Noname manuscript No.
(will be inserted by the editor)

Characterizing the Efficiency of Multicore and Manycore Processors for the Solution of Sparse Linear Systems

José I. Aliaga · María Barreda · Ernesto Dufrechou · Pablo Ezzatti · Enrique S. Quintana-Ortí

Received: date / Accepted: date

Abstract We analyze the efficiency of servers equipped with state-of-the-art general-purpose multicore processors as well as platforms based on accelerators such as graphics processing units (GPUs) and the Intel Xeon Phi. Following the proposal recently advocated in the High Performance Conjugate Gradient (HPCG) benchmark, we leverage for this purpose efficient implementations of ILUPACK, a preconditioned solver for sparse linear systems that comprises numerical kernels and data access patterns analogous to those of HPCG. Our study analyzes the (computational) performance and energy efficiency, with two different metrics for each: time/floating-point throughput for the former; and energy/floating-point throughput-per-Watt for the latter.

Keywords Sparse linear algebra · preconditioned iterative solvers · ILUPACK · energy efficiency · multicore processors · hardware accelerators.

1 Introduction

For over two decades, the *LINPACK benchmark* [9] has been employed to compile performance and throughput-per-power unit rankings of most of the world's fastest supercomputers twice per year [2]. Unfortunately, this test boils down to the LU factorization [7], a compute-bound operation that may not be representative of the performance and power dissipation experienced by many of the complex applications running in current high performance computing (HPC) sites.

J. I. Aliaga, M. Barreda, E. S. Quintana-Ortí
Depto. de Ingeniería y Ciencia de Computadores
Universitat Jaume I, 12.071–Castellón, Spain
E-mails: {aliaga,mvaya,quintana}@icc.uji.es

E. Dufrechou, P. Ezzatti
Instituto de Computación
Universidad de la República
11.300–Montevideo, Uruguay
E-mails: {edufrechou,pezzatti}@fing.edu.uy

The alternative *High Performance Conjugate Gradients (HPCG) benchmark* [1,6] has been recently introduced with the specific purpose of exercising computational units and producing data access patterns that mimic those present in an ample set of important HPC applications. This attempt to change the reference benchmark is crucial because such metrics may guide computer architecture designers, e.g. from AMD, ARM, IBM, Intel and NVIDIA, to invest in future hardware features and components with a real impact on the performance and energy efficiency of these applications.

The HPCG benchmark consists of basic numerical kernels such as the sparse matrix-vector multiplication (SpMV) and sparse triangular solve; basic vector operations as e.g. vector updates and dot products; and a simple smoother combined with a multigrid preconditioner. The reference implementation is written in C++, with parallelism extracted via MPI and OpenMP [1]. However, in an era where general-purpose processors (CPUs) as well as the Intel Xeon Phi accelerator contain dozens of cores, the concurrency level that is targeted by this legacy implementation may be too fine-grain for these architectures. Furthermore, the reference implementation is certainly not portable to heterogeneous platforms equipped with graphics processing units (GPUs) comprising thousands of simple arithmetic processing units (e.g., NVIDIA's CUDA cores).

In this paper we investigate the performance and energy efficiency of state-of-the-art multicore CPUs and many-core accelerators, using optimized multi-threaded iterative sparse linear system solvers. For this purpose, we leverage task-parallel and data-parallel versions [3,4] of ILUPACK[1] (Incomplete LU PACKage), a CG solver enhanced with a sophisticated algebraic multilevel factorization preconditioner. Compared with the HPCG benchmark, these multi-threaded implementations of ILUPACK are composed of the same sort of numerical kernels and, therefore, exhibit analogous data access patterns and arithmetic-to-memory operations ratios. On the other hand, our task-parallel version of ILUPACK is likely better suited to exploit the hardware parallelism of both general-purpose processors and the Intel Xeon Phi, while our data-parallel implementation targets the large volume of CUDA cores in NVIDIA's architectures. The main contribution of this work thus lies in the experimental evaluation of these solvers, from the point of views of performance and energy efficiency, which exposes important insights that are summarized at the end of this paper.

The rest of the paper is structured as follows. In Section 2 we review the task-parallel and data-parallel versions of ILUPACK for multi-threaded architectures based, respectively, on x86-based processors and GPUs. In Section 3 we describe the experimental setup (servers and system/application software), and we briefly introduce some relevant architecture-specific details of our parallel ILUPACK versions. Finally, in Section 4 we perform the analysis from the perspectives of execution time, performance (in terms of GFLOPS; i.e., billions of floating-point operations, or flops, per second), energy-efficiency (GFLOPS/W) and energy-to-solution (ETS). We close the paper in Section 5 with some remarks.

2 Multi-threaded Versions of ILUPACK

2.1 Overview

Given a linear system $Ax = b$, where $A \in \mathbb{R}^{n\times n}$ is sparse, $b \in \mathbb{R}^n$ and $x \in \mathbb{R}^n$ is the sought-after solution, ILUPACK integrates an "inverse-based approach" into the incomplete factorization of A, in order to obtain an algebraic multilevel preconditioner. In analogy with the HPCG benchmark, in this paper we only consider linear systems with symmetric positive definite (s.p.d.) coefficient matrix A, on which the preconditioned CG (PCG) solver underlying ILUPACK is applied.

Figure 1 describes the PCG method algorithmically. The first step of the solver (O0) corresponds to the computation of the preconditioner M, while the subsequent iteration involves a SpMV (O1), the application of the preconditioner (O5), and several vector operations (DOT products, AXPY-like updates, vector norm; in O2–O4 and O6–O8). We emphasize that the same PCG iteration is the basis of the HPCG benchmark. In the remainder of the paper, we focus on the parallelization of the PCG iteration in general (i.e., the operations in the loop of Figure 1), and the multi-threaded application of ILUPACK's preconditioner in particular (O5). For the numerical details, see [5].

2.2 Exploiting task-parallelism in ILUPACK's PCG

Our task-parallel version of ILUPACK employs the task-based programming model embedded in the OmpSs[2] framework to decompose the solver into tasks (routines annotated by the user via OpenMP-like directives) as well as to detect data dependencies between tasks at execution time (with the help of directive clauses that specify the directionality and size of the task operands). With this information, OmpSs implicitly generates a task graph during the execution, which is utilized by the CPU threads in order to exploit the task parallelism implicit to the operation via a dynamic out-of-order but dependency-aware schedule.

Let us consider the PCG iteration. The variables that appear in these operations define a partial order which enforces an almost strict serial execution. Specifically, at the $(k+1)$-th iteration,

$$\ldots \rightarrow O7 \rightarrow \overbrace{O1 \rightarrow O2 \rightarrow O4 \rightarrow O5 \rightarrow O6 \rightarrow O7}^{(k+1)\text{-th iteration}} \rightarrow O1 \rightarrow \ldots$$

must be computed in that order, but O3 and O8 can be computed any time once O2 and O4 are respectively available. Further concurrency can be exposed by dividing some of these operations into subtasks of finer granularity. As we will expose next, this is especially involved for the application of the preconditioner; see [5].

Our multi-threaded version of the preconditioner extracts the task parallelism implicit to this procedure via nested dissection [10]. To illustrate the approach, consider a graph-based symmetric reordering, defined by a permutation matrix $\bar{P} \in \mathbb{R}^{n\times n}$, such that

$$\bar{P}^T A \bar{P} = \left[\begin{array}{cc|c} A_{00} & 0 & A_{02} \\ 0 & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right]. \tag{1}$$

Computing a partial incomplete Cholesky (IC) factorizations of the two leading blocks, A_{00} and A_{11}, yields the approximation

$$\left[\begin{array}{cc|c} L_{00} & 0 & 0 \\ 0 & L_{11} & 0 \\ \hline L_{20} & L_{21} & I \end{array}\right] \left[\begin{array}{cc|c} D_{00} & 0 & 0 \\ 0 & D_{11} & 0 \\ \hline 0 & 0 & S_{22} \end{array}\right] \left[\begin{array}{cc|c} L_{00}^T & 0 & L_{20}^T \\ 0 & L_{11}^T & L_{12}^T \\ \hline 0 & 0 & I \end{array}\right] + E_{01} \tag{2}$$

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

[2] https://pm.bsc.es/ompss

$A \rightarrow M$	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
$w_k := A d_k$	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
$\tau_{k+1} := \parallel r_{k+1} \parallel_2$	O8. vector 2-norm
$k := k + 1$	
endwhile	

Fig. 1 Algorithmic formulation of the preconditioned CG method. Here, $\tau_{\max}$ is an upper bound on the relative residual for the computed approximation to the solution.

of $\bar{P}^T A \bar{P}$, where

$$S_{22} = A_{22} - (L_{20} D_{00} L_{20}^T) - (L_{21} D_{11} L_{21}^T) + E_2 \tag{3}$$

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

The block structure in (1) exposes a coarse-grain concurrency during these computations. Concretely, the permuted matrix there can be decoupled into two submatrices, so that the IC factorizations of the leading block of both submatrices can be concurrently obtained:

$$A_{22} = A_{22}^0 + A_{22}^1, \tag{4}$$

with

$$\begin{aligned} \left[\begin{array}{c|c} A_{00} & A_{02} \\ \hline A_{20} & A_{22}^0 \end{array}\right] &= \left[\begin{array}{c|c} L_{00} & 0 \\ \hline L_{20} & \mathrm{I} \end{array}\right] \left[\begin{array}{c|c} D_{00} & 0 \\ \hline 0 & S_{22}^0 \end{array}\right] \left[\begin{array}{c|c} L_{00}^T & L_{20}^T \\ \hline 0 & \mathrm{I} \end{array}\right] + E_0, \\ \left[\begin{array}{c|c} A_{11} & A_{12} \\ \hline A_{21} & A_{22}^1 \end{array}\right] &= \left[\begin{array}{c|c} L_{11} & 0 \\ \hline L_{21} & \mathrm{I} \end{array}\right] \left[\begin{array}{c|c} D_{11} & 0 \\ \hline 0 & S_{22}^1 \end{array}\right] \left[\begin{array}{c|c} L_{11}^T & L_{21}^T \\ \hline 0 & \mathrm{I} \end{array}\right] + E_1. \end{aligned} \tag{5}$$

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

$$\begin{aligned} S_{22}^0 &= A_{22}^0 - \left(L_{20} D_{00} L_{20}^T\right) + E_2^0; \\ S_{22}^1 &= A_{22}^1 - \left(L_{21} D_{11} L_{21}^T\right) + E_2^1. \end{aligned} \tag{6}$$

However, (3) involves a synchronization before the addition of these two blocks can be computed

$$E_2 \approx E_2^0 + E_2^1 \ \rightarrow \ S_{22} \approx S_{22}^0 + S_{22}^1. \tag{7}$$

To unveil 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, a reordering and renaming of blocks yields a block structure similar to (1), from which four submatrices can be disassembled:

$$\left[\begin{array}{cccc|cc|c} A_{00} & 0 & 0 & 0 & A_{04} & 0 & A_{06} \\ 0 & A_{11} & 0 & 0 & A_{14} & 0 & A_{16} \\ 0 & 0 & A_{22} & 0 & 0 & A_{25} & A_{26} \\ 0 & 0 & 0 & A_{33} & 0 & A_{35} & A_{36} \\ \hline A_{40} & A_{41} & 0 & 0 & A_{44} & 0 & A_{46} \\ 0 & 0 & A_{52} & A_{53} & 0 & A_{55} & A_{56} \\ \hline A_{60} & A_{61} & A_{62} & A_{63} & A_{64} & A_{65} & A_{66} \end{array}\right] \tag{8}$$

The factorization of the diagonal blocks in this expression are done using a multilevel approach as well, yielding a recursive calculation of the preconditioner and its application during the PCG iteration.

Figure 2 illustrates the dependency tree for the factorization of the diagonal blocks in (8) during the computation of the preconditioner (O0). The edges of the preconditioner *directed acyclic graph* (DAG) define the dependencies between the diagonal blocks (tasks), i.e., the order in which these blocks of the matrix have to be processed. On the other hand, at each iteration of the PCG, the preconditioner DAG has to be traversed two times per solve $z_{k+1} := M^{-1} r_{k+1}$, once from bottom to top and a second time from top to bottom (with dependencies/arrows reversed in the DAG), in order to complete the recursion step in the task-parallel case; see [5].

The task-parallel version of ILUPACK partitions the original matrix into a number of decoupled blocks, and then delivers a partial multilevel IC factorization during the computation of (5), with some differences with respect to the sequential procedure [5]. Concretely, although the recursive definition of the preconditioner is still valid in the task-parallel case, some recursion steps are now related to the edges of the corresponding preconditioner DAG. Different preconditioner DAGs thus involve distinct recursion steps yielding distinct preconditioners, which nonetheless exhibit close numerical properties to that obtained with the original (sequential) version of ILUPACK [5].

2.3 Data-Parallel ILUPACK

In [4], we introduced a data-parallel implementation of ILUPACK that off-loads the application of the multilevel preconditioner to the GPU, performing this operation via *ad-hoc* kernels and the CUDA, CUBLAS and cuSPARSE libraries [8]. Concretely the residual r_{k+1} is transferred to the GPU when the preconditioner is to

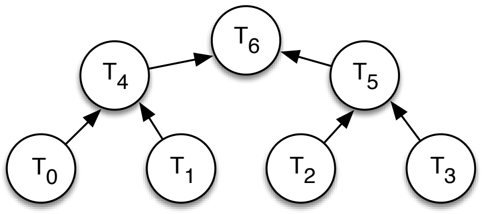

Fig. 2 Dependency tree of the diagonal blocks. Task T_j is associated with block A_{jj}.

be applied, the application (all levels) proceeds in the accelerator yielding $z_{k+1} := M^{-1}r_{k+1}$, and the residual z_{k+1} is retrieved back to the CPU upon completion.

The current version is enhanced to also off-load the SpMV present in the solver to the GPU. This requires that, at the beginning of each PCG iteration, d_k is transferred from the CPU to the GPU, the SpMV $w_k := Ad_k$ is computed there, and the result w_k is then recovered to the CPU. Matrix A is transferred to the GPU memory before the PCG iteration commences and resides there, together with the preconditioner data, for the duration of the solve. The matrix is stored in CSR format [10] and the SpMV is performed via the implementation of this kernel in cuSPARSE. In general, the vector operations contribute little to the computational cost of the solver. Therefore, these operations are performed in the CPU.

To close this brief review, we emphasize that the data-parallel version of ILUPACK proceeds exactly in the same manner as the sequential implementation and, therefore, preserves the semantics of a serial execution. This implies that both codes roughly require the same number of iterations to converge and perform the same number of arithmetic operations (with any differences due to rounding errors).

3 Experimental Setup and Implementation

3.1 Hardware and software configurations

All the experiments employed IEEE 754 real double-precision (DP) arithmetic on the following four platforms:

- SANDY: A server equipped with two hexacore Intel Xeon E5-2620 ("Sandy Bridge") processors (total of 12 cores) running at 2.0 GHz with 32 Gbytes of DDR3 RAM. The compiler is `gcc` 4.4.7.
- HASWELL: A system with two hexacore Intel Xeon E5-2603v3 ("Haswell") processors (total of 12 cores) at 1.6 GHz with 32 Gbytes of DDR4 RAM. The compiler is `gcc` 4.4.7.
- XEON PHI: A board with an Intel Xeon Phi 5110P co-processor. (The tests on this board were ran in native mode and, therefore, the specifications of the server are irrelevant.) The accelerator comprises 60 x86 cores running at 1,053 MHz and 8 Gbytes of GDDR5 RAM. The Intel compiler is `icc` 13.1.3.
- KEPLER: An NVIDIA K40 board ("Kepler" GK110B GPU with 2,880 cores) with 12 Gbytes of GDDR5 RAM, connected via a PCI-e Gen3 slot to a server equipped with an Intel i7-4770 processor (4 cores at 3.40 GHz) and 16 Gbytes of DDR3 RAM. The compiler for this platform is `gcc` 4.9.2, and the codes are linked to CUDA/cuSPARSE 6.5.

Other software included ILUPACK (2.4), the Mercurium C/C++ compiler/Nanox (releases 1.99.7/0.9a for SANDY, HASWELL and XEON PHI) with support for OmpSs, and METIS (5.0.2) for the graph reorderings.

Power/energy was measured via RAPL in SANDY and HASWELL, reporting the aggregated dissipation from the packages (sockets) and the DRAM chips. For XEON PHI we leveraged routine `mic_get_inst_power_readings` from the `libmicmgmt` library to obtain the power of the accelerator. In KEPLER, we use RAPL to measure the consumption from the server's package and DRAM, and NVML library to obtain the dissipation from the GPU.

Matrix	Size (n)	#Nonzeros (n_z)	Row density (n_z/n)
A171	5,000,211	19,913,121	3.98
A252	16,003,008	63,821,520	3.98
A318	32,157,432	128,326,356	3.98

Table 1 Laplace matrices employed in the evaluation.

For the analysis, we employed a s.p.d. linear system arising from the finite difference discretization of a 3D Laplace problem, with three instances of different size; see Table 1. In the experiments, all entries of the right-hand side vector b were initialized to 1, and the PCG was started with the initial guess $x_0 \equiv 0$. For the tests, the parameters that control the fill-in and convergence of the iterative process in ILUPACK were set as `droptool` = 1.0E-2, `condest` = 5, `elbow` = 10, and `restol` = 1.0E-6.

We use GFLOPS and GFLOPS/W to assess, respectively, the performance and energy consumption of the parallel codes/platforms. ILUPACK is in part a memory-bound computation. Therefore, an alternative performance metric could have been based on the attained memory transfer rate (Gbytes/s). Nevertheless, given that the data matrices are all off-chip, and ILUPACK performs a number of flops that is proportional to the volume of memory accesses, we prefer to stand with the more GFLOPS metric. This measure has the advantage of being more traditional among the HPC community.

3.2 Optimizing performance in the OmpSs version

The task-parallel version of ILUPACK based on OmpSs applies two architecture-aware optimization strategies:

– For multisocket servers, (e.g. SANDY and HASWELL,) we accommodate a NUMA-aware execution via the NANOS[3] environment variable `NX_ARGS` with the argument `--schedule=socket` combined with a careful modification of the ILUPACK code. Concretely, our code records in which socket each task was executed during the initial calculation of the preconditioner. This information is subsequently leveraged, during all iterations of the PCG solve, to enforce that tasks which operate on the same data that was generated/accessed during the preconditioner calculation are mapped to the same socket where they were originally executed.
– A critical aspect in the Intel Xeon Phi is how to bind the OmpSs threads to the hardware threads/cores in order to distribute the workload. In our executions, this mapping is controlled using the NANOS runtime environment variable `NX_ARGS`, passing the appropriate values via arguments `--binding_stride`, `--binding_start` and `--smp_workers`. In our experiments we evaluate several configurations of these parameters to balance the workload distribution and achieve an optimal saturation of the hardware cores.

3.3 Saving energy in the OmpSs version

The OmpSs runtime allows the user to trade off performance for power (and, hopefully, energy) consumption by controlling the behaviour of idle OmpSs threads, setting it to a range of modes that vary between pure blocking (idle-wait) and polling (busy-wait). To execute our application in blocking mode, we set the arguments `--enable-block` and `--spins=1` in the `NX_ARGS` NANOS environment variable. The first parameter enables the blocking mode while the second one indicates the number of spins before an idle thread is blocked. For the polling mode, we simply do not include the option `--enable-block`; we set `--enable-yield`, which forces threads to yield on an idle loop and a conditional wait loop; and we set `--yields=1000000` to specify the number of yields before blocking an idle thread.

[3] `http://pm.bsc.es/nanox`

3.4 Saving energy in the data-parallel version

On heterogeneous platforms, consisting of a CPU and a GPU, our data-parallel version off-loads a significant part of the computations to the graphics accelerator rendering the CPU idle for a significant fraction of the execution. In this scenario, a potential source of energy savings is to operate in the CUDA blocking synchronization mode, which allows that the operating system puts the CPU to sleep (i.e., to promote it to a deep C-state) when idle.

4 Experimental Evaluation

4.1 Tuning the parallel execution

There exists a considerable variety of factors that affect the efficiency of a parallel application on a target hardware. Among these, we next analyze the following configuration parameters:

– *Degree of software concurrency*, i.e., the number of threads that execute the application.
– *Operation "behaviour" of idle threads (CPU power states or C-states).* A thread without work can remain in an active state, polling for a new job to execute. Alternatively, it can be suspended (blocked) and awakened when a new job is assigned to it. The polling mode favors performance at the expense of higher power consumption in some platforms. The blocking mode, on the other hand, can produce lower power consumption, by allowing the operating system to promote the suspended core into a power-saving C-state, but may negatively impact the execution time because of the time it takes to reset the core into the active C0 state. The effect of these two modes on energy efficiency is uncertain, as energy is the product of time and power.
– *Operation frequency of active threads (CPU performance states or P-states).* Active threads can operate on a range of frequency/voltage pairs (P-states) that, for the Intel platforms evaluated in this work, can only be set on a per socket basis (i.e., for all cores of the same socket). These modes are controlled by the operating system, though the user can provide some general guidelines via the *Linux*

governor modes. In general, the P-states provide a means to trade off performance for power dissipation for active cores/sockets.

– *Binding of threads to hardware cores.* The degree of software concurrency translates into the exploitation of a certain level of hardware parallelism depending on the mapping of the software threads to the hardware (physical) cores. For the execution of numerical codes on general-purpose x86 CPUs, the standard approach maps one thread per core. For specialized hardware such as the Intel Xeon Phi (as well as the IBM Power A2), better results may be obtained by using 2 or 4 software threads per core.

The initial experiments in the remainder of this subsection aim to tune the previous configuration parameters for the execution of the task-parallel version of ILUPACK on SANDY, HASWELL and XEON PHI. For this purpose, we select the largest dataset that fits into the memory of each platform (A318 for both SANDY and HASWELL, and A171 on XEON PHI), and evaluate the GFLOPS and GFLOPS/W metrics as the number of threads grows. A direct comparison between the XEON PHI and the two general-purpose x86 platforms cannot be done at this point. For the task-parallel version of ILUPACK, the degree of software concurrency determines the number of tasks that should be present in the bottom level of the dependency tree (see Section 2.2), and the actual number of flops that is required for the solution of each problem case.

Figure 3 reports the performance and energy efficiency attained with the task-parallel version of ILUPACK, on SANDY and HASWELL, when OmpSs is instructed to operate in either the polling and blocking modes (see subsection 3.3). This first experiment reveals that the impact of these modes on both metrics is minor when up to 8 threads are employed. However, for 12 threads, we can observe quite a different behaviour depending on the target platform. Concretely, for SANDY, it is more convenient to rely on the blocking mode, especially from the point of view of GFLOPS/W while, for HASWELL, the polling mode yields superior performance and energy efficiency over its blocking counterpart. According to these results, in the following experiments we select the blocking and polling modes for SANDY and HASWELL, respectively.

Figure 4 evaluates the impact of three Linux governors available in SANDY and HASWELL: `performance`, `ondemand` and `userspace`, with the latter set so that the sockets operate in either the maximum or minimum frequencies (f_{max} and f_{min}, respectively) of the corresponding platforms (f_{max}=2.0 GHz and f_{min}=1.2 GHz for SANDY; and f_{max}=1.6 GHz and f_{min}=1.2 GHz for HASWELL). The four plots in the figure reveal the small impact of this configuration parameter on the performance and energy efficiency of the task-parallel version of ILUPACK on both servers, which is only visible when 12 threads/cores are employed in the execution. Given these results, we select the `userspace` governor, with the P0 state (i.e., maximum frequency), in the remaining experiments with these two platforms.

The last experiment with the configuration parameters, in Figure 5, exposes the effect of populating each hardware core of XEON PHI with 1, 2, 4 (software) threads (`Binding=4,2,1`, respectively). The best choice is clearly the first option which, given a fixed number of threads, maximizes the number of hardware cores employed in the experiment. This will be the configuration adopted for the following experiments with this platform.

4.2 Characterization of the platforms

Table 2 evaluates the task-parallel version of ILUPACK running on SANDY, HASWELL or XEON PHI, compared with the data-parallel version of the solver executed on KEPLER, using four efficiency metrics: execution time, GFLOPS, (total) energy-to-solution, and GFLOPS/W. For the Intel-based platforms, we use 12 threads in both SANDY and HASWELL, and 64 for XEON PHI.

A direct comparison of the platforms, using the same problem case is difficult: First, due to the small memory of the XEON PHI, the largest problem that could be solved in this platform (A171) seems too small to exploit the large amount of hardware parallelism of this accelerator. In addition, increasing the problem dimension shows different trends depending on the platform, with a decline in the GFLOPS and GFLOPS/W rates for SANDY and HASWELL, but a raise for KEPLER. Finally, even if the problem case is the same, the solvers do not necessarily perform the same amount of operations to converge as the exact number of flops depends, e.g., of the level of task-parallelism tackled by each solver/platform (12 tasks in the bottom level of the DAG for SANDY and HASWELL, 64 for XEON PHI, and a single task for KEPLER) as well as variations due to rounding errors, which affect the convergence rate.

As an alternative, let us perform a comparison based on the largest problem case that can be tackled on each platform: A318 for SANDY and HASWELL, A171 for XEON PHI and A252 for KEPLER. Consider first the two platforms equipped with general-purpose CPUs. As the two system comprise 12 cores, in principle we could expect better performance from HASWELL because the floating-point units (FPUs) available in this recent architecture can produce up to 16 DP flops/cycle compared with the 8 DP flops/cycle of SANDY. However, the irregular data access patterns present in ILUPACK

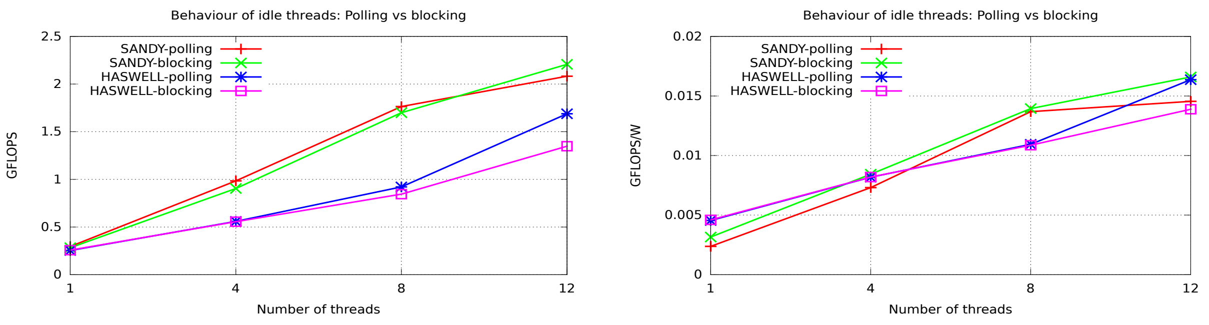

Fig. 3 GFLOPS (left) and GFLOPS/W (right) obtained with the task-parallel version of ILUPACK on SANDY and HASWELL, using the blocking and polling modes for benchmark A318.

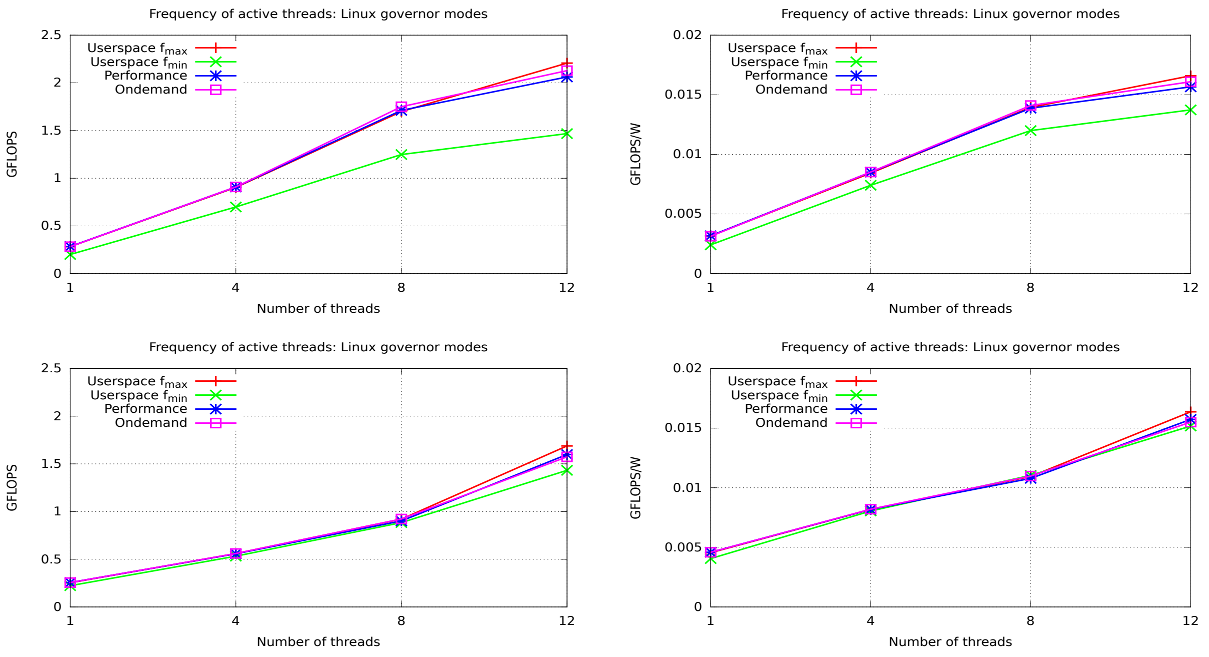

Fig. 4 GFLOPS (left) and GFLOPS/W (right) obtained with the task-parallel version of ILUPACK on SANDY (top) and HASWELL (bottom), using different Linux governors for benchmark A318.

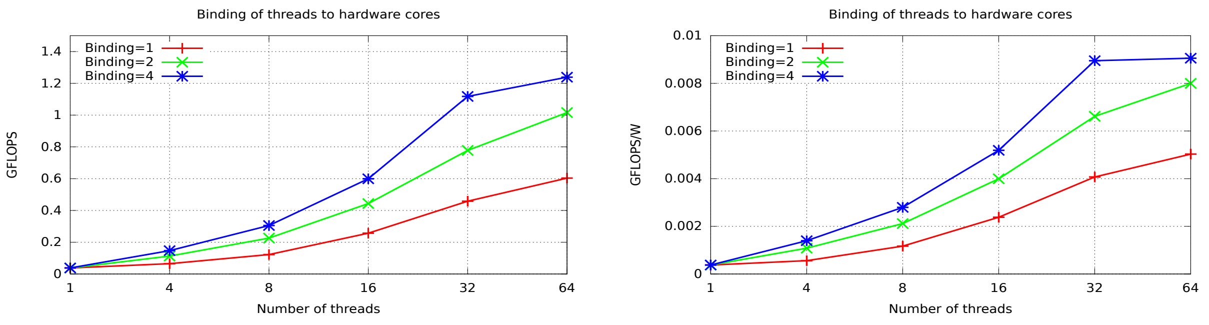

Fig. 5 GFLOPS (left) and GFLOPS/W (right) obtained with the task-parallel version of ILUPACK on XEON PHI, using different binding options for benchmark A171.

turns the exploitation of the wide vector units (SIMD) into a difficult task which, combined with the higher maximum frequency of SANDY, explains why this platform outperforms HASWELL. Interestingly, the gap between the GFLOPS rates of these two platforms, a factor of about 2.21/1.66=1.30, is captured to high accuracy by the difference between their maximum frequencies, 2.0/1.6=1.25. This variation is not reflected in the ETS and GFLOPS/W metrics, where HASWELL is only slightly behind SANDY. These particular trends make

Platform	Matrix	Time (s)	GFLOPS	Energy (J)	GFLOPS/W
SANDY	A171	21.12	2.95	2,827.89	0.0221
	A252	101.42	2.74	13,843.17	0.0201
	A318	322.06	2.21	42,827.13	0.0166
HASWELL	A171	31.89	1.95	3,277.67	0.0193
	A252	154.04	1.80	15,933.05	0.0174
	A318	421.13	1.69	43,419.49	0.0164
XEON PHI	A171	58.69	1.24	8,032.32	0.0090
KEPLER	A171	23.09	2.49	2,909.34	0.0198
	A252	83.82	3.16	11,449.81	0.0231

Table 2 Characterization of the four platforms obtained with the task-parallel and data-parallel versions of ILUPACK.

us believe that HASWELL could match SANDY's performance and outperform its energy efficiency if both platform were operated with the same maximum frequency. Let us include KEPLER and benchmark A252 in the comparison now. As the problems being solved are different, we will perform the comparison in terms of GFLOPS and GFLOPS/W, and obviate time and ETS. In spite of the large number of FPUs in KEPLER, we see that the difference in favor of this data-parallel architecture is moderate, with a factor of 1.43 and 1.86 over SANDY and HASWELL, respectively, in the GFLOPS rate; and roughly 1.40 over any of the two systems in the GFLOPS/W metric. Finally, we note that XEON PHI lags behind any of the other three platforms in both GFLOPS and GFLOPS/W.

5 Concluding Remarks

We have conducted an analysis of recent multicore technology from Intel, an state-of-the-art many-core GPU from NVIDIA, and the Intel Xeon Phi hardware accelerator. This study is relevant because it relies on ILUPACK, a package that features numerical kernels and data access patterns analogous to those of HPCG; and moreover we employ efficient implementations of this iterative solver for sparse linear systems that exploit either task-parallelism, when the target is an x86-based multicore CPU or accelerator, or data-parallelism, when the target is a GPU.

One key advantage of x86-based architectures over GPUs is the presence of well-known parallel programming interfaces, with standard tools such as OpenMP and MPI more amenable to programmers. However, our experience suggests that, for this particular class of applications, the difficulties of the data-parallel programming model are partially overcome by the existence of data-parallel numerical libraries. Furthermore, the concurrency intrinsic in some applications may be easier to extract at a data-parallel level, as we demonstrate for ILUPACK, favoring the implementation on a GPU. On the other hand, exploiting the concurrency at a task-level, while doable, requires a significant rewrite of ILUPACK to maintain semantics close to those of the sequential version, but nonetheless results in slightly different (sometimes worse) convergence properties.

Finally, many-core accelerators such as the Intel Xeon Phi and GPUs are generally preferred for their high performance and appealing energy efficiency. However, recent general-purpose processors, such as Intel's Sandy-Bridge and Haswell micro-architectures, have evolved rapidly to integrate wider SIMD (vector) units and aggressive energy saving mechanisms, blurring part of the energy gap in favor of accelerators.

Acknowledgements This work was supported by the CICYT project TIN2011-23283 of MINECO and FEDER, and the EU Project FP7 318793 "EXA2GREEN".

References

1. HPCG - high performance Conjugate Gradients (2015). `https://software.sandia.gov/hpcg`
2. The TOP500 list (2015). `http://www.top500.org`
3. Aliaga, J.I., Badia, R.M., Barreda, M., Bollhöfer, M., Quintana-Ortí, E.S.: Leveraging task-parallelism with OmpSs in ILUPACK's preconditioned CG method. In: 26th Int. Symp. on Computer Architecture and High Performance Computing (SBAC-PAD), pp. 262–269 (2014)
4. Aliaga, J.I., Bollhöfer, M., Dufrechou, E., Ezzatti, P., Quintana-Ortí, E.S.: Leveraging data-parallelism in ILUPACK using graphics processors. In: 13th Int. Symp. Parallel Distr. Comp. (ISPDC), pp. 119–126 (2014)
5. Aliaga, J.I., Bollhöfer, M., Martín, A.F., Quintana-Ortí, E.S.: Exploiting thread-level parallelism in the iterative solution of sparse linear systems. Parallel Computing **37**(3), 183–202 (2011)
6. Dongarra, J., Heroux, M.A.: Toward a new metric for ranking high performance computing systems. Sandia Report SAND2013-4744, Sandia National Lab. (2013)
7. Golub, G.H., Loan, C.F.V.: Matrix Computations, 3rd edn. The Johns Hopkins Univ. Press, Baltimore (1996)
8. NVIDIA Corporation: cuSPARSE library. Version 7.0 (2015). `http://docs.nvidia.com/cuda/cusparse/`
9. Petitet, A., Whaley, R.C., Dongarra, J., Cleary, A.: HPL - a portable implementation of the high-performance Linpack benchmark for distributed-memory computers (2008). `http://www.netlib.org/benchmark/hpl`
10. Saad, Y.: Iterative Methods for Sparse Linear Systems. SIAM (2003)