Exploiting Task and Data Parallelism in ILUPACK's Preconditioned CG Solver on NUMA Architectures and Many-core Accelerators

José Ignacio Aliaga[a], Rosa M. Badia[b], Maria Barreda[a], Matthias Bollhöfer[c], Ernesto Dufrechou[d], Pablo Ezzatti[d], Enrique S. Quintana-Ortí[a]

[a] *Dpto. de Ingeniería y Ciencia de Computadores, Universitat Jaume I, Castellón, Spain.*
[b] *Barcelona Supercomputing Center (BSC-CNS) and Artificial Intelligence Research Institute (IIAA). Spanish National Research Council (CSIC), Barcelona, Spain.*
[c] *Institute of Computational Mathematics, TU Braunschweig, Braunschweig, Germany.*
[d] *Instituto de la Computación, Universidad de la República, Montevideo, Uruguay.*

Abstract

We present specialized implementations of the preconditioned iterative linear system solver in ILUPACK for Non-Uniform Memory Access (NUMA) platforms and many-core hardware co-processors based on the Intel Xeon Phi and graphics accelerators. For the conventional x86 architectures, our approach exploits task parallelism via the OmpSs runtime as well as a message-passing implementation based on MPI, respectively yielding a dynamic and static schedule of the work to the cores, with different numeric semantics to those of the sequential ILUPACK. For the graphics processor we exploit data parallelism by off-loading the computationally expensive kernels to the accelerator while keeping the numeric semantics of the sequential case.

Key words: Sparse linear systems, preconditioned iterative solvers, Conjugate Gradient (CG) method, task and data parallelism, multi-core processors, Intel Xeon Phi, graphics processing units (GPUs)

Email addresses: aliaga@icc.uji.es (José Ignacio Aliaga), rosa.m.badia@bsc.es (Rosa M. Badia), mvaya@icc.uji.es (Maria Barreda), m.bollhoefer@tu-bs.de (Matthias Bollhöfer), edufrechou@fing.edu.uy (Ernesto Dufrechou), pezzatti@fing.edu.uy (Pablo Ezzatti), quintana@icc.uji.es (Enrique S. Quintana-Ortí)

1. Introduction

The solution of large sparse systems of equations is a key linear algebra problem arising, among others, in quantum physics, circuit and device simulation, and in general all sorts of applications involving the discretization of partial differential equations (PDEs), nonlinear sparse equations, and large-scale eigenvalue computations.

ILUPACK[1] (incomplete LU decomposition PACKage) is a numerical package that contains highly efficient multilevel incomplete LU (ILU) factorization solvers, based on Krylov subspace methods [1], for large-scale sparse application problems with up to millions of equations [2, 3, 4]. For 3D problems [5], ILUPACK often outperforms direct sparse methods [6].

In past work, we proposed the exploitation of task-level parallelism in ILUPACK's preconditioned Conjugate Gradient (PCG) solver, via OpenMP[2] for shared memory parallel computers [7] and MPI[3] for small clusters of multicore processors [8]. More recently, in [9] we refurnished our original OpenMP version of ILUPACK, porting it to the OmpSs[4] data-flow task parallel framework. Furthermore, there we introduced task priorities to accelerate the execution of critical tasks and merged small suboperations in the PCG iteration to reduce task management overhead. In these task parallel solvers, the sequential semantics of ILUPACK are traded off for increasing levels of task concurrency. Alternatively, in [10] we explored an approach that leverages data-level parallelism in the application of ILUPACK's preconditioner while preserving the numerical semantics of a sequential execution. This alternative employs NVIDIA's CUDA[5] programming interface to obtain a parallel execution on multicore servers equipped with a graphics accelerator (or GPU) by off-loading the computationally-intensive parts to the GPU.

In this work we revisit our task parallel and data parallel versions of ILUPACK, making the following new contributions:

- Our task parallel implementations target a pair of "conventional" x86-based architectures with large numbers of cores: an Intel Xeon Phi 60-core accelerator and a NUMA (non-uniform memory access) server

[1] `http://ilupack.tu-bs.de`
[2] `http://www.openmp.org`
[3] `http://www.mpi-forum.org`
[4] `https://pm.bsc.es/ompss`
[5] `http://www.nvidia.com/object/cuda`

with 4 AMD Opteron 6276 sockets and 64 cores. For the data parallel version, we consider a Kepler GPU with 2,496 CUDA cores.

- For the task parallel version of ILUPACK based on OmpSs, we reformulate our previous implementation to exploit nested parallelism in order to tackle the ample hardware concurrency of the Intel- and AMD-based systems. In addition, we analyze the benefits of a "scattered" mapping of the threads on the Intel Xeon Phi and we enhance the solver to produce a NUMA-aware execution for the AMD server.
  Alternatively, on these two conventional platforms we also explore the use of the MPI-based implementation of ILUPACK to extract task parallelism and transparently deal with NUMA effects. On the Intel Xeon Phi, we expose the similarities between the thread mapping strategy and the MPI rank mapping policy in this case.
- We enhance our previous data parallel, GPU-enabled implementation to off-load the sparse matrix-vector (SpMV) product to the GPU in addition to the application of the preconditioner during the PCG method.
- Finally, we use a common reference application to experimentally evaluate these parallelization alternatives (OmpSs or MPI combined with task parallelism vs CUDA combined with data parallelism), target platforms (AMD x86 many-core server, Intel Xeon Phi accelerator, NVIDIA GPU) and numerical semantics (sequential vs task parallel) from the perspectives of performance, convergence rate and numerical accuracy.

The rest of the paper is structured as follows. In section 2 we review the basic principles of the multilevel preconditioned solver in ILUPACK, and its task parallel variant. In sections 3 and 4, we introduce the main changes (improvements) applied, respectively, to the task parallel and data parallel implementations. In section 5 we evaluate the impact of these modifications and compare the solvers using a common experimental framework. Finally, in section 6 we close the paper with a few concluding remarks.

2. Overview of ILUPACK

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

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

Figure 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.

Figure 1 offers an algorithmic description of the PCG method. The computation of the preconditioner M is the first step of the solver (O0). The subsequent iteration involves a SpMV (O1), the application of the preconditioner (O5), and several vector operations (DOT products, AXPY-like updates, 2-norm; in O2–O4 and O6–O8). In the remainder of this section, we mainly focus on the computation and application of the preconditioner, which are by far the most challenging operations.

2.1. Sequential (and data parallel) ILUPACK

Computation of the preconditioner. This operation of ILUPACK relies on the Crout variant of the incomplete Cholesky (IC) factorization, yielding the approximation $A \approx L\Sigma L^T$, with $L \in \mathbb{R}^{n \times n}$ sparse lower triangular and $\Sigma \in \mathbb{R}^{n \times n}$ diagonal. Before the factorization commences, a scaling and a reordering (defined respectively by $P, D \in \mathbb{R}^{n \times n}$) are applied to A in order to improve the numerical properties as well as reduce the fill-in in L. After these initial transforms, the factorization operates on $\hat{A} = P^T DADP$. At each step of the Crout variant, the "current" column of $\hat{A}$ is initially updated with respect to the previous columns of the triangular factor L, and the current column of L is then computed. An estimation of the norm of the inverse of L, with the new column appended, is obtained next. If this estimation is below a predefined threshold κ, the new column is accepted into the factor; otherwise the updates are reversed, and the corresponding row and column of $\hat{A}$ are moved to the bottom-right corner of the matrix. This process is

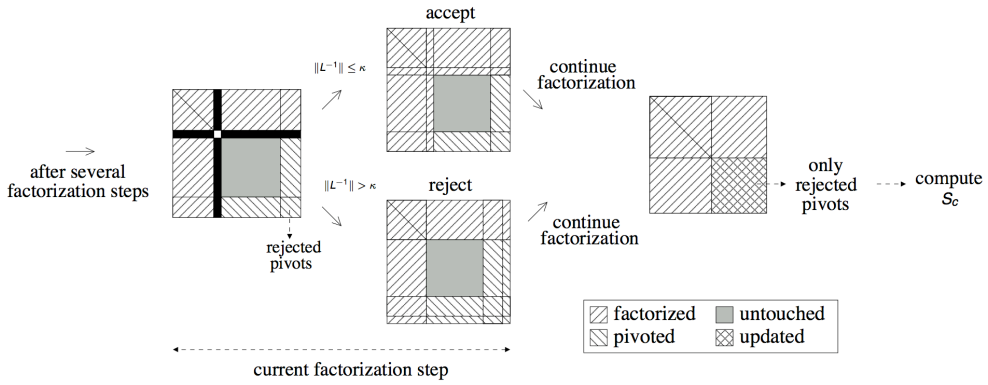

Figure 2: A step of the Crout variant of the preconditioner computation.

graphically depicted in Figure 2. Once $\hat{A}$ is completely processed in this manner, the trailing block only contains rejected pivots, and a partial IC factorization of a permuted matrix is computed:

$$\hat{P}^T \hat{A} \hat{P} \equiv \begin{bmatrix} B & F^T \\ F & C \end{bmatrix} = \begin{bmatrix} L_B & 0 \\ L_F & I \end{bmatrix} \begin{bmatrix} D_B & 0 \\ 0 & S_c \end{bmatrix} \begin{bmatrix} L_B^T & L_F^T \\ 0 & I \end{bmatrix} + E. \qquad (1)$$

Here, $\|L_B^{-1}\| \lessapprox \kappa$ and E contains the elements dropped during the IC factorization. Restarting the process with $A = S_c$, we obtain a multilevel approach.

Application of the preconditioner. For simplicity, let us next remove the subscripts in the corresponding operation (O5) of Figure 1: $z := M^{-1}r$. From (1), the preconditioner can be recursively defined, at level l, as

$$M_l = D^{-1} P \hat{P} \begin{bmatrix} L_B & 0 \\ L_F & I \end{bmatrix} \begin{bmatrix} D_B & 0 \\ 0 & M_{l+1} \end{bmatrix} \begin{bmatrix} L_B^T & L_F^T \\ 0 & I \end{bmatrix} \hat{P}^T P^T D^{-1}, \qquad (2)$$

where $M_0 = M$. Operating properly on the vectors,

$$\hat{P}^T P^T D^{-1} z = \hat{z} = \begin{bmatrix} \hat{z}_B \\ \hat{z}_C \end{bmatrix}, \; \hat{P}^T P^T D r = \hat{r} = \begin{bmatrix} \hat{r}_B \\ \hat{r}_C \end{bmatrix}, \qquad (3)$$

and applying $L_F = F L_B^{-T} D_B^{-1}$ (derived from (1)), we can expose the following computations to be performed at each level of the preconditioner [10]:

$$\begin{array}{lll} \text{Before:} & \hat{r} := \hat{P}^T P^T D r, & \text{Solve } L_B D_B L_B^T s_B = \hat{r}_B \text{ for } s_B, \\ & t_B := F s_B, & y_C := \hat{r}_B - t_B, \\ \text{Recursive step:} & \text{Solve } M_{l+1} \hat{z}_C = y_C \text{ for } \hat{z}_C, & \\ \text{After:} & \hat{t}_B := F^T \hat{z}_C, & \text{Solve } L_B D_B L_B^T \hat{s}_B = \hat{t}_B \text{ for } \hat{s}_B, \\ & \hat{z}_B := s_B - \hat{s}_B, & z := D P \hat{P} \hat{z}. \end{array} \qquad (4)$$

To conclude this subsection, 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.

2.2. Task parallel ILUPACK

Computation of the preconditioner. The task parallel version of this procedure exploits the connection between sparse matrices and adjacency graphs [1], extracting parallelism via nested dissection. Consider for example a graph-based symmetric reordering, defined by a permutation $\bar{P} \in \mathbb{R}^{n \times n}$, such that

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

Computing a partial IC factorizations of the two leading blocks, A_{00} and A_{11}, yields the following partial approximation of $\bar{P}^T A \bar{P}$

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

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 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 (5) 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:

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

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

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

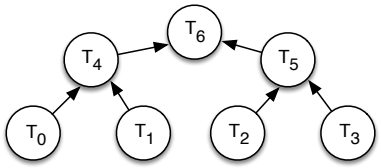

Figure 3: Dependency tree of the diagonal blocks. Task T_j is associated with block A_{jj}.

However, the construction of (6) involves a synchronization before the addition of these two blocks can be computed

$$E_2 \approx E_2^0 + E_2^1 \;\rightarrow\; S_{22} \approx S_{22}^0 + S_{22}^1. \tag{8}$$

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, by reordering and renaming the blocks properly, a block structure similar to (5) is obtained, from which four submatrices can be disassembled:

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

Figure 3 illustrates the dependency tree for the factorization of the diagonal blocks in (9). 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.

The task parallel version of ILUPACK partitions the original matrix into a number of decoupled blocks, and then delivers a partial IC factorization during the computation of (7), with some differences with respect to the sequential procedure. The main change is that the computation is restricted to the leading block, and therefore the rejected pivots are moved to the bottom-right corner of the leading block; see Figure 4. Although the recursive definition of the preconditioner, shown in (2), 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

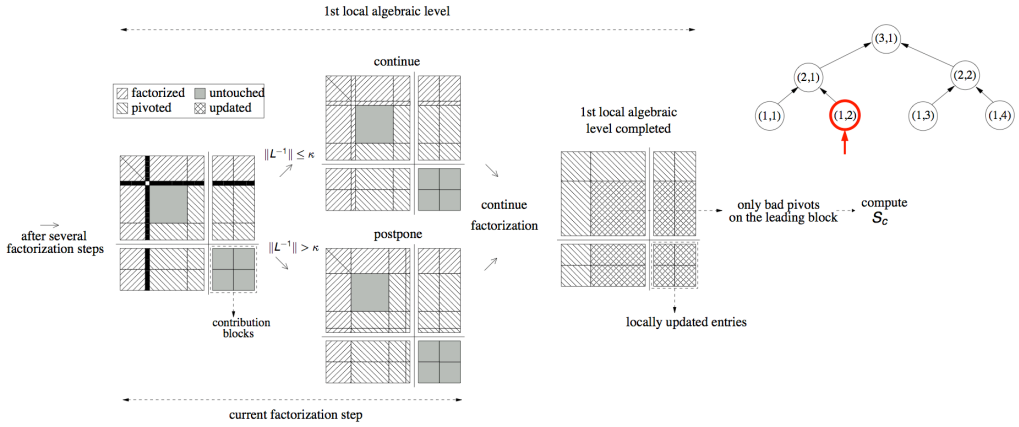

Figure 4: A step of the Crout variant of the parallel preconditioner computations.

recursion steps yielding distinct preconditioners, which nonetheless exhibit close numerical properties to that obtained with the sequential ILUPACK [7].

Application of the preconditioner. As the definition of the recursion is maintained, the operations to apply the preconditioner, in (4), remain valid. However, to complete the recursion step in the task parallel case, the preconditioner DAG has to be crossed two times per solve $z_{k+1} := M^{-1}r_{k+1}$ at each iteration of the PCG: once from bottom to top and a second time from top to bottom (with dependencies/arrows reversed in the DAG).

Other operations in the iteration. If the vectors involved in the PCG are partitioned conformally to matrix A, see e.g. (9), the SpMV and AXPY-like kernels only operate with the data associated with the leaves of the preconditioner DAGs. For example, for a matrix partitioned as in (9), the SpMV (O1) is decoupled into four matrix-vector products, with $\bar{A}_{00}$, $\bar{A}_{11}$, $\bar{A}_{22}$ and $\bar{A}_{33}$, which are accumulated as part of the subsequent DOT product (O2) [7]. In consequence, the computations with each one of these four leaves in SpMV is fully independent from the others. On the other hand, the DOT and the 2-norm require a reduction (synchronization) to obtain the result.

3. Tuning the Task Parallel ILUPACK on Many-core Architectures

3.1. OmpSs implementations

OmpSs is a task-based programming model that detects data dependencies between tasks at execution time, with the help of directionality clauses

embedded in the code as OpenMP-like directives. With this information, OmpSs implicitly generates a task graph during the execution that is simultaneously employed by the threads to exploit the task parallelism implicit to the operation via a dynamic out-of-order but dependency-aware schedule.

3.1.1. Exploiting nested parallelism

The operations that appear in the iterative PCG solve (`while` loop in Figure 1) define a partial order which enforces an almost strict serial execution. Specifically, at the $(k+1)$-th iteration

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

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, as described, for example, at the end of the previous section for the SpMV involving (9).

Handling the dependencies is easy at the task/subtask levels but rapidly becomes a burden for the OmpSs runtime when the number of cores in the target architecture is large. This scenario asks for a high number of (sub)tasks which, in ILUPACK, necessarily exhibit a small computational cost except for the operations involving the leaf nodes of the preconditioner DAG. In [10] we increased the granularity of the subtasks by modifying the code to *merge* three pairs of operations in the PCG solve into a single "group" of subtasks each: O1+O2, O3+O4 and O6+O8; see Figure 1. For example, the SpMV+dot in O1+O2, applied to (9), are combined by merging each one of the small matrix-vector products $\bar{w}_i := \bar{A}_{ii} d_k$, $i = 0, \ldots, 3$, with the reduction of the corresponding elements of O2. Additionally, the ordered execution of the groups was controlled by inserting explicit barriers (`#pragma omp barrier`) between O1+O2, O3+O4, O5, O6+O8 and O7.

In the new implementation, we eliminate the explicit barriers and instead rely on OmpSs to elegantly deal with the nested parallelism exhibited by the task/subtask dependencies. Concretely, the nested variant defines O1+O2, O3+O4, O5, O7 and O6+O8 as five coarse-grain OmpSs tasks (via `#pragma omp task`) and off-loads the complete detection and control of the dependencies to the OmpSs runtime. In addition, this version also divides these five macro-operations into fine-grain subtasks, and merges pairs of them as described above. In order to illustrate this, consider for example

O1+O2, consisting of the SpMV $w_k := Ad_k$ and the DOT $\rho_k := \beta_k / d_k^T w_k$. The code that performs this operation is annotated as follows:

```
1  // SpMV_DOT computes w_k := A * d_k and  rho_k := beta_k / (d_k^T * w_k)
2  // Coarse-grain task
3  #pragma omp task input (n, beta, d[0:n-1]) output (w[0:n-1], rho)
4  {
5    SpMV_DOT(int *n, double *beta,
6             double d[], double w[], double *rho) {
7      // Initialization code ...
8      for (id_task = 0; id_task < num_leaves_in_DAG; id_task++) {
9        // Fine-grain (sub)task
10       #pragma omp task
11       {
12         SpMV_DOT_LEAF(int *task_id, int *n, double *beta,
13                       double d[], double w[], double *rho) {
14           // Merged SpMV and DOT operating with leaf task_id ...
15         }
16       }
17     }
18     // Termination code ...
19   }
20 }
```

For simplicity, we do not illustrate how to deal with the reduction on ρ_k (variable `rho`) in the previous excerpt of code.

3.1.2. Mapping threads to cores on the Intel Xeon Phi

The Intel Xeon Phi supports up to 4 hardware threads per physical core, while our task parallel approach spawns one OmpSs thread per leave in the preconditioner DAG. A critical aspect in this platform is how to bind the OmpSs threads to the hardware threads/cores in order to distribute the workload. The mapping is controlled using the NANOS[6] runtime environment variable `NX_ARGS`, passing the appropriate values via arguments `--binding_stride`, `--binding_start` and `--smp_workers`. Specifically, the first argument governs how many hardware threads are to be skipped between the mapping of two consecutive OmpSs threads; the second identifies the starting point (first hardware thread) for a strided round-robin mapping; and the third argument specifies the total number of OmpSs threads. Thus, for example, by setting `--binding_stride`=1 we completely populate a core with 4 OmpSs threads before mapping threads to a new core. On the other extreme, `--binding_stride`=4 populates all cores with a single, two,... OmpSs thread(s) before assigning a second, third,... thread to them.

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

3.1.3. NUMA-aware execution on the AMD server

In order to attain high performance on the four-socket target AMD server, it is important to accommodate a NUMA-aware execution. This is achieved in our implementation via the use of the NANOS 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. The following fragment of code illustrates how this is achieved in the merged code for O1+O2:

```
1   for (id_task = 0; id_task < num_leaves_in_DAG; id_task++) {
2     // Fine-grain (sub)task
3     socket = preconditioner_socket[task_id];
4     nanos_current_socket(socket);
5     #pragma omp task
6     {
7       SpMV_DOT_LEAF(int *task_id, int *n, double *beta,
8                     double d[], double w[], double *rho) {
9         // Merged SpMV and DOT operating with leaf task_id ...
10      }
11    }
12  }
```

This strategy ensures that, during the PCG iteration, a task is always executed on (any core of) the same socket that computed the corresponding task during the computation of the preconditioner (recorded into array `preconditioner_socket`) using NANOS routine `nanos_current_socket()`.

3.2. MPI implementations

For this particular work, we ported the task-parallel MPI implementation of ILUPACK described in [8] to the Intel Xeon Phi accelerator and the AMD server. The MPI implementation also exploits the task concurrency explicitly exposed by the preconditioner DAG during the calculation of the preconditioner and the subsequent PCG iteration, but employs MPI ranks (i.e., processes) instead of the threads leveraged in the OmpSs version. A second major difference is that, in the MPI implementation, the tasks are mapped to the MPI ranks *a priori*, that is, before the execution commences. The execution with the MPI implementation is thus the result of a static schedule (static mapping of tasks to MPI ranks) instead of a dynamic one as occurs with the OmpSs implementation.

To distribute the MPI ranks among the processor cores of the Intel Xeon Phi, we include the options:

```
-genv I\_MPI\_PIN\_MODE=lib \
-genv I\_MPI\_PIN\_PROCESSOR\_LIST=\$mapping
```

in the `mpirun` invocation, with a list of cores in `$mapping` specifying the binding of ranks to cores.

To reduce inter-process communication, the original MPI implementation already ensures that the same MPI rank executes the operations associated with the "same" tasks of the preconditioner calculation and the PCG iteration. Compared with this, we note that we had to manually modify the OmpSs version to enforce a similar behaviour at the socket level in the NUMA-aware implementation for the AMD server.

4. Data Parallel ILUPACK

In [10], we introduced two data parallel variants of ILUPACK that off-load the computationally-intensive parts of the PCG iteration to a graphics accelerator. The most efficient variant among these two performs the complete application of the multilevel preconditioner, see (4), on the GPU, via *ad-hoc* kernels and the CUDA and CUSPARSE libraries [11]. In more detail, this implies that the residual r_{k+1} is transferred to the GPU when the preconditioner is to be applied; the complete 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. However, due to the complexity of ILUPACK, in both variants the SpMV and the vector operations of the PCG iteration were performed using the original code in the CPU.

For this work, we have enhanced our previous implementation to off-load the SpMV kernel also to the GPU. This requires that, at the beginning of each PCG iteration, vector 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 memory. Matrix A is transferred to the GPU memory before the PCG iteration commences and resides there, together with the preconditioner data, for the complete duration of the solve. The matrix is stored in CSR format [1] and the matrix-vector product is performed using the implementation of this kernel from CUSPARSE. In addition, we note that in general the vector operations contribute little to the computational cost of the solver. Therefore, these operations are performed in the CPU.

5. Experimental Results

5.1. Set up

All the experiments reported next were performed using IEEE 754 real double-precision arithmetic on three platforms:

- XEON PHI: A board with an Intel Xeon Phi 5110P co-processor attached to a server through a PCI-e Gen3 slot. (The tests on this board were ran in native mode and, therefore, the specifications of the server are irrelevant.) The accelerator board comprises 60 x86 cores running at 1,053 MHz and 8 Gbytes of GDDR5 RAM. The compiler and MPI implementation are part of Intel `icc` 13.1.3 20130607 (Intel MPI Library for Linux* OS, Version 4.1 Update 1 Build 20130507).
- OPTERON: A server with four AMD Opteron 6276 (16-core) processors at 2.1 GHz and 64 Gbytes of DDR3 RAM. The compiler is Intel `icc` 11.1 20100806 and the MPI implementation is OpenMPI 1.6.
- K20: An NVIDIA K20 board (2,496 cores at 706 MHz) with 6 Gbytes of GDDR5 RAM, connected via a PCI-e Gen3 slot to a server equipped with an Intel i3-3220 processor (2 cores at 3.4 GHz) and 16 Gbytes of DDR3 RAM. The compiler for this platform is `gcc` v4.4.7, and the codes are linked to CUDA 5.0 and CUSPARSE 5.0.

Other software included ILUPACK (2.4), the Mercurium C/C++ compiler/Nanox (releases 1.99.6/0.9a for XEON PHI and 1.99.1/0.8a for OPTERON) with support for OmpSs, and METIS (4.0.01) and ParMETIS (4.0.2) for the graph reorderings with the OmpSs and MPI implementations respectively.

For the analysis, we employed a s.p.d. linear system arising from the finite difference discretization of a 3D Laplace problem, with 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.

5.2. Task parallel ILUPACK

In this subsection we analyze the performance of different task parallel implementations of ILUPACK, based on MPI and OmpSs, on XEON PHI and OPTERON. For each platform, we employ the largest problem size that fits

	Matrix	Dimension n	#non-zeros	Density (%)
Laplace	A100	1,000,000	3,970,000	3.97E-6
	A126	2,000,376	7,953,876	1.99E-6
	A159	4,019,679	16,002,873	9.90E-7
	A171	5,000,211	19,913,121	7.96E-7
	A182	6,028,568	24,014,900	6.61E-7
	A191	6,967,781	27,762,041	5.72E-7
	A200	8,000,000	31,880,000	4.98E-7
	A318	32,157,432	128,326,356	1.24E-7

Table 1: Matrices employed in the experimental evaluation.

into its main memory. The experiments report the speed-up compared with the sequential implementation of ILUPACK, running on the corresponding platform (a single core of XEON PHI or OPTERON). Therefore, the results show the execution time of the parallel solver normalized with respect to the sequential version. In order to expose enough task concurrency, for the task parallel cases we partition the matrix into DAGs with one leaf per worker (either an OmpSs thread or an MPI rank), while the sequential version "solves" a DAG/matrix with a single task. We emphasize that the semantics of the task parallel version differ with the number of leaves (hereafter l) in the preconditioner DAG, and they are also different from the sequential semantics. However, we ensure that the solvers are comparable by stopping convergence when the same residual, of order `restol`, is attained.

On XEON PHI, the number of physical cores that are actually used in the task parallel executions, denoted by c, depends on the number of workers w that are spawned, between 1 and 32, and how many workers are mapped per core, w_c=1, 2 or 4 (inverse of the binding factor): $c = w/w_c$; see Table 2. On OPTERON, the number of cores is simply given by $c = w$, as one worker is at most mapped per core; see Table 2. On both platforms, $w = l$.

Table 3 reports the speed-ups attained by the OmpSs and MPI implementations of ILUPACK in XEON PHI for the benchmark A171. (Similar results were obtained for the smaller test cases A126 and A159.) The data comprised there reveals the following trends along different dimensions:

- *Iso-workers and Iso-DAGs (same w or column of the table).* Fixing the number of workers while we increase the level of "saturation" of the cores (i.e., raise w_c) has a clear negative effect on the OmpSs implementation and a slightly smaller one on the MPI one, for both the

#Workers	$w, l=$	1	2	4	8	16	32	64
XEON PHI	w_c=1	1	2	4	8	16	32	--
	w_c=2	1	1	2	4	8	16	--
	w_c=4	1	1	1	2	4	8	--
OPTERON		1	2	4	8	16	32	64

Table 2: Number of cores (c) for the experimental evaluation on XEON PHI and OPTERON. The cases with 64 workers were not evaluated on XEON PHI due to lack of enough memory for the MPI implementations.

		OmpSs					MPI				
#Workers	$w, l=$	2	4	8	16	32	2	4	8	16	32
Precond.	w_c=1	1.9	3.8	7.5	12.8	22.4	2.0	3.9	7.3	13.3	23.2
	w_c=2	1.4	2.9	5.5	9.8	16.9	1.4	2.9	5.7	10.6	18.3
	w_c=4	1.4	1.7	3.3	5.9	10.5	1.4	1.6	3.3	6.3	11.6
PCG solve	w_c=1	1.9	3.9	8.0	15.5	27.7	2.0	3.9	7.6	13.4	22.6
	w_c=2	1.5	3.1	6.2	11.9	18.0	1.6	3.1	5.9	10.9	19.4
	w_c=4	1.5	1.8	3.5	5.0	4.1*	1.6	1.5	2.8	5.1	11.4

Table 3: Speed-ups of the task parallel OmpSs and MPI implementations of the preconditioner computation and PCG solve in XEON PHI for matrix A171.

preconditioner computation and the PCG solve.

- *Iso-saturation (same w_c or row of the table).* Keeping constant the saturation, while increasing the number of workers w, implies a growth also in the number of physical cores (hardware resources) and, as could be expected, an increase of performance (except for one case in the OmpSs implementation of the PCG solve, marked with the superscript "$\star$").
- *Iso-cores (same c, cell color or diagonal of the table).* Fixing the number of cores to solve a problem, as the number of workers w grows, involves a proportional increase of the level of saturation of the cores. In other words, we maintain a constant volume of cores, while we increase the amount of workers and, simultaneously, the saturation of these hardware resources. This has a positive effect on both the OmpSs and MPI implementations of the preconditioner computation as well as the MPI implementations of the PCG solve. When the saturation level is w_c=4, the OmpSs implementation of the PCG solve suffers from the

increase to 32 workers.

- *Overall performance of OmpSs vs MPI.* In general, when the number of cores/workers is small, there appear slight performance differences in favor of the MPI implementation for the preconditioner computation and the OmpSs implementation for the PCG solve. On the other hand, as these values increase, OmpSs becomes the overall winner for the PCG solve while both implementations offer close performance for the preconditioner computation.

		OmpSs						MPI					
#Workers	$w, l=$	2	4	8	16	32	64	2	4	8	16	32	64
Precond.		2.0	4.0	6.3	10.5	14.5	22.9	2.0	3.9	7.3	12.8	19.5	23.9
PCG solve	NO	1.9	3.9	6.1	8.6	6.9	10.2	--	--	--	--	--	--
	NA	2.2	5.1	8.2	13.4	14.7	27.2	2.3	4.1	8.2	13.1	21.3	17.2

Table 4: Speed-ups of the task parallel OmpSs and MPI implementations of the preconditioner computation and PCG solve in OPTERON for matrix A318. NO and NA denote respectively the NUMA-oblivious and NUMA-aware implementations of the PCG solve.

Table 4 shows the speed-ups attained by the OmpSs and MPI implementations of ILUPACK in OPTERON. The number of parameters is now more reduced, which leads to a simpler analysis. First, the NUMA-aware OmpSs implementation of the PCG solve clearly outperforms its NUMA-oblivious counterpart, with the difference rapidly growing with the number of threads (and therefore cores). Also, as expected, increasing the number of workers yields higher speed-ups, as more resources are employed in the solution of the problem. We note here that this result demonstrates that the computational overhead intrinsic to partitioning into the matrix into a DAG consisting of more tasks is by far compensated by a superior amount of task concurrency to be exploited by a larger number of workers/cores. Finally, the OmpSs and MPI implementations deliver similar performance in the preconditioner computation when the number of cores is large, but OmpSs attains a much higher speed-up for the PCG solve when 64 workers are employed.

5.3. Data parallel ILUPACK

We next study the performance of the data parallel version of ILUPACK on K20. We remind that this implementation off-loads the SPMV and application of the preconditioner to the graphics accelerator, while all other

operations (including the preconditioner computation) are performed on the server's multicore CPU. Figure 5 reports the speed-up of the two GPU-accelerated operations as well as the global PCG solve with respect to the sequential version of ILUPACK running on a single Intel i3-3320 core of the server. For this particular collection of sparse problems, the GPU implementation of SpMV in CUSPARSE delivers a close-to-constant speed-up factor between 2.2× and 2.4×, independently of the matrix size. The application of the preconditioner offers better results, with a speed-up that is close to 4× for the largest matrix case. Combined, these two factors yield a speed-up of almost 3× for the PCG solve operating with the largest problem.

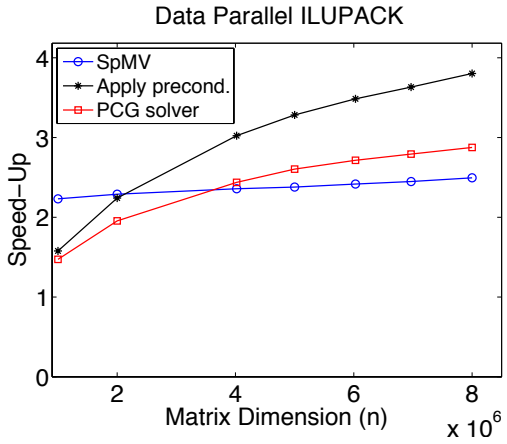

Figure 5: Speed-up of different parts of the PCG solve in K20 for several matrix cases; see Table 1.

5.4. Comparison of the solvers

We finally evaluate the numerical behaviour of the task and data parallel solvers, using a common matrix case (A171). For this purpose, we leverage the A-norm defined in [12], with the estimator in [13], as a measure of the numerical accuracy of the approximate solution x_j computed at the j-th iteration of the PCG solve: $\|x - x_j\|_A$, where x stands for the correct solution to the linear system. For the task parallel solvers, we use 32 workers/cores of XEON PHI and 64 on OPTERON.

Figure 6 relates the estimated residual of the solutions computed by the parallel solvers to the execution time. These results show that the numerical behaviour of the task parallel implementations based on OmpSs and MPI

is almost identical, which could be expected as they operate on DAGs with the same number of leaves. The small differences are due to the use of different versions of the METIS graph partitioning package to decompose the problem into tasks. The time difference between XEON PHI and OPTERON is explained by the use of 64 cores at 2.1 GHz in the latter vs only half of that number of cores at about half the frequency as well (1,053 MHz) in the former. Interestingly, when executed on K20 the data parallel solver, with a numerical behaviour equivalent to that of the sequential ILUPACK, shows an execution time between those of the task parallel implementations on XEON PHI and OPTERON.

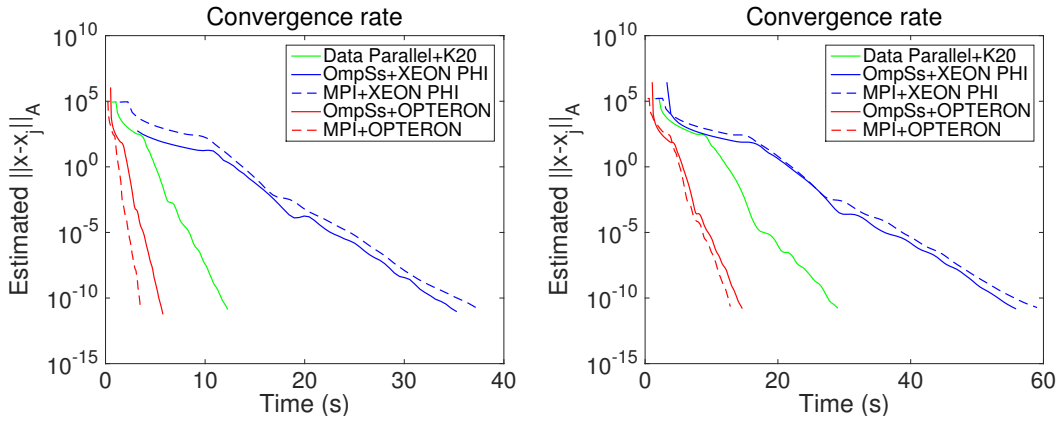

Figure 6: Convergence speed of the task and data parallel solvers for matrices A126 (left) and A171 (right).

6. Concluding Remarks

We have presented three parallel implementations of ILUPACK, based on the OmpSs runtime and the MPI message-passing libraries to exploit task parallelism on x86 many-core architectures, and using CUDA to exploit data parallelism on graphics processors. Compared with our previous work using this library, our OmpSs-based implementations employ nested parallelism to tackle task dependencies (instead of explicit barriers) at execution time, and introduce an architecture-aware implementation of the PCG solve for NUMA systems. Furthermore, our enhanced CUDA implementation performs the most expensive computational kernels of the PCG solve on the graphics accelerator (instead of only the preconditioner application).

Our experimental results on three many-core platforms (an Intel Xeon Phi accelerator, a 64-core NUMA AMD server, and an NVIDIA GPU with more than 4,000 cores) reveal that there exists ample task and data concurrency in the preconditioned solver embedded into ILUPACK, showing notable speed-ups in all these architectures. The direct comparison between our parallel implementations also exposes that, while they all can achieve similar residuals in the computed solution, from the point of view of performance the best option is to employ the MPI or OmpSs versions on the AMD server.

One advantage of x86-based architectures over GPUs is the existence of parallel programming tools such as OpenMP and MPI, more appealing to most programmers than the CUDA interface. However, our experience suggests that, for ILUPACK, the difficulties of the data-parallel programming model are partially overcome by the existence of data-parallel numerical libraries. Furthermore, the concurrency intrinsic to this application is easier to extract at a data-parallel level, favoring the implementation on a GPU. Exploiting the concurrency at the task-level for ILUPACK, on the other hand, is considerably more difficult, requiring a significant rewrite of the application to maintain semantics close to those of the sequential version.

Acknowledgements

The authors from the *Universitat Jaume I* were supported by the projects EU FP7 318793 (Exa2Green), TIN2011-23283 of the *Ministerio de Economía y Competitividad* (MINECO) and EU FEDER, and P11B2013-20 of the *Fundació Caixa Castelló-Bancaixa* and UJI. Rosa M. Badia was supported by project TIN2012-34557 of MINECO and EU FEDER, and by the Generalitat de Catalunya (contract 2009-SGR-980). María Barreda was supported by the FPU program of the *Ministerio de Educación, Cultura y Deporte*.

We thank Francisco D. Igual, from *Universidad Complutense de Madrid* (Spain), for his help with the Intel Xeon Phi.

References

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

[2] T. George, A. Gupta, V. Sarin, An empirical analysis of the performance of preconditioners for SPD systems, ACM Trans. Math. Softw. 38 (4) (2012) 24:1–24:30.

[3] O. Schenk, M. Bollhöfer, R. A. Römer, On large scale diagonalization techniques for the Anderson model of localization, SIAM Review 50 (2008) 91–112.

[4] O. Schenk, A. Wächter, M. Weiser, Inertia-revealing preconditioning for large-scale nonconvex constrained optimization, SIAM J. Sci. Comput. 31 (2) (2009) 939–960.

[5] A. George, Nested dissection of a regular finite element mesh, SIAM J. Numer. Anal. 10 (2) (1973) 345–363.

[6] T. Davis, Direct Methods for Sparse Linear Systems, SIAM, 2006.

[7] J. I. Aliaga, M. Bollhöfer, A. F. Martín, E. S. Quintana-Ortí, Exploiting thread-level parallelism in the iterative solution of sparse linear systems, Parallel Computing 37 (3) (2011) 183–202.

[8] J. I. Aliaga, M. Bollhöfer, A. F. Martín, E. S. Quintana-Ortí, Parallelization of multilevel ILU preconditioners on distributed-memory multiprocessors, in: Applied Parallel and Scientific Computing, LNCS, Vol. 7133, 2012, pp. 162–172.

[9] J. I. Aliaga, R. M. Badia, M. Barreda, M. Bollhöfer, E. S. Quintana-Ortí, Leveraging task-parallelism with OmpSs in ILUPACK's preconditioned cg method, in: 26th Int. Symp. on Computer Architecture and High Performance Computing (SBAC-PAD 2014), 2014, pp. 262–269.

[10] J. I. Aliaga, M. Bollhöfer, E. Dufrechou, P. Ezzatti, E. S. Quintana-Ortí, Leveraging data-parallelism in ILUPACK using graphics processors, in: 13th Int. Symp. Parallel and Distributed Computing (ISPDC 2014), 2014, pp. 119–126.

[11] CUDA Toolkit 5.5. CUSPARSE Library, NVIDIA Corporation, 2013, version 5.5.

[12] M. R. Hestenes, E. Stiefel, Methods of conjugate gradients for solving linear systems, J. Research Nat. Bur. Standards 49 (1952) 409–435.

[13] Z. Strakoš, P. Tichý, On error estimation in the Conjugate Gradient method and why it works in finite precision computations, Electronic Trans. Numer. Anal. 13 (2002) 56–80.