CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2016; **00**:1–21
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/cpe

Communication in Task-Parallel ILU-Preconditioned CG Solvers using MPI+OmpSs

José I. Aliaga[1], María Barreda[1]*, Goran Flegar[1], Matthias Bollhöfer[2], Enrique S. Quintana-Ortí[1]

[1] *Depto. de Ingeniería y Ciencia de Computadores, Universidad Jaume I (UJI), 12071–Castellón, Spain.*
{aliaga,mvaya,flegar,quintana}@uji.es
[2] *Institute of Computational Mathematics, TU Braunschweig, Braunschweig, Germany.*
m.bollhoefer@tu-bs.de.

SUMMARY

We target the parallel solution of sparse linear systems via iterative Krylov subspace-based methods enhanced with ILU-type preconditioners on clusters of multicore processors. In order to tackle large-scale problems, we develop task-parallel implementations of the classical iteration for the CG method, accelerated via ILUPACK and ILU(0) preconditioners, using MPI+OmpSs. In addition, we integrate several communication-avoiding (CA) strategies into the codes, including the butterfly communication scheme and Eijkhout's formulation of the CG method. For all these implementations, we analyze the communication patterns and perform a comparative analysis of their performance and scalability on a cluster consisting of 16 nodes, with 16 cores each. Copyright © 2016 John Wiley & Sons, Ltd.

Received . . .

KEY WORDS: Sparse linear systems; Krylov subspace methods; Incomplete LU (ILU) preconditioner; ILUPACK; task-parallelism; MPI; OmpSs

1. INTRODUCTION

The solution of linear systems is an ubiquitous computational problem appearing in fundamental numerical simulations as well as in recent methods for data analytics [1]. In order to tackle sparse instances of these linear algebra problems, ILUPACK[†] provides sequential implementations of several key Krylov subspace methods, such as CG, BiCG, SQMR and GMRES [2], enhanced with a sophisticated multilevel Incomplete LU (ILU) preconditioner [3].

The notable cost of computing and applying ILUPACK's preconditioner, in the framework of the CG method for large symmetric positive definite (s.p.d.) linear systems [2], motivated the design of a task-parallel procedure for clusters of multicore processors (see [4] and the references therein). For this type of computer architectures, this parallel version of ILUPACK exposes task-parallelism via nested dissection applied to the coefficient matrix of the linear system. As a consequence, it usually performs more floating-point operations (flops) than the original ILUPACK, with the overhead mildly growing with the number of tasks [5]. Furthermore, the parallel implementation calculates a preconditioner which differs from that computed by the sequential ILUPACK, yielding distinct convergence rates (though they are not necessarily slower).

*Correspondence to: María Barreda, Depto. de Ingeniería y Ciencia de Computadores, Universidad Jaume I (UJI), 12071–Castellón, Spain. mvaya@uji.es
[†]Incomplete LU decomposition PACKage. Available at http://ilupack.tu-bs.de.

In [6] we introduced a parallel implementation of ILUPACK's preconditioned CG (PCG) solver that combines the advantages of the programming models embedded in MPI and OmpSs [7, 8] to improve the performance of a pure MPI-based solution. Due to the classical formulation of the iterative PCG scheme and the use of an ILU-type preconditioner, the (pure-MPI and MPI+OmpSs) task-parallel variants feature a series of communication (and synchronization) points that impair their concurrency. In this work, we analyze the parallel performance of several task-parallel ILU-PCG solvers, explicitly exposing the communication patterns in their MPI+OmpSs implementation. Concretely, our work makes the following specific contributions:

- We develop task-parallel implementations of the classical iteration for the CG method, preconditioned with ILUPACK and ILU(0), that rely on a hybrid programming solution based on MPI and OmpSs.
- We integrate three communication-avoiding (CA) strategies into the classical ILUPACK-PCG and ILU(0)-PCG:
  - The butterfly transformation scheme for collective communication primitives [9, 10].
  - Eijkhout's formulation of the PCG method [11] that shifts all reduction operations into a single synchronization point per iteration.
  - A modification to merge and reduce the volume of tasks corresponding to multiple "communication-less" vector operations that diminishes the scheduling overhead.
- We discuss in detail the communication and synchronization patterns appearing in these 8 parallel implementations (two preconditioners $\times$ four formulations: classical+enhanced with 3 CA strategies).
- For all these implementations we perform a comparative analysis of their weak and strong scalability on a cluster consisting of 16 nodes, with two 8-core sockets per node. Our experiments show that, in order to offer a significant acceleration, the CA strategies will require additional optimization and/or an evaluation at a larger scale. Unfortunately, the current parallelization approach offers a limited degree of scalability. The primary reason is that, as the amount of computational resources grows, the additional concurrency which is explicitly exposed by partitioning the computational load (sparse matrix/adjacency graph) of the problem into further levels does not compensate the overhead that is introduced by the splitting.

The communication analysis performed in this work carries over to several alternative CA variants of the CG solver (see related work next), enhanced with an ILU-preconditioner such as ILU(0), ILU(p), ILUT, etc. [2], that rely on task-parallelism. On the other hand, the analysis of the numerical properties of these distinct preconditioners and CA variants has been performed elsewhere and it is out-of-scope for this paper. We instead focus on their parallel properties, communication patterns/costs, and synchronization overheads.

1.1. Related work

Task-parallelism is usually exploited by parallel direct solvers for sparse linear systems [12, 13, 14, 15, 16]. Our approach also leverages task-parallelism but, when applied to iterative ILU-preconditioned solvers, yields significantly different properties for the task dependency graph (TDG) associated with the problem and the distribution of the computational cost among the graph nodes [5]. These factors dictate the regular communication patterns yielding the parallel efficiency discussed in this work.

The cost of process/thread synchronization in parallel computers has resulted in a number of research efforts pursuing CA re-formulations of iterative solvers for linear systems. In general, these solutions trade off rounding errors, which may produce slower convergence to the solution, for higher parallel performance. A collection of CA variants of the CG method for s.p.d. linear systems are surveyed in [11, 17, 18]. Our work differs in that we target the solution of sparse linear systems using *task-parallel CG solvers* and CA variants of these, in combination with (simple) ILU(0) as well as (complex) ILUPACK preconditioners.

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

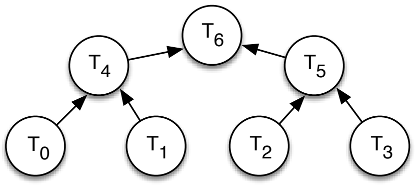

Figure 1. Partitioning of the coefficient matrix A (left) and task dependency tree for the diagonal blocks (right). Task T_j is in charge of processing the diagonal block A_{jj}. T_6 is associated with the separator obtained from the first application of nested dissection, and T_4, T_5 with those after the second one.

2. TASK-PARALLEL ILU-PCG SOLVERS

2.1. *ILUPACK*

Consider the linear system $Ax = b$, where A is the $n \times n$ sparse coefficient matrix, b is the right-hand side vector, and x is the sought-after solution, with both b, x of size n. For s.p.d. linear systems, ILUPACK's implementation of the CG method integrates an efficient preconditioner obtained from an incomplete LU factorization $A \approx LU = LL^T = M$. This method relies on dropping combined with pivoting to bound the norm of the inverse triangular factor L, yielding a numerical multilevel hierarchy of partial inverse-based approximations [19, 20]. (For s.p.d., ILUPACK specifically computes the factorization $A = L(DL^T) = LU$, where L is a lower triangular factor and D is a diagonal matrix. For simplicity, we will skip the diagonal scaling from our discussion.)

From the perspective of a parallel implementation, the computation and application of the preconditioner are two challenging operations that, furthermore, concentrate most of the computational cost of the solve. We next review the approach to extract task-parallelism for these operations; for details on the numerical foundations of the method, see e.g. [5].

Exposing task-parallelism The parallel version of ILUPACK recursively applies nested dissection in order to identify a collection of independent blocks in A via row/column permutations [5]. This process defines a hierarchy of subgraphs and separators, which then dictates the order in which the blocks have to be factorized during the preconditioner computation and the triangular systems need to be solved during the subsequent PCG iterations. Concretely, the hierarchy defines a TDG with a binary-tree form, where the subgraphs occupy the leaves of the graph/tree and the separators correspond to the internal nodes. Figure 1 (left) illustrates the effect of applying two nested dissection steps on a sparse matrix, yielding 4 subgraphs and 3 separators. Note that, due to the symmetry, $A_{ij} = A_{ji}$. Figure 1 (right) shows the corresponding TDG of the transformed matrix. The edges of that graph specify the dependencies between the diagonal blocks (tasks) for the preconditioner computation (i.e., the approximate matrix factorization).

Computing the preconditioner In order to increase the degree of concurrency during the computation of the preconditioner, the submatrices for the graph in Figure 1 are decomposed as

$$\left[\begin{array}{c|cc}
A_{00} & A_{04} & A_{06} \\
\hline
A_{40} & A_{44}^{0} & A_{46}^{0} \\
A_{60} & A_{64}^{0} & A_{66}^{0}
\end{array}\right],
\left[\begin{array}{c|cc}
A_{11} & A_{14} & A_{16} \\
\hline
A_{41} & A_{44}^{1} & A_{46}^{1} \\
A_{61} & A_{64}^{1} & A_{66}^{1}
\end{array}\right],
\left[\begin{array}{c|cc}
A_{22} & A_{25} & A_{26} \\
\hline
A_{52} & A_{55}^{2} & A_{56}^{2} \\
A_{62} & A_{65}^{2} & A_{66}^{2}
\end{array}\right],
\left[\begin{array}{c|cc}
A_{33} & A_{35} & A_{36} \\
\hline
A_{53} & A_{55}^{3} & A_{56}^{3} \\
A_{63} & A_{65}^{3} & A_{66}^{3}
\end{array}\right], \qquad (1)$$

where

$$\begin{array}{lll}
A_{44} = A_{44}^{0} + A_{44}^{1}, & A_{55} = A_{55}^{2} + A_{55}^{3}, & A_{66} = A_{66}^{0} + A_{66}^{1} + A_{66}^{2} + A_{66}^{3}, \\
A_{46} = A_{46}^{0} + A_{46}^{1}, & A_{56} = A_{56}^{2} + A_{56}^{3}, & \\
A_{64} = A_{64}^{0} + A_{46}^{1}, & A_{65} = A_{65}^{2} + A_{65}^{3}. &
\end{array} \qquad (2)$$

After this reorganization, the factorizations of the leading blocks of these four submatrices can proceed in parallel, while the modified blocks A_{ij}^{0-3} are needed to solve the dependencies of the ancestor tasks. This process continues traversing the dependency tree, until the root task factorizes its local submatrix; see [5] for details.

Applying the preconditioner The subsequent application of the preconditioner $M = LL^T$ during the iterative PCG solve requires the solution of two triangular systems, for the lower triangular factor L and its transpose, per iteration. Consider, for example, the solution of the linear system $Ly = c$, with L featuring the same block structure as the lower triangular part of A in Figure 1 (left); and assume $y = [y_0, y_1, y_2, y_3 \,|\, y_4, y_5 \,|\, y_6]$, $c = [c_0, c_1, c_2, c_3 \,|\, c_4, c_5 \,|\, c_6]$ denote partitions conformal to that of A in the same figure. In this case, to unleash a parallel execution, the triangular solve $Ly = c$ is decomposed as

$$\begin{array}{llll}
\left[\begin{array}{c} L_{00} \\ \hline L_{40} \\ \hline L_{60} \end{array}\right] y_0 = \left[\begin{array}{c} c_0 \\ \hline c_4^0 \\ \hline c_6^0 \end{array}\right], &
\left[\begin{array}{c|c} L_{11} & \\ \hline L_{41} & L_{44} \\ \hline L_{61} & L_{64} \end{array}\right] \left[\begin{array}{c} y_1 \\ \hline y_4 \end{array}\right] = \left[\begin{array}{c} c_1 \\ \hline c_4^1 \\ \hline c_6^1 \end{array}\right], \\
\left[\begin{array}{c} L_{22} \\ \hline L_{52} \\ \hline L_{62} \end{array}\right] y_2 = \left[\begin{array}{c} c_2 \\ \hline c_5^2 \\ \hline c_6^2 \end{array}\right], &
\left[\begin{array}{c|c|c} L_{33} & & \\ \hline L_{53} & L_{55} & \\ \hline L_{63} & L_{65} & L_{66} \end{array}\right] \left[\begin{array}{c} y_3 \\ \hline y_5 \\ \hline y_6 \end{array}\right] = \left[\begin{array}{c} c_3 \\ \hline c_5^3 \\ \hline c_6^3 \end{array}\right],
\end{array} \tag{3}$$

where

$$c_4 = c_4^0 + c_4^1, \quad c_5 = c_5^2 + c_5^3, \quad c_6 = c_6^0 + c_6^1 + c_6^2 + c_6^3. \tag{4}$$

In (3), the leading blocks $L_{00}, \ldots, L_{33}$ define four lower triangular linear systems that can be solved concurrently for $y_0, \ldots, y_3$. After that, forward substitution yields updates values for c_4, c_5 which are used in the solves with L_{44}, L_{55} for y_4, y_5; and repeating the process we obtain an updated value c_6 to be used in the final solve with L_{66} for y_6. The TDG for the lower triangular system therefore presents the same binary-tree structure identified via nested dissection; see Figure 1 (right). For the triangular solve involving L^T though, the dependencies are reversed, pointing down from the root to the leaves.

The key to high performance during a task-parallel execution of ILUPACK's CG solver lies in concentrating most of the computational work on the leaves of the TDG so that the internal nodes contribute little to the global computation from the perspective of the operation count [5]. The reason is that, as one proceeds upwards in the *binary-tree* TDG, the amount of concurrency is reduced. This lack of concurrency can be compensated if most of the computational work is concentrated in the leaves while the internal nodes only represent a small fraction of the cost.

2.2. *Other ILU-PCG solvers*

The previous approach to formulate a task-parallel execution of ILUPACK's PCG solver also applies to other ILU-type solvers in general and ILU(0) [2] in particular. As a result, the communication patterns that we identify in the next two sections apply to a generic task-parallel ILU-PCG solver.

3. CLASSICAL TASK-PARALLEL ILU-PCG SOLVERS

3.1. *The iterative PCG solve*

We focus the analysis in this section on the iterative PCG solve, as this phase is usually more expensive than the computation of the preconditioner. Figure 2 offers an algorithm description of the classical iterative PCG scheme. The loop body consists of a sparse matrix-vector product (SPMV, S1), the preconditioner application (equivalent to two triangular solves, S5), three DOT products (S2, S6 and S9), three AXPY(-like) operations (S3, S4 and S8), and a few scalar operations.

```
Compute preconditioner for $A \approx LL^T = M$
Set starting guess $x^{(0)}$
Initialize $z^{(0)}, d^{(0)}, \beta^{(0)}, \tau^{(0)}, k := 0$

$r^{(0)} := b - Ax^{(0)}$
$\tau^0 := < r^{(0)}, r^{(0)} >$
while ($\tau^k > \tau_{\max}$)

  Step     Operation                                                   Kernel
  S1 :     $w^{(k)} := Ad^{(k)}$                                       SpMV
  S2 :     $\rho^{(k)} := \beta^{(k)} / < d^{(k)}, w^{(k)} >$          DOT product
  S3 :     $x^{(k+1)} := x^{(k)} + \rho^{(k)} d^{(k)}$                 AXPY
  S4 :     $r^{(k+1)} := r^{(k)} - \rho^{(k)} w^{(k)}$                 AXPY
  S5 :     $z^{(k+1)} := M^{-1} r^{(k+1)}$                             Apply preconditioner:
    S5.1 :   $y := L^{-1} r^{(k+1)}$                                     Lower triangular solve
    S5.2 :   $z^{(k+1)} := L^{-T} y$                                     Upper triangular solve
  S6 :     $\beta^{(k+1)} := < z^{(k+1)}, r^{(k+1)} >$                 DOT product
  S7 :     $\alpha^{(k)} := \beta^{(k+1)} / \beta^k$                   Scalar operation
  S8 :     $d^{(k+1)} := \alpha^k d^{(k)} + z^{(k+1)}$                 AXPY-like
  S9 :     $\tau^{(k+1)} := < r^{(k+1)}, r^{(k+1)} >$                  DOT product
           $k := k + 1$
end while
```

Figure 2. Classical formulation of the iterative ILU-PCG solver with annotated computational kernels. The threshold $\tau_{\max}$ is an upper bound on the relative residual for the computed approximation to the solution. In the notation, $< \cdot, \cdot >$ computes the DOT (inner) product of its vector arguments.

3.2. *Communications in the classical PCG*

For clarity, in the remainder of the paper we will drop the superindices that denote the iteration count in the variable names. Thus, for example, $x^{(k)}$ becomes x, where the latter basically stands for the storage space employed to keep the sequence of approximations $x^{(0)}, x^{(1)}, x^{(2)}, \ldots$ computed during the iteration. For the communication analysis, we will first explore the following simplified scenario:

1. The parallel platform consists of two processes, P0 and P1, where the term "process" is generic and refers to either MPI ranks or OmpSs threads. Without loss of generality, we will assume a message-passing communication mechanism operating in a distributed-memory setting.
2. The target TDG results from a single dissection step and consists only of two leaves connected by a root node. Correspondingly, the coefficient matrix A is decoupled into 3×3 blocks:

$$\left[\begin{array}{cc|c} A_{00} & & A_{02} \\ & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right],$$

and is disassembled as

$$A^0 = \left[\begin{array}{c|c} A_{00} & A_{02} \\ \hline A_{20} & A_{22}^0 \end{array}\right], \quad A^1 = \left[\begin{array}{c|c} A_{11} & A_{12} \\ \hline A_{21} & A_{22}^1 \end{array}\right],$$

where $A_{22} = A_{22}^0 + A_{22}^1$ is not built explicitly but instead maintained as two separate addends.

State Hereafter, we will say that an operand like this is in *inconsistent* state, and we will distinguish this property by adding a hat to its variable name, as in $\hat{A}$. This implies that recovering a consistent state requires some arithmetic (e.g., $A_{22} = A_{22}^0 + A_{22}^1$).

Storage The data for an inconsistent matrix (or vector) is *always distributed* between the processes. For the particular case of matrix A, P0 and P1 respectively store A^0 and A^1.

This means that some blocks of the original matrix are kept by a single process only (e.g., A_{00} and A_{20} in P0), while others can be only retrieved after some communication (e.g., forming A_{22} requires that either A_{22}^0 or A_{22}^1 are transferred from one process to the other).

3. All vectors are partitioned conformally with $\hat{A}$. For example, $x = [x_0, x_1 \,|\, x_2]$.
4. The residual $\hat{r}$ and $\hat{w}$ are maintained in inconsistent state. For example, the subvectors $r^0 = [r_0 \,|\, r_2^0]$ and $r^1 = [r_0 \,|\, r_2^1]$ (with $r_2 = r_2^0 + r_2^1$) are respectively stored in P0 and P1. This is conformal with the inconsistent state (and storage) of $\hat{A}$.
5. The solution vector x and the recurrence vectors z, d maintain consistent information; i.e., no arithmetic is required to recover their complete state. In most cases, the data entries of consistent variables are *partially replicated* so that, e.g., P0 and P1 store $x^0 = [x_0 \,|\, x_2]$ and $x^1 = [x_1 \,|\, x_2]$, respectively. Thus, both processes keep their own copy of the "common" part x_2, while each process stores its "local" part with x_0 in P0 and x_1 in P1.
6. At each iteration, during application of the preconditioner, a consistent/partially replicated copy is created for $\hat{r}$.
7. The scalars $\alpha, \beta, \rho, \tau$ are globally replicated.

Sparse matrix-vector product Let us analyze next each one of the operations comprised by the loop body of the PCG iteration, starting with the SPMV (S1). The inputs to this operation are $\hat{A}$ (inconsistent) and d (consistent/partially replicated). Therefore, the following computations can proceed concurrently:

$$\text{P0}: \left[\begin{array}{c} w_0 \\ \hline w_2^0 \end{array}\right] := A^0 \left[\begin{array}{c} d_0 \\ \hline d_2 \end{array}\right] = \left[\begin{array}{c|c} A_{00} & A_{02} \\ \hline A_{20} & A_{22}^0 \end{array}\right] \left[\begin{array}{c} d_0 \\ \hline d_2 \end{array}\right],$$

$$\text{P1}: \left[\begin{array}{c} w_1 \\ \hline w_2^1 \end{array}\right] := A^1 \left[\begin{array}{c} d_1 \\ \hline d_2 \end{array}\right] = \left[\begin{array}{c|c} A_{11} & A_{12} \\ \hline A_{21} & A_{22}^1 \end{array}\right] \left[\begin{array}{c} d_1 \\ \hline d_2 \end{array}\right].$$

Thus, by keeping the result inconsistent (in our notation, $\hat{w} := \hat{A}d$,) *this operation requires no communication*. This is a direct consequence of the states of the input/output operands, defined in the introductory part of this section.

DOT products The next operation is the DOT product S2 between d and $\hat{w}$. Here, it is easy to see that the following partial results can be computed concurrently:

$$\begin{array}{ll} \text{P0}: & \sigma^0 := < [d_0, d_2], [w_0, w_2^0] >, \\ \text{P1}: & \sigma^1 := < [d_1, d_2], [w_1, w_2^1] >, \end{array}$$

after which, P0 and P1 exchange σ^0 and σ^1, and then compute the globally replicated scalar $\rho := \beta/(\sigma^0 + \sigma^1)$. The same idea applies to the DOT products in S6 and S9.

AXPY(-like) vector updates Consider now the AXPY operation in S3, which involves vectors x, d and the scalar ρ. Both processes replicate part of this computation (concretely, that involving x_2, d_2), to obtain the result without any communication:

$$\text{P0}: \left[\begin{array}{c} x_0 \\ \hline x_2 \end{array}\right] := \left[\begin{array}{c} x_0 \\ \hline x_2 \end{array}\right] + \rho \left[\begin{array}{c} d_0 \\ \hline d_2 \end{array}\right],$$

$$\text{P1}: \left[\begin{array}{c} x_1 \\ \hline x_2 \end{array}\right] := \left[\begin{array}{c} x_1 \\ \hline x_2 \end{array}\right] + \rho \left[\begin{array}{c} d_1 \\ \hline d_2 \end{array}\right].$$

The same applies to S8 if we require that z is maintained consistent/partially replicated during the iteration. On the other hand, for the AXPY in S4, the input vectors $\hat{r}, \hat{w}$ are inconsistent. This means that the updated result can be maintained in the same state. Moreover, it can be computed concurrently, without any communication, as:

$$\mathsf{P0}: \left[\begin{array}{c} r_0 \\ \hline r_2^0 \end{array}\right] := \left[\begin{array}{c} r_0 \\ \hline r_2^0 \end{array}\right] - \rho\left[\begin{array}{c} w_0 \\ \hline w_2^0 \end{array}\right],$$

$$\mathsf{P1}: \left[\begin{array}{c} r_1 \\ \hline r_2^1 \end{array}\right] := \left[\begin{array}{c} r_1 \\ \hline r_2^1 \end{array}\right] - \rho\left[\begin{array}{c} w_1 \\ \hline w_2^1 \end{array}\right].$$

Lower triangular solve Let us partition the triangular factor resulting from the ILU factorization $A \approx LL^T = M$ as

$$\left[\begin{array}{cc|c} L_{00} & & \\ & L_{11} & \\ \hline L_{20} & L_{21} & L_{22} \end{array}\right].$$

This factor is maintained in consistent state while its data is disassembled and *(fully) distributed* among the processes with

$$L^0 = \left[\begin{array}{c} L_{00} \\ \hline L_{20} \end{array}\right] \text{ and } L^1 = \left[\begin{array}{c|c} L_{11} & \\ \hline L_{21} & L_{22} \end{array}\right]$$

stored in P0 and P1, respectively. Thus, the processes do not keep duplicate copies of any block of the triangular factor, but the information in it is consistent (no arithmetic is required to recover any block of L). Hereafter, we distinguish this by adding a bar to the name of such variables, as in $\bar{L}$.

Now, the lower triangular system in S5.1 boils down to

$$\left[\begin{array}{cc|c} L_{00} & & \\ & L_{11} & \\ \hline L_{20} & L_{21} & L_{22} \end{array}\right] \left[\begin{array}{c} y_0 \\ y_1 \\ \hline y_2 \end{array}\right] = \left[\begin{array}{c} r_0 \\ r_1 \\ \hline r_2^0 + r_2^1 \end{array}\right],$$

and we can then solve it by first computing in parallel

$$\begin{array}{ll} \mathsf{P0}: & y_0 := L_{00}^{-1} r_0, \quad t_2^0 := r_2^0 - L_{20} y_0, \\ \mathsf{P1}: & y_1 := L_{11}^{-1} r_1, \quad t_2^1 := r_2^1 - L_{21} y_2. \end{array}$$

After this, P0 sends t^0 to P1 (the owner of L_{22}) which then proceeds to compute:

$$\mathsf{P1}: \quad t_2 := t_2^0 + t_2^1, \quad y_2 := L_{22}^{-1} t_2,$$

while P0 remains idle. We note that the solution shares the properties of $\bar{L}$: it is consistent but distributed conformally, with y_0 in P0 and y_1, y_2 in P1. We will therefore refer to it as $\bar{y}$.

Upper triangular solve The solve in S5.2, $\bar{L}^T z = \bar{y}$, can be partitioned as

$$\left[\begin{array}{cc|c} L_{00}^T & & L_{20}^T \\ & L_{11}^T & L_{21}^T \\ \hline & & L_{22}^T \end{array}\right] \left[\begin{array}{c} z_0 \\ z_1 \\ \hline z_2 \end{array}\right] = \left[\begin{array}{c} y_0 \\ y_1 \\ \hline y_2 \end{array}\right],$$

where we will leverage that the triangular matrix $\bar{L}$ and the right-hand side $\bar{y}$ are both consistent and distributed. In this case, we commence by solving a triangular system in P1:

$$\mathsf{P1}: \quad z_2 := L_{22}^{-T} y_2,$$

after which, this process sends the result z_2 to P0. Next, in parallel, we obtain

$$\begin{array}{ll} \mathsf{P0}: & y_0 := y_0 - L_{20}^T z_2, \quad z_0 := L_{00}^{-T} y_0, \\ \mathsf{P1}: & y_1 := y_1 - L_{21}^T z_2, \quad z_1 := L_{11}^{-T} y_1. \end{array}$$

Here, we note that $z^0 = [z_0 | z_2]$ are stored in P0, while P1 contains a copy of $z^1 = [z_1 | z_2]$. This corresponds to a consistent/partially replicated result.

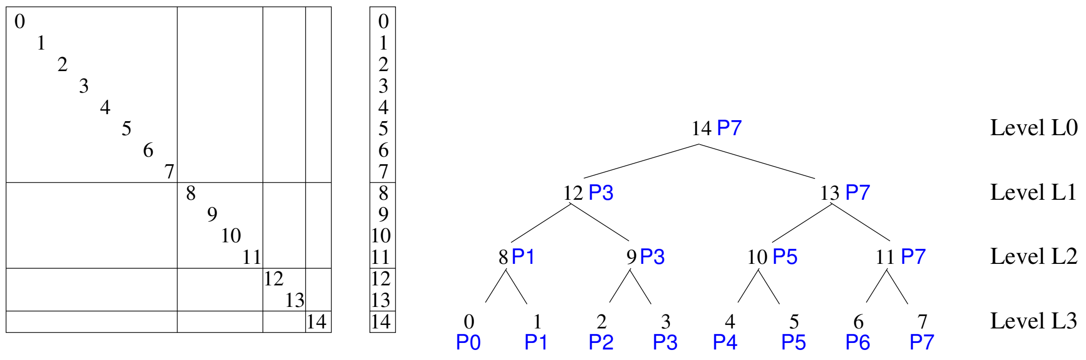

Figure 3. Left: Partitioning of the inconsistent matrix $\hat{A}$ using three nested dissection steps and conformal partitioning of the inconsistent vector $\hat{r}$. (For clarity, the variable names for the matrix blocks and subvectors, and some of the nonzero blocks in $\hat{A}$ are not shown.) Right: Corresponding TDG consisting of 4 levels (Level 0–Level 3) and mapping of nodes/tasks to processes (P0–P7).

Level	Send				Compute	
3	P0→P1:	r_{14}^{0} r_{12}^{0} r_{8}^{0}	(P1→P0:	r_{14}^{1} r_{12}^{1} r_{8}^{1})	In P1 (and P0):	$r_{14}^{0-1} := r_{14}^{0} + r_{14}^{1}$ $r_{12}^{0-1} := r_{12}^{0} + r_{12}^{1}$ $r_{8} = r_{8}^{0-1} := r_{8}^{0} + r_{8}^{1}$
	P2→P3:	r_{14}^{2} r_{12}^{2} r_{9}^{2}	(P3→P2:	r_{14}^{3} r_{12}^{3} r_{9}^{3})	In P3 (and P2):	$r_{14}^{2-3} := r_{14}^{2} + r_{14}^{3}$ $r_{12}^{2-3} := r_{12}^{2} + r_{12}^{3}$ $r_{9} = r_{9}^{2-3} := r_{9}^{2} + r_{9}^{3}$
	P4→P5:	r_{14}^{4} r_{13}^{4} r_{10}^{4}	(P5→P4:	r_{14}^{5} r_{13}^{5} r_{10}^{5})	In P5 (and P4):	$r_{14}^{4-5} := r_{14}^{4} + r_{14}^{5}$ $r_{13}^{4-5} := r_{13}^{4} + r_{13}^{5}$ $r_{10} = r_{10}^{4-5} := r_{10}^{4} + r_{10}^{5}$
	P6→P7:	r_{14}^{6} r_{13}^{6} r_{11}^{6}	(P7→P6:	r_{14}^{7} r_{13}^{7} r_{11}^{7})	In P7 (and P6):	$r_{14}^{6-7} := r_{14}^{6} + r_{14}^{7}$ $r_{13}^{6-7} := r_{13}^{6} + r_{13}^{7}$ $r_{11} = r_{11}^{6-7} := r_{11}^{6} + r_{11}^{7}$
2	P1→P3 (and P0→P2):	r_{14}^{0-1} r_{12}^{0-1}	(P3→P1, P2→P0:	r_{14}^{2-3} r_{12}^{2-3})	In P3 (and P0–P2):	$r_{14}^{0-3} := r_{14}^{0-1} + r_{14}^{2-3}$ $r_{12} = r_{12}^{0-3} := r_{12}^{0-1} + r_{12}^{2-3}$
	P5→P7 (and P4→P6):	r_{14}^{4-5} r_{13}^{4-5}	(P7→P5, P6→P4:	r_{14}^{6-7} r_{13}^{6-7})	In P7 (and P4–P6):	$r_{14}^{4-7} := r_{14}^{4-5} + r_{14}^{6-7}$ $r_{13} = r_{13}^{4-7} := r_{13}^{4-5} + r_{13}^{6-7}$
1	P3→P7 (and P0→P4, P1→P5, P2→P6):	r_{14}^{0-3}	(P7→P3, P4→P0, P5→P1, P6→P2:	r_{14}^{4-7})	In P7 (and P0–P6):	$r_{14} = r_{14}^{0-7} := r_{14}^{0-3} + r_{14}^{4-7}$

Figure 4. From top to bottom, sequence of messages and computations required to carry out the transformation $\hat{r} \rightarrow \bar{r}$. Inside parenthesis: additional messages and operations to perform the transformation $\hat{r} \rightarrow r$ in one step.

Transformation of the residual In preparation for the DOT product in S9, the application of the preconditioner creates a consistent/partially replicated version of $\hat{r}$. For this purpose, during the lower triangular solve, P0 sends r_2^0 (together with the message for t_2^0) to P1. This information is then used by P1 to obtain $r_2 := r_2^0 + r_2^1$, building a consistent but distributed copy of the residual vector. In our notation, this is equivalent to the transformation $\hat{r} \rightarrow \bar{r}$. Next, during the subsequent upper triangular solve, (and together with the message containing z_2,) P1 sends a copy of r_2 to P0, so that upon completion the platform stores the sough-after consistent/partially replicated residual vector after the preconditioner application: $\bar{r} \rightarrow r$.

Generalization Let us consider now a more general scenario, consisting in a parallel platform with 8 processes, P0–P7, and a TDG resulting from three nested dissection steps, composed of 8 subgraphs (leaf nodes 0–7) and 7 separators (internal nodes 8–14); see Figure 3.

The inconsistent state of $\hat{r}$ implies that the subvectors $r_8, r_9, \ldots, r_{14}$ are maintained implicitly as:

$$\begin{aligned} r_i &= r_i^j + r_i^{j+1}, && i = 8, 9, 10, 11, && j = 2(i-8); \\ r_i &= r_i^j + r_i^{j+1} + r_i^{j+2} + r_i^{j+3}, && i = 12, 13, && j = 4(i-12); \text{ and} \\ r_{14} &= r_{14}^0 + r_{14}^1 + \cdots + r_{14}^7, \end{aligned}$$

where the superindex indicates the process that owns that subvector. Thus, for example, the 8 subvectors necessary to build $r_{14} = r_{14}^0 + r_{14}^1 + \cdots + r_{14}^7$ are mapped to P0–P7, respectively.

Figure 4 shows the messages exchanged and the computations performed in order to transform $\hat{r}$ from inconsistent to consistent/distributed $\bar{r}$. This communication pattern can be easily recognized as a *binary-tree reduction* following the mapping of TDG nodes to processes in Figure 3 (right). The same pattern appears during the simultaneous lower triangular solve $\bar{L}\bar{y} = \hat{r}$, with the difference between transformation and solve residing in the specific operator that is applied to $\hat{r}$. For example, in Level 3 of the solve procedure, the following computations occur initially in P0 and P1:

$$\begin{array}{lll} \text{P0}: & y_0 := L_{00}^{-1} r_0, & t_{14}^0 := r_{14}^0 - L_{14,0} y_0, \\ & & t_{12}^0 := r_{12}^0 - L_{12,0} y_0, \\ & & t_8^0 := r_8^0 - L_{80} y_0; \\ \text{P1}: & y_1 := L_{11}^{-1} r_1, & t_{14}^1 := r_{14}^1 - L_{14,1} y_1, \\ & & t_{12}^1 := r_{12}^1 - L_{12,1} y_1, \\ & & t_8^1 := r_8^1 - L_{81} y_1. \end{array}$$

Next, P0 sends $t_{14}^0, t_{12}^0, t_8^0$ to P1, which are then used by the latter process to aggregate these with the local information forming t_{14}^{0-1}, t_{12}^{0-1}, $t_8^{0-1} (= t_8)$ in preparation for the next level of the triangular solve. The upper triangular solve $\bar{L}^T z = \bar{y}$ basically reverses the communication pattern in Figure 4, yielding a *binary-tree broadcast*.

After some initial local computations, from the communication point of view, the DOT product simply involves a *global reduction* (`Alltoall` in the MPI world).

To close this discussion, the dimension of each message is directly derived from the number of elements the corresponding subvector of $\hat{r}$, which in turn depends on the dimension of each diagonal block of $\hat{A}$. Thus, all r_i^j and t_i^j have, at least, the same size as the number of columns/blocks in A_{ii}.

Summary The previous elaboration identifies the communication patterns underlying a task-parallel ILU-PCG solver as well as some interesting observations for a TDG of arbitrary dimension:

1. The SPMV does not require any sort of communication.
2. The AXPY(-like) operations do not involve any communication either. This type of kernel is embarrassingly parallel, but those calls operating with consistent/partially replicated vectors replicate part of the computation to avoid the communication.
3. Each DOT product requires a global reduction (synchronization point). The loop body of the PCG solve can be re-arranged so that S9 is pushed up next to S6. A simultaneous execution of these two reductions then decreases the number of synchronization points from 3 to 2 per iteration in the classical PCG iteration.
4. The application of the preconditioner requires a binary-tree reduction for the lower triangular solve followed by a binary-tree broadcast for the upper triangular one.
5. The transformation $\hat{r} \to \bar{r} \to r$ occurs during the application of the preconditioner and the message exchanges required for this operation can be combined with those for the triangular solves.
6. The price to pay in order to expose increased levels of concurrency in some kernels is a partial replication of arithmetic, in particular due to the operation with the inconsistent matrices and vectors. An appropriate application of nested dissection aims to find a graph with small separators that concentrates most of the computational work on the leaves.

The transitions between states and storage patterns dictating the communications are summarized in Figure 5.

```
Compute preconditioner for $\hat{A} \approx \bar{L}\bar{L}^T = M$
Set starting guess $x$
Initialize $z, d, \beta, \tau, k := 0$

$\hat{r} := \hat{b} - \hat{A}x$
$\tau :=< r, \hat{r} >$
while $(\tau > \tau_{\max})$
```

Step	Operation	Communication pattern
	$\beta' := \beta$	–
S1 :	$\hat{w} := \hat{A}d$	–
S2 :	$\rho := \beta / < d, \hat{w} >$	Global reduction
S3 :	$x := x + \rho d$	–
S4 :	$\hat{r} := \hat{r} - \rho\hat{w}$	–
S5 :	$z := M^{-1}\hat{r}$	
S5.1 :	$\bar{y} := \bar{L}^{-1}\hat{r}$ $\bar{r} := \hat{r}$	Binary-tree reduction
S5.2 :	$z := \bar{L}^{-T}y$ $r := \bar{r}$	Binary-tree broadcast
S6 : S9 :	$\beta :=< z, \hat{r} >$ $\tau :=< r, \hat{r} >$	Combined global reduction
S7 :	$\alpha := \beta/\beta'$	–
S8 :	$d := \alpha d + z$	–
	$k := k + 1$	–

```
end while
```

Figure 5. Detailed implementation of the classical formulation of the task-parallel ILU-PCG solver annotated with state and storage modes for the variables and communication patterns: v for consistent/partially replicated; $\hat{v}$ for inconsistent; and $\bar{v}$ for consistent/distributed. Note the reorganization of the code to enable the merge of S6 and S9.

3.3. Mapping the TDG

The task-parallel implementation of ILUPACK's PCG method for clusters of multicore processors in [6] partitions the TDG at a certain level, mapping the tasks in the parts of the tree above this division to MPI ranks and those in the lower levels to OmpSs threads. The same observation applies to similar task-parallel implementations of alternative ILU-PCG solvers. Figure 6 shows this partitioning between the MPI and OmpSs "worlds" for a simple TDG consisting of 8 leaves that represents the solve with the lower triangular factor L. In the figure, two OmpSs threads per node collaborate to execute the tasks within the bottom two levels. For the top two levels, four MPI ranks take over the computation in order to process the tasks, while the OmpSs threads remain idle. The same partitioning between MPI and OmpSs occurs during other operations of the PCG iteration. On the leaves, the OmpSs threads operate with their local submatrices/subvectors. For the DOT products, these computations are propagated into the MPI world next in order to obtain the global result.

From the perspective of synchronization, there is little difference between the communication operations performed inside the MPI and OmpSs worlds. However, the combination of MPI+OmpSs allows this hybrid programming model solution to exploit dynamic scheduling within the cluster nodes via OmpSs. This implies that there is no a priori mapping of the tasks to the thread team, offering higher flexibility than the MPI solution and, in general, higher performance [6].

Inside OmpSs, task dependencies are mostly controlled with a fine-grain granularity that operates at the level of nodes of the TDG. Thus, for example, the computation of the tasks for the DOT product in S2 commence as soon as the tasks for SPMV (S1) generating the corresponding parts of $\hat{w}$ have completed their execution, with these dependencies in the loop body controlled by the OmpSs runtime. Inside the triangular solves (S5.1 and S5.2) the task dependencies are also controlled with a granularity dictated by the nodes of the TDG. The fact that some of these operations require explicit (MPI) communication points turn these synchronization among OmpSs tasks unavoidable,

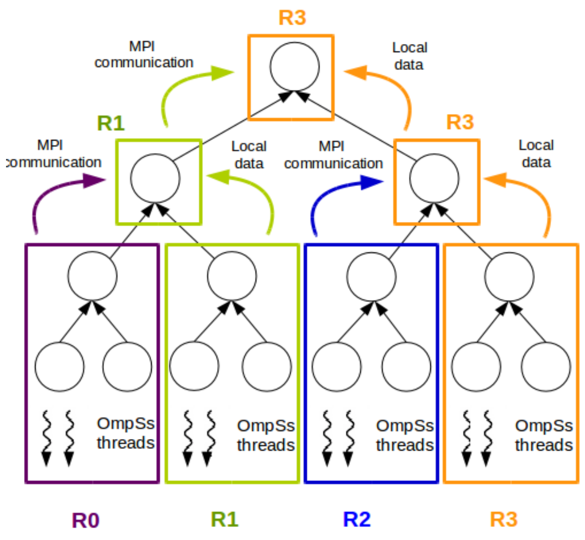

Figure 6. Mapping of a TDG to 4 MPI ranks (R0–R3) with 2 OmpSs threads per rank.

in practice yielding a barrier. This occurs, for example, within the DOT products as well as in the transition between the lower and upper triangular solves.

At this point, we recognize that a similar task-parallel hybrid (MPI+X) version could have been obtained using, e.g., OpenMP instead of OmpSs. The primary reason for adopting OmpSs in our codes is that, when we commenced our development of a multi-threaded task-parallel version of ILUPACK, the OpenMP standard did not support tasks. During the past few years, OpenMP has integrated this mechanism (in part, under the influence of task-parallel programming languages like OmpSs) and the differences between OpenMP and OmpSs tasking mechanisms are now blurry. However, we believe that OmpSs still provides some advanced features, such as sophisticated support for nested parallelism and task priorities, that can deliver slightly higher performance for some of our implementations.

4. COMMUNICATION-AVOIDING TECHNIQUES FOR ILU-PCG SOLVERS

4.1. Butterfly communication pattern

The butterfly transformation is an efficient communication scheme for collective operations [9, 10]. In the framework of the task-parallel PCG, this technique can be leveraged during the application of the preconditioner in S5, provided the lower triangular factor L is consistent and partially replicated. For our simplified scenario, this implies that P0 stores L_{00}, L_{20}, P1 stores L_{11}, L_{21}, and *both of them have a copy of* L_{22}. The direct consequence is an increase in memory usage, but there are also some changes on the communication patterns for the preconditioning step that we review next.

Lower triangular solve In this new scenario, with L consistent and partially replicated, we solve the lower triangular system in S5.1 by first computing in parallel

$$\begin{array}{ll} \mathsf{P0}: & y_0 := L_{00}^{-1} r_0, \quad t_2^0 := r_2^0 - L_{20} y_0, \\ \mathsf{P1}: & y_1 := L_{11}^{-1} r_1, \quad t_2^1 := r_2^1 - L_{21} y_2. \end{array}$$

However, now P0 and P1 *exchange* t_2^0 and t_2^1, and then replicate the computation:

$$\begin{array}{ll} \mathsf{P0}: & t := t_2^0 + t_2^1, \quad y_2 := L_{22}^{-1} t, \\ \mathsf{P1}: & t := t_2^0 + t_2^1, \quad y_2 := L_{22}^{-1} t. \end{array}$$

```
Compute preconditioner for $\hat{A} \approx LL^T = M$
Set starting guess $x$
Initialize $z, d, \beta, \tau, k := 0$

$\hat{r} := \hat{b} - \hat{A}x$
$\tau :=< r, \hat{r} >$
while ($\tau > \tau_{\max}$)
```

Step	Operation	Communication pattern
	$\beta' := \beta$	
S1 :	$\hat{w} := \hat{A}d$	–
S2 :	$\rho := \beta / < d, \hat{w} >$	Global reduction
S3 :	$x := x + \rho d$	–
S4 :	$\hat{r} := \hat{r} - \rho \hat{w}$	–
S5 :	$z := M^{-1}\hat{r}$	
S5.1 :	$y := L^{-1}\hat{r}$	Binary-tree reduction-broadcast
	$r := \hat{r}$	
S5.2 :	$z := L^{-T}y$	–
S6 :	$\beta := < z, \hat{r} >$	Combined global reduction
S9 :	$\tau := < r, \hat{r} >$	
S7 :	$\alpha := \beta/\beta'$	–
S8 :	$d := \alpha d + z$	–
	$k := k + 1$	

```
end while
```

Figure 7. Detailed implementation of the classical formulation of the task-parallel ILU-PCG solver enhanced with the butterfly transformation.

After these operations, the solution y is consistent and partially replicated. Compared with this, in the classical PCG solve implementation described in section 3, P0 had to send t_2^0 to P1, and y_2 was only computed in the latter. As a result, the solution $\bar{y}$ was consistent/distributed.

Upper triangular solve To solve the triangular system in S5.2, both processes replicate the computation

$$\begin{aligned} \text{P0}: &\quad z_2 := L_{22}^{-T} y_2, \\ \text{P1}: &\quad z_2 := L_{22}^{-T} y_2, \end{aligned}$$

and next they compute in parallel

$$\begin{aligned} \text{P0}: &\quad y_0 := y_0 - L_{20}^{T} z_2, \quad z_0 := L_{00}^{-T} y_0, \\ \text{P1}: &\quad y_1 := y_1 - L_{21}^{T} z_2, \quad z_1 := L_{11}^{-T} y_1. \end{aligned}$$

Thus, there is no communication during the upper triangular solve, and the result z is consistent and partially replicated, as required by subsequent operations. With respect to the classical iterative PCG scheme, we save the communication of z_2 from P1 to P0.

Transformation of the residual In order to create a consistent/partially replicated version of $\hat{r}$ in r, during the lower triangular solve in S5.1, P0 and P1 exchange r_2^0 and r_2^1. This information is then used by both processes to obtain $r_2 := r_2^0 + r_2^1$, building a consistent and partially replicated copy of the residual vector already during this step. Contrary to the classical PCG, no additional operation is required during S5.2 when the butterfly transformation is in place. In conclusion, the variations in the communication patterns during the transformation of the residual, with respect to the classical PCG solve, are analogous to those experienced by the lower/upper triangular solves.

Generalization Figure 4 illustrates the changes in the message exchanges for the transformation from inconsistent $\hat{r}$ to consistent/partially replicated r in a single step, analogous to those appearing

```
Compute preconditioner for $\hat{A} \approx LL^T = M$
Set starting guess $x^0$
Initialize $z, d, \hat{s}, \hat{w}, \beta, \gamma, \tau, \alpha, \epsilon, k := 0$
$\hat{r} := \hat{b} - \hat{A}x$
$\tau :=< r, \hat{r} >$
while $(\tau > \tau_{\max})$
```

Step	Operation	Kernel	Communication pattern
	$\beta' := \beta$	Copy	–
S1 :	$\rho := \beta/(\gamma + \alpha\epsilon)$	Scalar operation	–
S2 :	$d := \alpha d + z$	AXPY-like	–
S3 :	$\hat{w} := \alpha\hat{w} + \hat{s}$	AXPY-like	–
S4 :	$x := x + \rho d$	AXPY	–
S5 :	$\hat{r} := \hat{r} - \rho\hat{w}$	AXPY	–
S6 :	$z := M^{-1}\hat{r}$	Apply preconditioner	–
S6.1 :	$y := L^{-1}\hat{r}$ $r := \hat{r}$	Lower triangular solve	Binary-tree reduction-broadcast
S6.2 :	$z := L^{-T}y$	Upper triangular solve	–
S7 :	$\hat{s} := \hat{A}z$	SpMV	–
S8 :	$\beta := < z, \hat{r} >$	DOT product	Combined global reduction
S9 :	$\gamma := < z, \hat{s} >$	DOT product	
S10 :	$\epsilon := < z, \hat{w} >$	DOT product	
S11 :	$\tau := < r, \hat{r} >$	DOT product	
S12 :	$\alpha := \beta/\beta'$ $k := k + 1$	Scalar operation	–

```
end while
```

Figure 8. Detailed implementation of the Eijkhout's variant of the task-parallel ILU-PCG solver enhanced with the butterfly transformation.

in the solve $Ly = \hat{r}$ that yields a consistent/partially replicated y. The communication pattern can be identified as a simultaneous *binary-tree reduction-broadcast*. In this case, there is no communication for the upper triangular solve.

Summary The result of applying the butterfly transform is illustrated in Figure 7 using the notation for state and storage modes. This technique replicates some of the information of the preconditioner, increasing the demand of memory. On the positive side, it diminishes the amount of messages (though not the total volume of communication), and it does not change the numerical properties of the solver.

4.2. Eijkhout's PCG solver

Synchronization due to (global) reductions is a well-known cause for reduced scalability on massively-parallel platforms. For iterative Krylov subspace based methods, synchronization is in particular due to the DOT products. In response, a number of research efforts have addressed this bottleneck, via new CA methods or by overlapping (hiding) communication with computation; see, e.g., [17, 11] and the references therein. In brief, these approaches reformulate the operations in the iterative method, replacing some of the computations by alternative recurrences while aiming to shift together all DOT products in the loop body so that their communication stages can be merged. Unfortunately, these variants suffer from the accumulation of rounding error, presenting less favorable numerical properties than the classical CG method [17].

Eijkhout's variant of PCG [11] is a representative example of the effect of these CA approaches on the communication pattern of a parallel implementation. The variant is presented in Figure 8, where it is easy to detect that, in exchange for an increase in the number of AXPY(-like)+DOT kernels from 3+3 (in the classical formulation) to 4+4, the DOT products are now all consecutive (S8–S11). Clearly, this enables their combination into a single synchronization point. The communication due to the lower triangular solve, during the preconditioner application, still remains but now occurs right before the SPMV. Unfortunately, the data dependency between these two operations prevents their concurrent execution.

Figure 8 also illustrates how a task-parallel execution can integrate Eijkhout's variant, with the following similarities to the classical PCG (with the butterfly transformation in place in both cases):

1. Matrix $\hat{A}$ is inconsistent and factor L is consistent/partially replicated.
2. Vectors $\hat{r}, \hat{d}$ are inconsistent, with a consistent/partially replicated copy of the former being created during each lower triangular solve. The new recurrence vector $\hat{s}$ (not present in the classical PCG) also shares this state/storage mode.
3. Vectors x, z, d as well as the temporary y are maintained in consistent/partially replicated state during the iteration.

With these premises, the communication patterns appearing in Eijkhout's variant are similar to those in the classical algorithm (with butterfly transformation in place):

1. The AXPY(-like) operations always combine two vectors with the same state/storage mode, requiring no communication.
2. The lower triangular solve receives an inconsistent right-hand side vector producing a consistent partially replicated result. In combination with the transformation $\hat{r} \rightarrow r$, it only requires a binary-tree reduction-broadcast.
3. The SPMV receives an consistent/partially replicated input vector and calculates an inconsistent result, with no communication involved.
4. All four DOT products multiply an inconsistent vector with a consistent/partially replicated one, requiring a global reduction each that can be combined into a single synchronization point.

Our current implementation of Eijkhout's variant mimics our routine for the classical PCG in that the task dependencies are controlled by OmpSs with the granularity of nodes of the TDG.

4.3. *Other CA CG solvers*

There exist other variants that trade off numerical stability for higher scalability among which, we can mention Chronopoulos and Gear's [21], Meurant's [22], Saad's [23], and a pipelined version of the first [17]. Like Eijkhout's, all these variants present, for each iteration, a single synchronization point (in the form a combined global reduction), together with a SPMV and, for preconditioned versions, the application of the preconditioner. They differ slightly in the number of recurrence vectors and the order of the operations. Therefore, in a task-parallel implementation accelerated with an ILU-preconditioner, their communication patterns do not differ from those already exposed in this section for Eijkhout's variant.

Recent s-step Krylov methods further reduce the number of synchronization points, aggressively bargaining stability for parallel scalability, as their numerical properties rapidly deteriorate with the size of the step s; see [18] and the references therein. The communication analysis of these variants is part of ongoing work.

4.4. *Merging task-parallel vector operations*

The task-parallel solvers present a few consecutive AXPY(-like) operations that require no communication: S3–S4 for the classical ILU-PCG, with and without the butterfly transformation; and S2–S5 for Eijkhout's variant. When these tasks are executed in parallel, they incur a certain scheduling cost for OmpSs because they are divided into tasks following the partitioning induced by the system matrix. This small overhead can be avoided if we merge these sequences into a single task-parallel operation from the point of view of the OmpSs scheduler.

4.5. *Relaxing the convergence test*

In Eijkhout's variant with the butterfly technique, the transformation of the residual vector from inconsistent to consistent/partially replicated during the triangular solve is performed to compute

Matrix	Dimension n	#non-zeros n_z	Density (%)
A159	4,019,679	16,002,873	9.90E-7
A200	8,000,000	31,880,000	4.98E-7
A252	16,003,008	63,821,520	2.49E-7
A318	32,157,432	128,326,356	1.24E-7
A400	64,000,000	255,520,000	6.23E-8

Table I. Matrices employed in the experimental evaluation. In the table n_z counts the non-zeros in the upper triangular part only.

the tolerance threshold τ. This in turn is necessary in order to check the convergence test at the beginning of the following iteration. If we perform the convergence test every s iterations, we can save the transformation $\hat{r} \rightarrow r$ during the lower triangular solve in S6.2 (and the DOT product for τ) during $s-1$ iterations. With this option, the binary-tree reduction still occurs, but involves less data. In exchange for this, the solver may incur up to $s-1$ extra iterations. Nevertheless, a rough analysis of the convergence rate for the iteration can offer some heuristic strategy to reduce this risk.

5. EXPERIMENTAL RESULTS

5.1. setup

The experiments in this section employed IEEE754 double-precision arithmetic and were carried out on the *MareNostrum* supercomputer at Barcelona Supercomputing Center (BSC). We performed our evaluation on 16 compute nodes of this infrastructure, connected via an Infiniband Mellanox FDR10 network. Each node comprises two 8-core Intel Xeon E5-2670 processors (2.6 GHz) and 64 Gbytes of DDR3 RAM. The codes were compiled using Mercurium C/C++ (1.99.8), with OpenMPI (1.8.1). Other software included OmpSs (16.06), ILUPACK (2.4), ParMetis[‡] (4.0.2) for the graph reorderings, and `Extrae+Paraver`[§] (3.4.1 + 4.6.3) to obtain and visualize execution traces.

For the experimental analysis, we relied on a s.p.d. coefficient matrix arising from the finite difference discretization of a 3D Laplace problem; see Table I and [5] for details. For the linear systems, the right-hand side vector b was initialized with all entries set to 1, and the PCG iterate was started with the initial guess $x_0 \equiv 0$. 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 have chosen the Laplacian operator since this particular problem is easy to generate and to verify. Certainly, there exits alternative problems that are harder to solve and tackling this equation is easy from the mathematical point of view. However, the Laplacian problem represents a worst-case scenario for parallel computing. Concretely, since solving single subsystems is easy and fast, the communication overhead for the Laplacian case is more significant and dominating than for other problems. Moreover, the graph structure of the underlying graph reveals all aspects of domain decomposition as obtained by a graph partitioner. This is worse than for many other problems because its uniform structures lead to relatively large-size interfaces, increasing the work that is required beyond the leaf tasks.

5.2. Analysis of single-node execution

Figure 9 provides `Extrae` traces corresponding to the execution one iteration of two of the task-parallel PCG solvers: classical scheme and Eijkhout's variant. We note that the butterfly transformation, the merge of AXPY(-like) vector operations, or the integration of an ILU(0)

[‡]`http://glaros.dtc.umn.edu/gkhome/metis/parmetis/`
[§]`http://www.bsc.es/computer-sciences/extrae`.

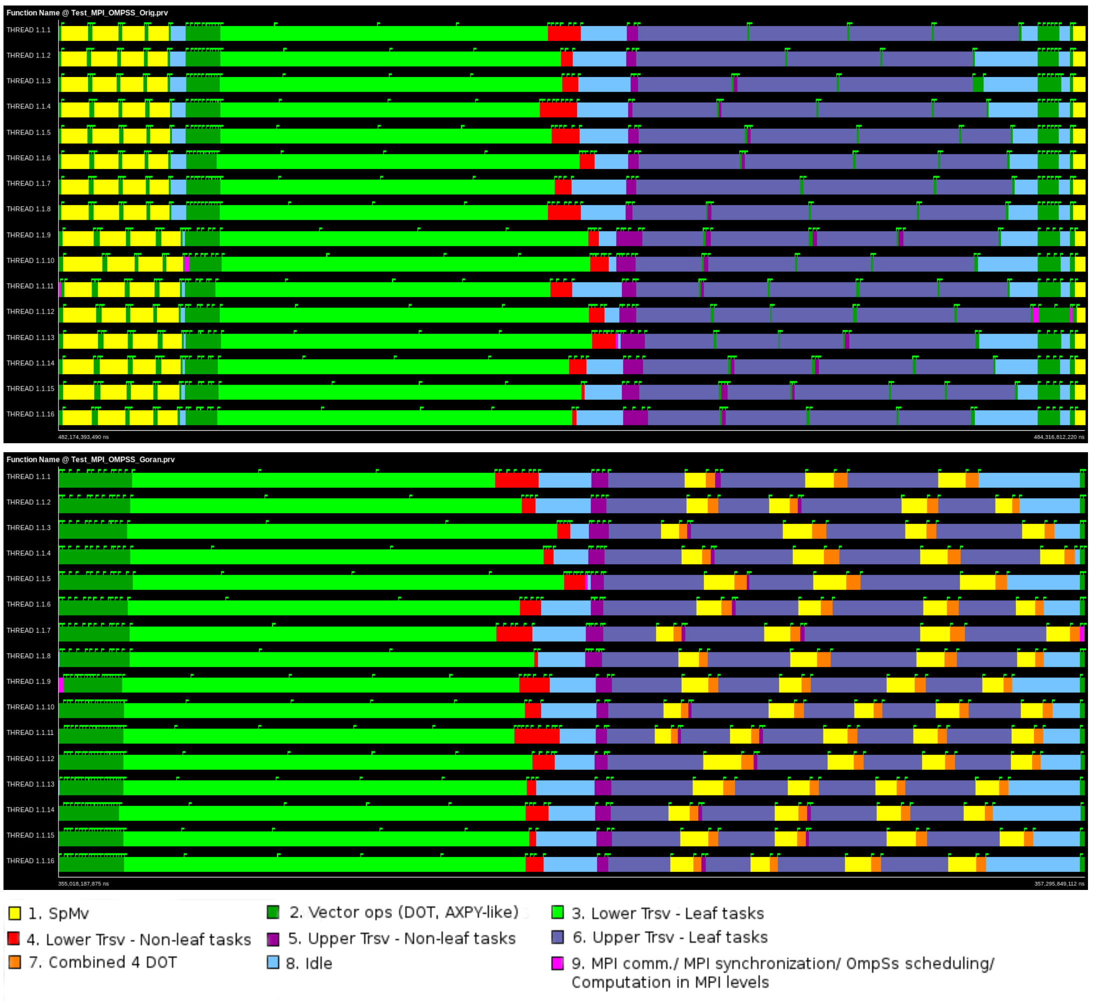

Figure 9. Execution traces for the classical CG and Eijkhout's variant (top and bottom, respectively), in both cases preconditioned with ILUPACK: single node, 1 MPI rank, 16 OmpSs threads per MPI rank. The execution time of one iteration of the classical CG (top) is 2.14 s while for Eijkhouts variant (bottom) it is 2.28 s.

preconditioner instead of ILUPACK's produce only minor changes in the execution traces. For this initial experiment, we employed a single node of the cluster, 1 MPI rank, and 16 OmpSs threads. The TDG was composed of 64 leaves (4 leaf tasks per thread) and 7 levels.

In the top timeline in the figure, we can observe the sequence of operations for the classical PCG iteration, consisting of SpMV (S1), DOT product (S2), and 2×AXPY (S3–S4); followed by the application of preconditioner (upper and lower triangular solves; S5.1 and S5.2, respectively); and terminated with three vector kernels: 2×DOT product (S6 and S9) and AXPY(-like) (S8). In the bottom timeline (Eijkhout's variant) the sequence commences with the four AXPY(-like) operations (S2–S5); which are directly followed by the application of the preconditioner (S6.1 and S6.2), and the SpMV (S7); and completed with the combined four DOT products (S7–S11).

Both traces show that, during the triangular solves, each OmpSs thread executes between 5 and 7 leaf tasks. The traces also expose that the SpMV and triangular solves dominate the execution time of the iteration, but the cost of the vector operations is not negligible. Furthermore, for the triangular solves, the computations corresponding to non-leaf tasks also contribute a mild cost compared with that involving the leaves.

The traces identify a synchronization point in the transition between the upper triangular solve and the lower one, in addition to those imposed by the DOT products. This yields three synchronization points per iteration for the classical PCG solver compared with only two for Eijkhout's variant. The negative impact of these synchronizations comes from the communication overhead and, especially for this single node execution, workload imbalance that leads to idle waits and waste of computational resources.

5.3. *Optimal configuration*

In [6] we evaluated the performance of the parallel MPI+OmpSs version of the classical PCG solve preconditioned with ILUPACK for different combinations of MPI ranks and OmpSs threads per node (configurations). Given the node architecture (the same computer targeted in the present work), with 2 sockets/8 cores per socket, we employed 1, 2, 4, 8 or 16 MPI ranks per node and the corresponding number of OmpSs threads that filled all cores per node: 16, 8, 4, 2 or 1, respectively. The analysis of performance and weak/strong scalability in that work offered three key insights:

1. In general, the best configuration maps 2 MPI ranks/8 OmpSs threads per node, mimicking the internal socket/core architecture of the servers.
2. The MPI+OmpSs implementation outperforms its pure-MPI counterpart by a non-negligible margin that is in the range 5–10%.
3. The best configuration employs 1 leaf per core. While a raise in the number of leaves can potentially produce a more balanced distribution of the workload, it also increases the number of levels and the cost of processing the intermediate (non-leaf) tasks. At a certain point during the application of the preconditioner, the number of tasks in a level becomes smaller than the amount of cores, yielding idle waits.

We thus adopt the same decisions for the remaining experiments, mapping 2 MPI ranks per node, with 8 OmpSs threads per MPI rank; and creating a TDG composed of 1 leaf per core.

5.4. *Scalability*

We next evaluate the scalability of the task-parallel ILU-PCG solvers, focussing this analysis on the effect of the two major CA techniques: the butterfly transformation and Eijkhout's variant. The top two graphs in Figures 10 and 11 show the *execution time-per-iteration* of the task-parallel PCG solver for the A400 problem as the resources are increased from 1 to 16 nodes respectively using a single socket or both sockets per node. For reference, the latter figure also includes the results for an implementation that relies on MPI only. We note that the results collect the average iteration time. The execution time of each iteration, for the distinct versions, which already provides some information on the dispersion of the values, showed relative small variations between iterations. For example, in the execution with a single node and 16 cores, the average time-per-iteration for the classical variant was about 0.278 s and the standard deviation between the time of the iterations was around 0.0219 s only. Similar dispersion results were observed for other variants and parallel configurations. In general, all implementations show a similar decrease in the iteration time as the number of cores grows, exposing their scalability.

The bottom two graphs in both figures assess the weak scalability of the task-parallel ILU-PCG solvers. To perform this evaluation, we set the number of non-zeros of the sparse matrix (n_z) to be roughly proportional to the number of cores. However, we note that n_z only offers an estimation of the computational cost, as the fill-in/quality of the preconditioner, among other factors, has a significant role. The bottom row of plots reports the execution time per iteration of the parallel implementations for the different matrices in Table I. These results show that, for all implementations, the execution times grow with the number of nodes/cores and the problem dimension at a similar pace. The reason is that the number of flops per iteration increases with the number of cores and, therefore, levels due to the overhead caused by the operation with inconsistent data.

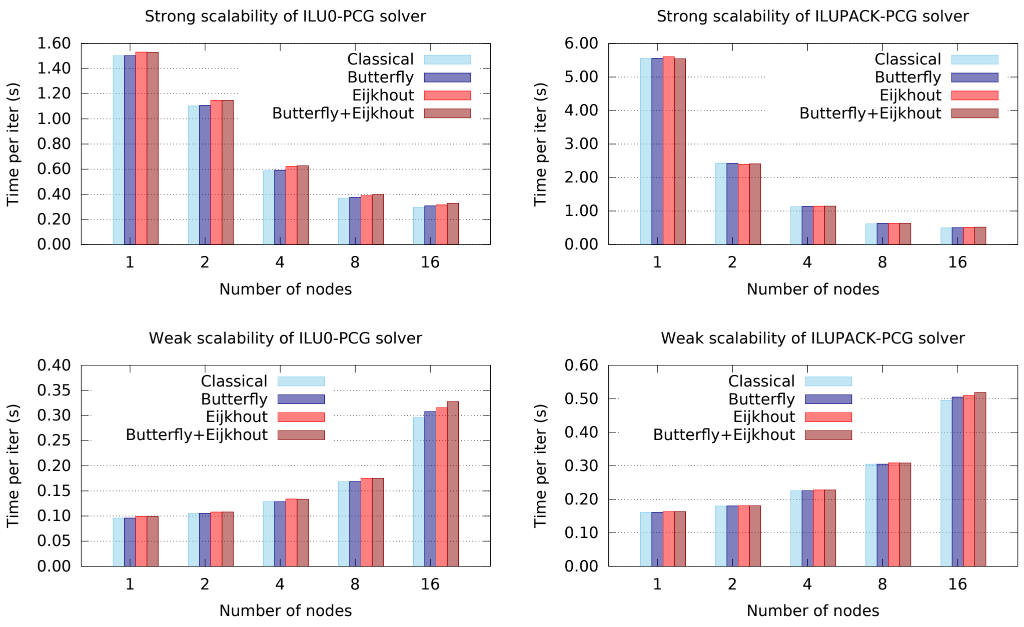

Figure 10. Scalability of the task-parallel ILU-PCG solvers using a single socket (8 cores) per node.

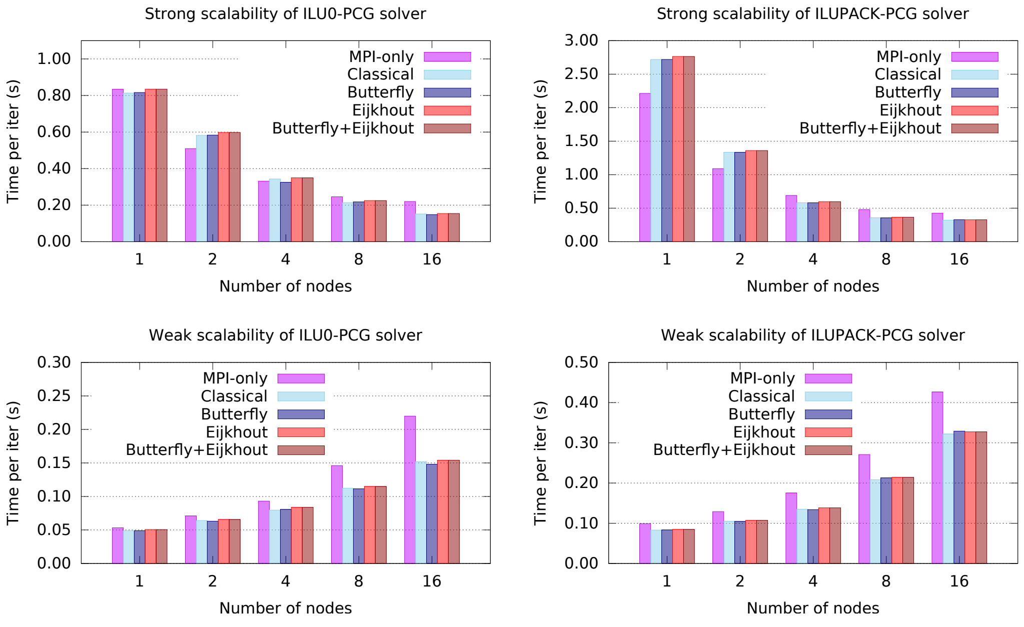

Figure 11. Scalability of the task-parallel ILU-PCG solvers using two sockets (16 cores) per node.

5.5. *Parallel performance*

All plots in Figures 10 and 11 reveal that the butterfly transformation as well as Eijkhout's CA variant introduce small overheads compared with the classical iteration, independently of whether the implementations are preconditioned via ILU(0) or ILUPACK. This is more visible for the

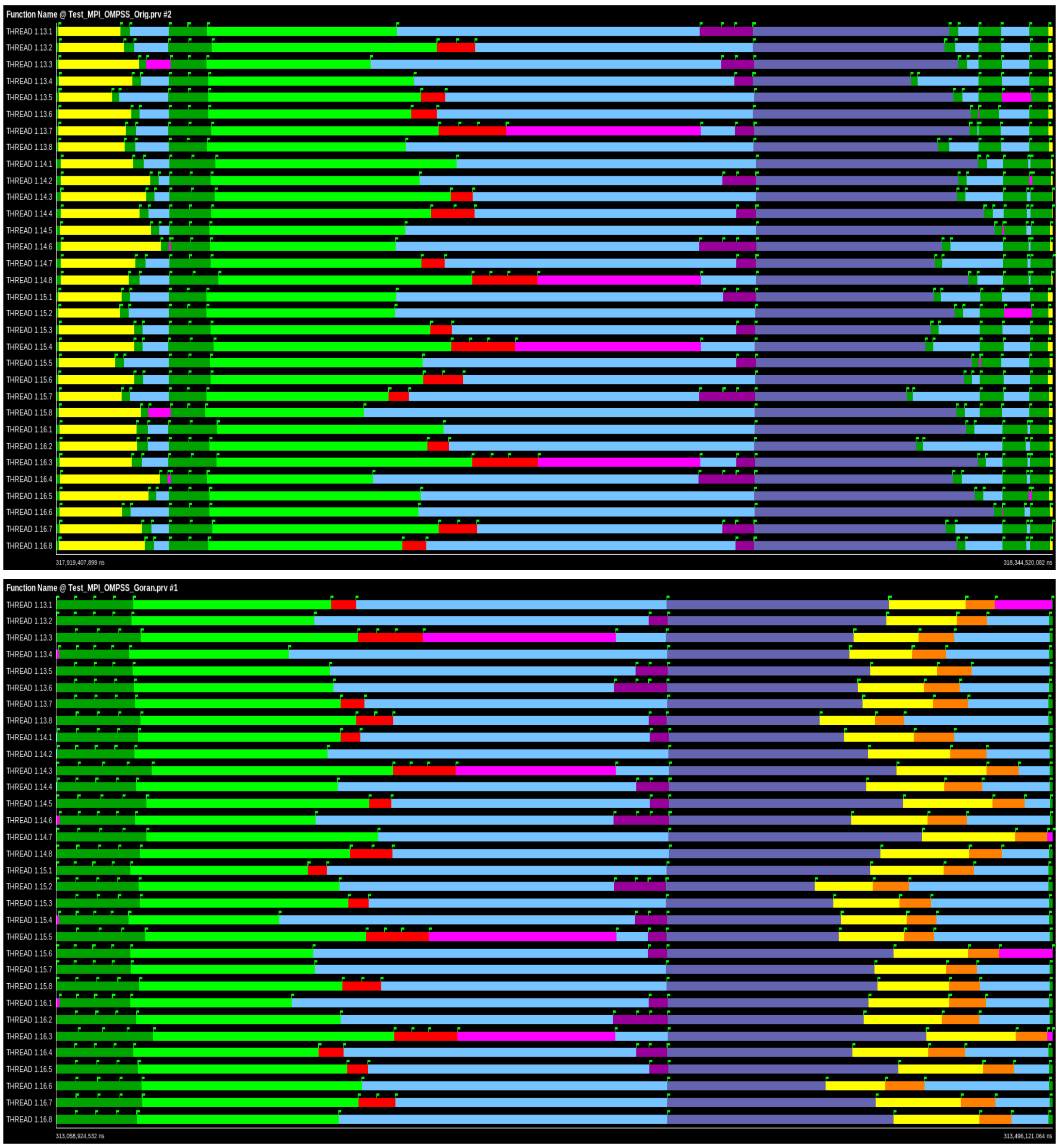

Figure 12. Execution traces for the classical CG and Eijkhout's variant (top and bottom, respectively) preconditioned with ILUPACK: 8 nodes, 2 MPI ranks per node, and 8 OmpSs threads per MPI rank. For clarity, we only show the detail for nodes 7 and 8. The legend of colors is the same as that in Figure 9. The execution time of one iteration of the classical CG (top) is 0.42 s while for Eijkhouts variant (bottom) it is 0.44 s.

execution that employs a single node for the strong scaling experiment and 8–16 nodes for the weak scaling one.

For the butterfly transform, a separate analysis of communication traces revealed that the use of MPI's `Sendrecv` primitive (necessary to implement the binary-tree reduction-broadcast that is present in this scheme) produces a slightly worse overlap of communication and computation than the regular MPI message exchanges via `Send` and `Recv`. We next discuss in detail the comparison between the classical PCG iteration and Eijkhout's variant. For this purpose, we will employ traces for an execution using 8 nodes; with 2 MPI ranks per node and 8 OmpSs threads per MPI rank; and the A400 problem. This yields a TDG with 128 leaf tasks (16 leaves per node, 8 per MPI rank), organized into a binary-tree consisting of 8 levels.

Figure 12 reports traces for the classical iteration vs Eijkhout's variant, in both cases accelerated with ILUPACK's preconditioner. The timelines in the figure reveal several observations:

- The limited dimension of the problem compared with the amount of resources (128 cores) produces a substantial increase of the overheads due to inter-process communication/synchronization and the computation with top levels of the tree hierarchy during the triangular solves (bars in magenta color).
- The generation of a TDG with one leaf per OmpSs thread is the source of a significant workload imbalance, especially visible for the SPMV and the triangular solves.
- The binary tree structure imposes a special synchronization pattern with the same structure in both execution because the upper triangular solve cannot commence till the result of the lower triangular system is available.
- For the classical iteration (first and third timelines), the SPMV is immediately followed by a DOT product, which introduces a synchronization in the execution. The duration of these two operations is determined by that of the slowest core. A similar situation appears at the end of the iteration, when the upper triangular solve is followed by (an AXPY) and a DOT product.
- Instead of the two periods of idle time per iteration present in the classical PCG solve, we easily identify a single one in Eijkhout's variant. However, the workload imbalance due to the upper triangular solve and SPMV aggregate so that the duration of this single period is approximately equal to the addition of the two periods in the classical PCG iteration. As a result, the higher execution time of Eijkhout's variant is determined by the larger number of flops per iteration (4+4 AXPY(-like) and DOT products vs 3+3).
- Unfortunately, increasing the number of leaves/levels of the TDG, which could improve the distribution of the workload, will likely shift more cost to the computation with the non-leaf tasks and an increase of idle resources.

6. CONCLUSIONS

We have proposed, analyzed and evaluated a number of parallel implementations of the PCG method. All these parallel routines extract task-parallelism via nested dissection and leverage the message-passing programming model underlying MPI and the task-parallel approach in OmpSs to map the execution of the iterative PCG solve to a cluster of multicore processors. They differ in the definition of the iteration recurrences (classical PCG iteration vs Eijkhout's CA formulation) and the type of preconditioner they integrate (ILU(0) vs ILUPACK). In addition, we accommodate complementary communication/synchronization-avoiding strategies such as the butterfly communication scheme and a strategy to merge OmpSs tasks.

Our experimental results show strong and weak scalabilities for all variants on up to 16 nodes/256 cores. (Our experiences with asynchronous versions of `MPI_Send` did not show any performance differences.) These results can be expected to carry over to any task-parallel formulation of a PCG solver that relies on an ILU-type preconditioner as well as Chronopoulos and Gear's, Meurant's and Saad's CA formulations of the CG method, among others.

The analysis also marks a research direction that aims to increase the asynchronism of our task-parallel PCG implementations, and improve workload balance, in order to render a faster execution. For this purpose, we plan to decompose some of the computational kernels in the iteration (especially the SPMV) to expose further levels of task-parallelism that can be then exploited from OmpSs. This is part of ongoing work, together with the analysis of the communication patterns for s-step Krylov methods, and the use of asynchronous versions of `MPI_Recv`.

ACKNOWLEDGEMENT

J. Aliaga, M. Barreda, G. Flegar and E. S. Quintana-Ortí were supported by project TIN2014-53495-R of the MINECO and FEDER, project P1-1B2015-26 of the *Universitat Jaume I*, and the H2020 EU FETHPC Project 671602 "INTERTWinE". María Barreda was supported by the FPU program of the *Ministerio de Educación, Cultura y Deporte*.

The authors thankfully acknowledge the computer resources provided by BSC-CNS (*Centro Nacional de Supercomputación*).

REFERENCES

1. Kepner J, Gilbert J ((eds.)). *Graph Algorithms in the Language of Linear Algebra*. SIAM, 2011.
2. Saad Y. *Iterative Methods for Sparse Linear Systems*. SIAM, 2003.
3. George T, Gupta A, Sarin V. An empirical analysis of the performance of preconditioners for SPD systems. *ACM Trans. Mathematical Software* August 2012; **38**(4):24:1–24:30.
4. Aliaga JI, Badia RM, Barreda M, Bollhöfer M, Dufrechou E, Ezzatti P, Quintana-Ortí ES. Exploiting task- and data-parallelism in ILUPACK's preconditioned CG solver on NUMA architectures and many-core accelerators. *Parallel Computing* 2016; **54**:97–107.
5. Aliaga JI, Bollhöfer M, Martín AF, Quintana-Ortí ES. Exploiting thread-level parallelism in the iterative solution of sparse linear systems. *Parallel Computing* 2011; **37**(3):183–202.
6. Aliaga JI, Barreda M, Bollhöfer M, Quintana-Ortí ES. Exploiting task-parallelism in message-passing sparse linear system solvers using OmpSs. *Euro-Par 2016: Parallel Processing: 22nd Int. Conf. Parallel and Distributed Computing*, Springer, 2016; 631–643.
7. Duran A, Ferrer R, Ayguadé E, Badia RM, Labarta J. A proposal to extend the OpenMP tasking model with dependent tasks. *International Journal of Parallel Programming* 2009; **37**(3):292–305, doi:10.1007/s10766-009-0101-1.
8. Programming Models@BSC. The OmpSs programming model. https://pm.bsc.es/ompss 2014.
9. Bruck J, Ho CT, Kipnis S, Upfal E, Weathersby D. Efficient algorithms for all-to-all communications in multi-port message-passing systems. *IEEE Trans. on Parallel and Distributed Systems* 1997; **8**(11).
10. Hoefler T, Schneider T. Runtime detection and optimization of collective communication patterns. *Proc. 21st Int. Conf. on Parallel Architectures and Compilation Techniques*, PACT '12, 2012; 263–272.
11. Dongarra J, Eijkhout V. Finite-choice algorithm optimization in Conjugate Gradients. *LAPACK Working Note 159 UT-CS-03-502*, University of Tennessee 2003.
12. Hénon P, Ramet P, Roman J. PaStiX: A high-performance parallel direct solver for sparse symmetric definite systems. *Parallel Computing* 2002; **28**(2):301–321.
13. P R Amestoy JK I S Duff, L'Excellent JY. A fully asynchronous multifrontal solver using distributed dynamic scheduling. *SIAM J. Matrix Analysis and Applications* 2001; **23**(1):15–41.
14. Irony D, Shklarski G, Toledo S. Parallel and fully recursive multifrontal supernodal sparse Cholesky. *Future Generation Computer Systems — Special issue: Selected numerical algorithms archive* 2004; **20**(3):425–440.
15. Schenk O, Gärtner K. On fast factorization pivoting methods for symmetric indefinite systems. *Electronic Trans. Numerical Analysis* 2006; **23**(1):158–179.
16. Li XS. An overview of SuperLU: Algorithms, implementation, and user interface. *ACM Trans. Mathematical Software* 2005; **31**(3):302–325.
17. Cools S, Y EF, Agullo E, Giraud L, Vanroose W. Analysis of rounding error accumulation in Conjugate Gradients to improve the maximal attainable accuracy of pipelined CG. *Research Report RR-8849*, Inria Bordeaux Sud-Ouest 2016.
18. Hoemmen M. Communication-avoiding Krylov subspace methods. PhD Thesis, Berkeley, CA, USA 2010. AAI3413388.
19. Bollhöfer M, Grote MJ, Schenk O. Algebraic multilevel preconditioner for the Helmholtz equation in heterogeneous media. *SIAM J. Scientific Computing* 2009; **31**(5):3781–3805.
20. Bollhöfer M, Saad Y. Multilevel preconditioners constructed from inverse–based ILUs. *SIAM J. Scientific Computing* 2006; **27**(5):1627–1650. Special issue on the 8–th Copper Mountain Conference on Iterative Methods.
21. Chronopoulos A, Gear C. s-step iterative methods for symmetric linear systems. *J. Computational and Applied Mathematics* 1989; **25**(2):153–168.
22. Meurant G. Multitasking the Conjugate Gradient method on the Cray X-MP/48. *Parallel Computing* 1987; **5**(3):267 – 280.
23. Saad Y. Krylov subspace methods on supercomputers. *SIAM J. Scientific and Statistical Computing* 1989; **10**(6):1200–1232.