leaf nodes) introduces some overhead, ultimately constraining the practical number of leaf nodes to a few hundreds. 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 separators of the TDG.

Figure 1 displays an `Extrae` trace for one iteration of the PCG solver, executed on an Intel Xeon 16-core processor using 16 threads. The TDG in this example is composed of 64 leaves (4 leaf tasks per thread) and 7 levels. This trace shows that, even with $4\times$ more leaf nodes than threads, there still appear significant idle times for SpMV, LwTrSv and UpTrSv, motivating the approach to save energy described in the next section.

# 3 Applying ABES in ILUPACK

In order to describe the principle underlying the ABES technique, let us consider initially a simple TDG consisting of three tasks, $\mathsf{T}_0$, $\mathsf{T}_1$, $\mathsf{T}_2$, organized into two levels, with data dependencies $\mathsf{T}_0 \rightarrow \mathsf{T}_1$, $\mathsf{T}_0 \rightarrow \mathsf{T}_2$; and $\mathsf{T}_1$, $\mathsf{T}_2$ independent of each other. (This reflects the scenario occuring during the lower triangular solve, LwTrSv).

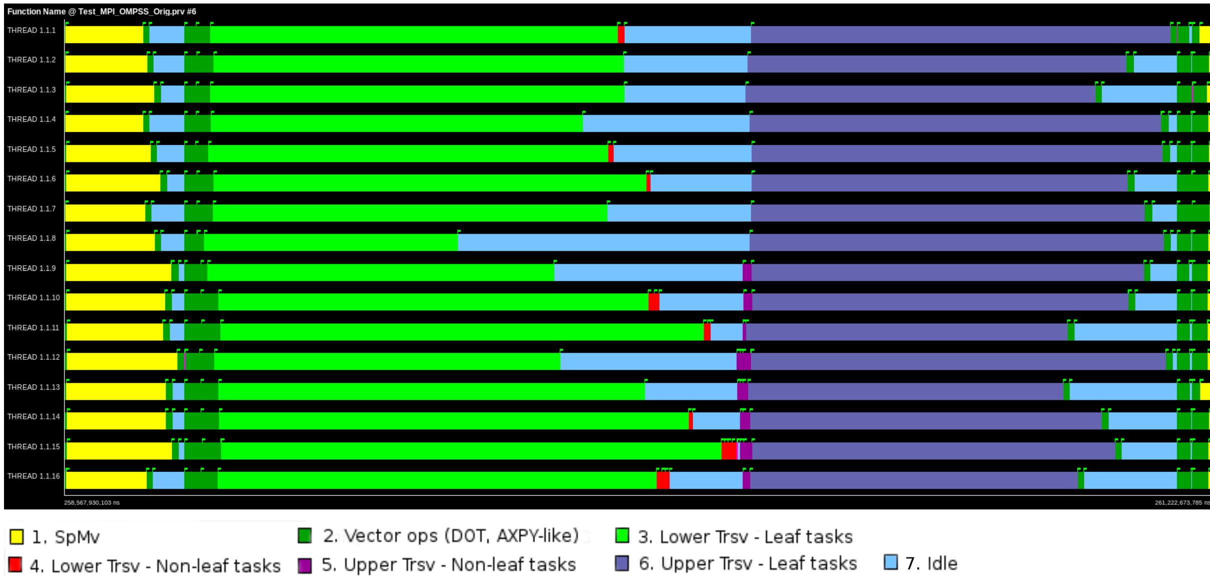


Figure 1: Execution traces of the PCG iterative solve preconditioned with ILUPACK for 16 threads.

In addition, assume a task-parallel execution using two threads, on a platform consisting of two hardware cores, denoted as $\mathsf{C}_1$, $\mathsf{C}_2$; and let us map the execution of $\mathsf{T}_1$ to $\mathsf{C}_1$ and that of $\mathsf{T}_2$ to $\mathsf{C}_2$. Thus, in case the execution time of $\mathsf{T}_1$ does not exactly match that of $\mathsf{T}_2$, due to the data dependencies, the thread in charge of the less expensive task will have to wait for its counterpart to complete its task. It is precisely this "slack" (or idle) period that we aim to eliminate with our ABES technique.

Consider the execution of ILUPACK next. The execution of the main tasks appearing in the iterative solve of ILUPACK's PCG method yield idle periods, due to an unbalanced distribution of the workload (see Figure 1), that our ABES technique targets as follows:

- For the preconditioner computation, we allow a dynamic mapping of tasks to threads, and the same mapping is enforced for the PCG solve. Furthermore, threads are assigned to specific cores (no thread migration is allowed) and all threads/cores initially proceed at the nominal frequency $f_n$. Because of the strict mapping of threads to cores, we will use "thread" to refer to both terms hereafter.

- During the first five iterations, the ABES mechanism records the termination time for each thread–operation pair, identifying the *slowest thread*. The ABES mechanism then determines the operating frequency of each thread–operation for subsequent iterations. Concretely, the frequency-tuning policy aims to slow-down the last task of all threads that terminate the execution of their tasks earlier than the slowest thread.

- Each four iterations, the ABES mechanism analyze the impact of the frequency-tuning policy. If the defined operating frequencies in a thread yields in longer execution time than the execution time of slowest thread, the last changed task is fixed to the next higher level and the corresponding thread is removed to the ABES policy. Otherwise, the policy aims to slow-down the corresponding task, or the previous task, if its minimum operating frequency has been reached.

- When all the threads are removed to the policy, the operating frequency of each thread–operation has been fixed.

# 4 Experimental Results

For the experiments in this section, we employ a server equipped with two 8-core Intel Xeon(R) E5-2630 processors (2.4 GHz), with 64 GBytes of DDR3 RAM. 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. 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.

All the experiments employed IEEE754 real double-precision arithmetic. In the first experiment we generated 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 $\partial\Omega$, and a discretization that resulted in a sparse symmetric positive system. This *A200* matrix has $8 \cdot 10^6$ rows/columns. The second matrix in the experimentation corresponds to the sparse symmetric `audikw_1` example from the SuiteSparse Matrix Collection [1], with close to 1,000,000 rows/columns.

Energy was measured using Intel's RAPL (Running Average Power Limit) interface [7], 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 power was obtained during the executing the Linux sleep command by all cores during 100 sec. This value was then subtracted to the total power in order to obtain the net energy. The experiments were executed after a warm up period of 150 sec. using a busy-wait loop, and each experiment was repeated 5 times, showing the average values.

In our energy consumption analysis, we consider the following configurations: performance-oriented (PO), energy–aware (EnAw), and three variants of ABES. For PO, the PCG is executed using a power-oblivious runtime; in contrast, for EnAw, the runtime exploits the power-saving modes of the Intel Xeon processors, promoting the idle threads to one of the power-saving C-states [4]. The ABES variants combine EnAw with the ABES technique applied to SpMV(ABES1), SpMV & LwTrSv(ABES2), or SpMV & LwTrSv & UpTrSv(ABES3).

Tables 1 and 2 report the time-power-energy of the five policies applied to the iterative solution of the two sparse examples, using a 32-leaf TDG executed on 8 cores. Two different mappings are considered: balanced and unbalanced. In the first mapping, the operations are issued in decreasing order of computational cost; the second mapping is manually generated to enforce additional ABES steps. The last column in the tables includes the ABES steps which are added to the policy sited in previous row, thus ABES1, ABES2 and ABES3 respectively represent the ABES steps in SpMV, LwTrSv and UpTrSv. Moreover, the remaining columns numbers show the relative improvement of the corresponding variant with respect to the PO policy, therefore, negative values reflect a decrease of performance, power dissipation or energy consumption. Several conclusions can be obtained from the analysis of these tables:

- The number of ABES steps is greater in unbalanced configuration than in their balanced counterparts.
- A higher number of ABES steps is necessary for `audikw_1` than for A200, mainly because the nonzero pattern of the last example is more regular.
- A significant part of the increase in the execution time is due to the introduction of EnAw while the impact of ABES variants are smaller.

| Balanced mapping | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | Total energy | | | Net energy | | | Time | Total power | | | Add. |
| | Package | DRAM | Total | Package | DRAM | Total | | Package | DRAM | Total | steps |
| EnAw | 0.62 | -0.57 | 0.49 | 1.09 | -0.55 | 0.90 | -0.65 | 1.28 | 0.07 | 1.14 | – |
| ABES1 | 0.97 | -0.72 | 0.77 | 1.56 | -0.76 | 1.29 | -0.63 | 1.61 | -0.09 | 1.41 | 21 |
| ABES2 | 1.09 | -0.65 | 0.89 | 1.73 | -0.65 | 1.46 | -0.65 | 1.75 | 0.00 | 1.55 | 0 |
| ABES3 | 0.99 | -0.74 | 0.79 | 1.62 | -0.75 | 1.35 | -0.71 | 1.72 | -0.03 | 1.52 | 5 |

| Unbalanced mapping | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | Total energy | | | Net energy | | | Time | Total power | | | Add. |
| | Package | DRAM | Total | Package | DRAM | Total | | Package | DRAM | Total | steps |
| EnAw | 0.62 | -0.58 | 0.48 | 1.14 | -0.50 | 0.95 | -0.79 | 1.42 | 0.22 | 1.28 | – |
| ABES1 | 1.75 | -0.82 | 1.45 | 2.71 | -0.83 | 2.29 | -0.81 | 2.58 | -0.01 | 2.28 | 41 |
| ABES2 | 1.72 | -0.86 | 1.42 | 2.69 | -0.85 | 2.28 | -0.87 | 2.61 | 0.01 | 2.31 | 10 |
| ABES3 | 1.57 | -0.68 | 1.31 | 2.46 | -0.63 | 2.10 | -0.84 | 2.43 | 0.16 | 2.16 | 16 |

Table 1: 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 processed by 8 cores.

| Balanced mapping | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | Total energy | | | Net energy | | | Time | Total power | | | add. |
| | Package | DRAM | Total | Package | DRAM | Total | | Package | DRAM | Total | steps |
| EnAw | 1.02 | -0.46 | 0.86 | 1.59 | -0.40 | 1.39 | -0.61 | 1.63 | 0.14 | 1.48 | – |
| ABES1 | 3.26 | -0.96 | 2.81 | 4.73 | -1.03 | 4.15 | -0.79 | 4.08 | -0.17 | 3.63 | 41 |
| ABES2 | 3.41 | -1.00 | 2.95 | 4.97 | -1.07 | 4.35 | -0.84 | 4.29 | -0.16 | 3.82 | 22 |
| ABES3 | 3.24 | -0.96 | 2.80 | 4.73 | -1.01 | 4.14 | -0.84 | 4.11 | -0.12 | 3.67 | 17 |

| Unbalanced mapping | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | Total energy | | | Net energy | | | Time | Total power | | | Add. |
| | Package | DRAM | Total | Package | DRAM | Total | | Package | DRAM | Total | steps |
| EnAw | 2.30 | -0.91 | 1.97 | 3.46 | -0.99 | 3.01 | -0.75 | 3.08 | -0.16 | 2.74 | – |
| ABES1 | 5.92 | -2.25 | 5.03 | 8.67 | -2.79 | 7.45 | -0.97 | 6.95 | -1.29 | 6.06 | 65 |
| ABES2 | 6.10 | -2.26 | 5.19 | 8.96 | -2.78 | 7.70 | -1.03 | 7.21 | -1.24 | 6.29 | 60 |
| ABES3 | 5.88 | -2.36 | 4.98 | 8.68 | -2.90 | 7.44 | -1.10 | 7.06 | -1.27 | 6.16 | 59 |

Table 2: 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 processed by 8 cores.

- In general, the improvements in net energy are higher than those observed in the total energy.
- The savings of the ABES variants occur in package energy consumption; in contrast, the DRAM energy is increased. These figures grow with the number of ABES steps.
- The application of ABES to SpMV produces larger savings than the use of the same technique in LwTrSv or UpTrSv.

# 5 Conclusions

We have introduced the ABES technique to improve the performance of energy-aware variants of a PCG solver. The results demonstrate that the application of this methodology on the main operations of the solver reduces the energy consumption with a negligible impact on the execution time. Furthermore, the technique adapts to the problem, increasing the energy savings for unbalanced mappings.

## Acknowledgements

This work was supported by the CICYT project TIN2014-53495-R of the MINECO and FEDER.


## References


[1] *The suitesparse matrix collection.* `http://www.cise.ufl.edu/research/sparse/matrices`, 2017.

[2] J. I. Aliaga, R. M. Badia, M. Barreda, M. Bollhöfer, E. Dufrechou, P. Ezzatti, and E. S. Quintana-Ortí, *Exploiting task- and data-parallelism in ILUPACK's preconditioned CG solver on NUMA architectures and many-core accelerators*, Parallel Computing, 54 (2016), pp. 97–107.

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

[4] J. I. Aliaga, M. Barreda, M. F. Dolz, A. F. Martín, R. Mayo, and E. S. Quintana-Ortí, *Assessing the impact of the CPU power-saving modes on the task-parallel solution of sparse linear systems*, Cluster Computing, 17 (2014), pp. 1335–1348.

[5] M. Bollhöfer, M. J. Grote, and O. Schenk, *Algebraic multilevel preconditioner for the Helmholtz equation in heterogeneous media*, SIAM J. Sci. Comput., 31 (2009), pp. 3781–3805.

[6] M. Bollhöfer and Y. Saad, *Multilevel preconditioners constructed from inverse–based ILUs*, SIAM J. Sci. Comput., 27 (2006), pp. 1627–1650. special issue on the 8–th Copper Mountain Conference on Iterative Methods.

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

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