Exploiting Task-Parallelism in Message-Passing Sparse Linear System Solvers Using OmpSs

José I. Aliaga[1], María Barreda[1(✉)],
Matthias Bollhöfer[2], and Enrique S. Quintana-Ortí[1]

[1] Dpto. de Ingeniería y Ciencia de Computadores,
Universidad Jaume I, Castellón, Spain
{aliaga,mvaya,quintana}@icc.uji.es
[2] Institute of Computational Mathematics,
TU Braunschweig, Braunschweig, Germany
m.bollhoefer@tu-bs.de

Abstract. We introduce a parallel implementation of the preconditioned iterative solver for sparse linear systems underlying ILUPACK that explores the interoperability between the message-passing MPI programming interface and the OmpSs task-parallel programming model. Our approach commences from the task dependency tree derived from a multilevel graph partitioning of the problem, and statically maps the tasks in the top levels of this tree to the cluster nodes, fixing the inter-node communication pattern. This mapping induces a conformal partitioning of the tasks in the remaining levels of the tree among the nodes, which are then processed concurrently via the OmpSs runtime system.

The experimental analysis on a cluster with high-end Intel Xeon processors explores several configurations of MPI ranks and OmpSs threads per process showing that, in general, the best option matches the internal architecture of the nodes. The results also report significant performance gains for the MPI+OmpSs version over the initial MPI code.

Keywords: Programming models · Sparse linear systems · Preconditioned iterative solvers · Task-level parallelism · ILUPACK · MPI · OmpSs

1 Introduction

The solution of large sparse systems of linear equations is a key linear algebra problem arising in many scientific and engineering applications that involve the discretization of partial differential equations (PDEs) [18]. Moreover, the connection between sparse linear algebra and graph algorithms has turned this type of problem into an appealing means to mine the vast amount of information in social networks and other big data analytic processes [13].

ILUPACK[1] (Incomplete LU decomposition PACKage) is a numerical package that contains efficient multilevel ILU factorization solvers, based on Krylov

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

© Springer International Publishing Switzerland 2016
P.-F. Dutot and D. Trystram (Eds.): Euro-Par 2016, LNCS 9833, pp. 631–643, 2016.
DOI: 10.1007/978-3-319-43659-3_46

subspace methods [18], for large-scale sparse linear systems with up to millions of equations [10,19,21]. In previous work, we exploited the task-parallelism exposed by the task dependency graph (TDG) associated with the sparse matrix to develop parallel versions of ILUPACK's preconditioned Conjugate Gradient (PCG) solver. The target platforms for these past efforts included shared-memory multiprocessors via OpenMP [2,3], multicore architectures with OmpSs[2] [1], and clusters using MPI [1,4].

Unfortunately, the previous MPI version of ILUPACK [1,4] could only map one leaf of the TDG to each MPI rank, impeding the exploitation of other types of parallelism internally to the nodes. This is a strong limitation for clusters consisting of "fat" nodes equipped with a significant numbers of cores per node, as the static correspondence between tasks and MPI ranks may result in an unbalanced distribution of the workload and, therefore, inefficiency.

In this work, we present a new implementation of ILUPACK which merges MPI and OmpSs to exploit the benefits of each programming model, and allows the execution of the solver with more than one leaf per MPI process. In addition, we perform an experimental evaluation in order to assess the impact of the MPI+OmpSs configuration, problem dimension, and number of leaves per core on the performance of the iterative solve. Our results on a cluster equipped with 16 Intel Xeon cores per node reveals that the MPI+OmpSs version consistently outperforms the initial MPI code in terms of both strong and weak scaling.

There exist other packages also based on ILU factorizations and Krylov subspace methods. For example, pARMS [15] is an MPI-based library of parallel solvers for solving general sparse linear systems where the preconditioner is based on an algebraic recursive multilevel ILU. In contrast to ILUPACK, it relies on an independent set strategy for partitioning the leading systems into small diagonal blocks and then it re-applies the strategy recursively. In [7] a completely different parallel approach to ILUs was presented treating the error between the ILU and the original matrix as sequences of nonlinear equations to be improved in parallel rather than decomposing the system into a hierarchy of independent blocks. In [9], a parallel incomplete factorization approach uses direct solver techniques based on the level-of-fill and the underlying graph properties Beside ILU-based techniques, efficient parallel direct solvers [11,12,14,17,20] based on OpenMP/MPI rely on tree–parallelism and a variety of sophisticated techniques. Moreover, there are certainly many further parallel preconditioning methods, e.g., those based on approximate inverses or algebraic multigrid methods just to mention a few of them.

The rest of the paper is organized as follows. Section 2 offers a brief review of ILUPACK and the strategy to extract task-parallelism from this application. Section 3 describes the different parallelization approaches, based on either OmpSs or MPI only, and the new solution that combines both parallel programming interfaces. Section 4 analyzes the performance and scalability of the different parallel versions. Finally, Sect. 5 summarizes our work and offers several concluding remarks.

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

2 Exposing Task-Parallelism in ILUPACK

Introduction to ILUPACK. The C and Fortran routines included in ILUPACK can be leveraged to solve sparse linear systems of the form $Ax = b$ via Krylov subspace methods [18]. This library provides multilevel preconditioners that improve the numerical properties of the linear system, reducing the number of steps of the iterative solver. Concretely, the procedure obtains an efficient preconditioner from the ILU factorization of the system matrix, dropping the small entries of the factors, while relying on pivoting to bound the norm of the inverse triangular factors, yielding a numerical multilevel hierarchy of partial inverse-based approximations [5,6].

S1: Compute the preconditioner $A \rightarrow M \approx LU$	
S2: Initialize $x_0, r_0, z_0, d_0, \beta_0, \tau_0$	
S3: $k := 0$	
S4: **while** $(\tau_k > \tau_{\max})$	**Iterative PCG solve**
S5: $w_k := Ad_k$	(SPMV)
S6: $\rho_k := \beta_k / d_k^T w_k$	(DOT product)
S7: $x_{k+1} := x_k + \rho_k d_k$	(AXPY)
S8: $r_{k+1} := r_k - \rho_k w_k$	(AXPY)
S9: $z_{k+1} := M^{-1} r_{k+1} \approx U^{-1} L^{-1} r_{k+1}$	Apply preconditioner
S10: $\beta_{k+1} := r_{k+1}^T z_{k+1}$	(DOT product)
S11: $\alpha_k := \beta_{k+1} / \beta_k$	
S12: $d_{k+1} := z_{k+1} + \alpha_k d_k$	(AXPY-like)
S13: $\tau_{k+1} := \parallel r_{k+1} \parallel_2$	(2-norm)
S14: $k := k + 1$	
S15: **endwhile**	

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

For the particular case of a symmetric positive definite (s.p.d.) linear system, Fig. 1 illustrates a simplified version of the PCG solver underlying ILUPACK. The most challenging operations in this algorithm are the computation of the preconditioner (S1), before the iteration commences, and its application at each iteration (S9). We will describe in detail the task-parallelism implicit in these two operations.

Nested Dissection. Exploiting the relationship between sparse matrices and adjacency graphs, nested dissection can be recursively applied to permute a sparse matrix, yielding a collection of diagonal blocks that are linked to certain subgraphs and separators [3]. Moreover, the hierarchy of subgraphs and separators fixes the order in which the diagonal blocks have to be factorized. This process renders a TDG with the structure of a tree, where the subgraphs occupy the leaves and the separators correspond to the internal nodes. For example,

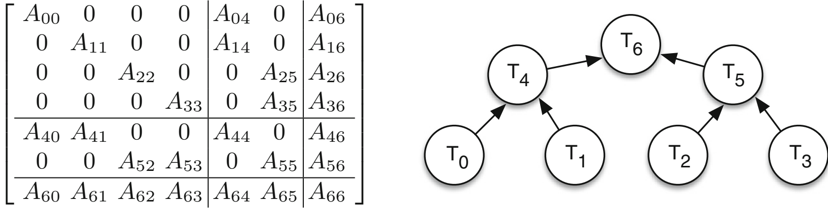

Fig. 2. Partitioning (left) and task dependency tree of the diagonal blocks (right). Task T_j is in charge of processing the diagonal block A_{jj}.

Fig. 2 (left) reflects the structure of a sparse matrix on which two nested dissection steps have been applied, yielding 4 subgraphs and 3 separators. Figure 2 (right) shows the TDG of the permuted matrix, where the edges of this directed acyclic graph define the dependencies between the diagonal blocks (tasks).

Computation of the Preconditioner. In order to improve the concurrency of this computation, the permuted matrix can be disassembled into one submatrix per leaf of the TDG. For instance, the submatrices for the graph in Fig. 2 are decomposed as

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

where

$$A_{44} = A_{44}^{0} + A_{44}^{1}, \; A_{55} = A_{55}^{2} + A_{55}^{3}, A_{66} = A_{66}^{0} + A_{66}^{1} + A_{66}^{2} + A_{66}^{3}. \quad (2)$$

Thus, the factorizations of the leading blocks of these four submatrices can proceed in parallel, while the modified blocks $A_{ij}^{0/1/2/3}$ 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.

Application of the Preconditioner. The application of the preconditioner requires the solution of two triangular systems, corresponding to the lower and the upper incomplete triangular factors. The TDG for the former triangular system presents the same structure and dependencies as that associated with the computation of the preconditioner. In the latter triangular solve, the structure is preserved but dependencies are reversed, pointing top-down from the root to the leaves. Therefore, concurrency increases/decreases as we move towards/away from the leaves.

Other Kernels in the PCG Iteration. The remaining operations of PCG conform the computation/application of the preconditioner. The matrix is disassembled following (1) and the vectors are partitioned in a conformal manner,

but the operation on vectors does not always fulfill (2). With this formulation all these computations only involve the leaves of the TDG and, therefore, can be computed fully in parallel, except for the dot products, which require an atomic addition (reduction) of the values locally computed in each leaf.

Degree of Concurrency. The number of leaves of the TDG grows exponentially with the number of nested dissection steps, so that the degree of concurrency can be easily increased by expanding additional levels. However, each dissection step introduces additional numerical levels in the computation yielding both a different TDG and a distinct preconditioner. While the numerical properties of all these preconditioners are similar, in practice the number of iterations of the PCG solver increases significantly after a few levels (8 and more) are expanded.

3 Exploiting Task-Parallelism with OmpSs and MPI

In this section, we first briefly review how to exploit the task-parallelism explicitly exposed by the TDG, using either OmpSs or MPI, to then introduce our approach that combines both parallel programming models to yield a task-parallel MPI+OmpSs solution.

3.1 Parallelization Using OmpSs

OmpSs is a task-based parallel programming model developed at Barcelona Supercomputing Center (BSC) [8,16]. At execution time, the runtime system underlying OmpSs detects data dependencies between tasks, with the help of OpenMP-like compiler directives (pragmas) annotated with clauses that indicate the task operands' directionality (input, output or input/output). OmpSs then generates a task graph during the execution, which is leveraged to schedule the tasks to the cores, exploiting the inherent task-level parallelism while fulfilling the dependencies embedded in the graph.

The opportunities to exploit task-level parallelism in ILUPACK's PCG method lie within the computations that involve the preconditioner (computation and application) as well as the vector operations. The introduction of OmpSs in the operations with the preconditioner is quite intricate, mostly due to the complexity of ILUPACK itself. Nonetheless, it is possible to create a "skeleton" structure that explicitly exposes/governs the dependencies associated with the TDG while requiring only minor modifications in the routines included in the OpenMP version of ILUPACK [1,2]. In contrast, as the sparse matrix and the vectors in the PCG iteration are disassembled conformally, according to (1), the operations on the latter can be decomposed into a number of independent vector suboperations, which are easily parallelized using OmpSs. The only exception are the dot products which, after the reduction of the subvectors local to each thread, involve an atomic addition and, therefore, a synchronization/barrier [1,2]. Although nested parallelism could be applied to optimize the operations related to each node, our experience with this technique is negative.

3.2 Parallelization with MPI

The original MPI-based parallel version of ILUPACK, introduced in [4], spawns one MPI rank per leaf (task) of the TDG, with a one-to-one static mapping between leaves and ranks. This task-rank correspondence is fixed before the preconditioner computation, by the root process, which sends the information for each leaf to the appropriate MPI rank. The same mapping is then maintained during the complete execution, for all computations and iterations, including the preconditioner computation/application and vector operations.

The operations with the preconditioner potentially transform the dependencies of the TDG into communications among MPI ranks. To reduce the number of transfers, an inner task is always mapped to one of the two MPI ranks where the two "children" tasks were mapped to. For example, consider a TDG consisting of 4 leaves mapped to 4 MPI ranks: R0–R3. Then, in order to collapse the first level when the graph is traversed bottom-up during the lower triangular system solve, ranks R0, R2 send their data to R1, R3, respectively. Next, the receivers accumulate this information with the results from their own computations, and process the tasks in the next higher level, while the senders block till the top-down traversal of the TDG during the upper triangular system solve. Following this strategy, traversing the TDG only requires a communication between "sibling" tasks/"neighbour" MPI ranks.

Disassembling the matrix and the vectors, according to (1), allows all the other computations in PCG to operate with the leaves, avoiding any communication, except for the dot operations, which require an MPI reduction (`MPI_Reduce`) to accumulate the values computed in each node.

3.3 Combining MPI+OmpSs

In general, a strong motivation for mixing OmpSs with MPI is to unleash a higher level of asynchronism, for example in order to overlap communication with computation reducign the number of global synchronizations. In this particular work, the major advantage of combining both programming models is to exploit dynamic scheduling within the cluster nodes via OmpSs.

The first step to obtain an MPI+OmpSs solution is to develop a *new* MPI version of ILUPACK where an MPI rank can handle a subtree of the TDG comprising several leaves and the related inner tasks. With this version, OmpSs can then be used to process the tasks mapped to each MPI rank, dynamically distributing the work between several OmpSs threads. For example, consider a 2-level TDG composed of one root task and two leaves to be executed on a processor with two cores. If the computational cost associated with the leaves is unbalanced, this can be tackled by expanding an additional level of the TDG, yielding a 3-level tree consisting of four leaves. Now, if the parallelization is based on MPI only, an optimal mapping of the tasks to MPI ranks requires a prior knowledge of the computational costs of the tasks. Compared with this, an OmpSs parallel version with 2 threads features a dynamic mapping of tasks to

threads that is more flexible and can use the resources more efficiently by, e.g., prioritizing the execution of the more expensive tasks.

The MPI+OmpSs version still requires an initialization where the root process distributes the data corresponding to (the leaves of) the subtrees among the MPI ranks. The MPI+OmpSs version of ILUPACK is then divided into a sequence of interleaved OmpSs and MPI stages, with the former ones computing the tasks internal to the subtrees local to the MPI ranks, and the latter requiring communication between MPI ranks. In particular, the computation of the preconditioner comprises only one stage of each type, but its application in the loop body of PCG has two OmpSs stages per iteration because the TDG is traversed twice. Figure 3 illustrates the initial distribution for a TDG with 8 leaves, together with a scheme of the execution of the two stages in the preconditioner computation. In that example, the OmpSs threads process the tasks within the bottom two levels, with no MPI communication involved. For the top two levels, the OmpSs threads remain inactive and it is the MPI ranks that are in charge of processing the tasks. The dot operations also exhibit the same two stages: On the leaves, the OmpSs threads accumulate their local subvectors, and an atomic reduction is then applied to compute the reduction inside each MPI rank. These local values are then reduced using an MPI collective primitive. The remaining vector computations of the PCG iteration operate in the bottom level only and, therefore, are computed by OmpSs threads with no MPI communication involved.

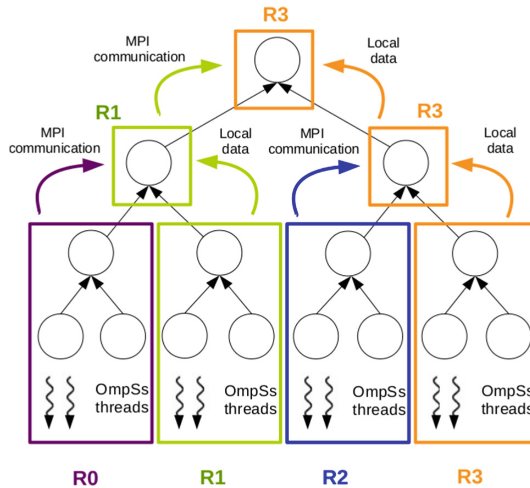

Fig. 3. Mapping of a TDG to 4 MPI ranks (R0–R3) with 2 OmpSs threads per rank.

4 Experimental Results

4.1 Setup and Preliminaries

The experiments in this section were performed using IEEE754 double-precision arithmetic on *MareNostrum*, a large-scale computing infrastructure at BSC. This platform connects 3,056 compute nodes via an Infiniband Mellanox FDR10 network. Each node contains two Intel Xeon E5-2670 processors for a total of 16 cores per server (2.6 GHz). The nodes employed in our experiments were also equipped with 64 Gbytes of DDR3 RAM.

For the experimental analysis, we employed an 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 iterate 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.

Table 1. Matrices employed in the experimental evaluation, where n_z only includes the non-zeros in the upper triangular part.

	Matrix	Dimension n	#non-zeros n_z	Density (%)
Laplace	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

In the following we analyze the performance of two parallel versions of the PCG solver in ILUPACK: one based on MPI that can handle several leaves per MPI rank, with no intervention of OmpSs (hereafter, referred to as MPI-only); and an alternative variant that combines MPI+OmpSs also capable of processing several leaves per MPI rank, but which does so via OmpSs threads internally to each node. The MPI+OmpSs code was compiled using Mercurium C/C++ (1.99.8), with the OpenMPI (1.8.1) flags `-showme:compile` and `-showme:link`. The MPI-only variant was compiled with the same version of OpenMPI. Other software included OmpSs (15.06), ILUPACK (2.4), and ParMetis[3] (4.0.2) for the graph reorderings. In the executions with the MPI-only version, we spawned one MPI rank per core (i.e., 16 per node). For MPI+OmpSs, we tested distinct combinations of MPI ranks and OmpSs threads, with the numbers of ranks multiplied by the number of threads always being equal to 16 per node.

In the following, we consider the behaviour of the iterative PCG solver only, without the preconditioner computation, because the computational cost of the

[3] http://glaros.dtc.umn.edu/gkhome/metis/parmetis/download.

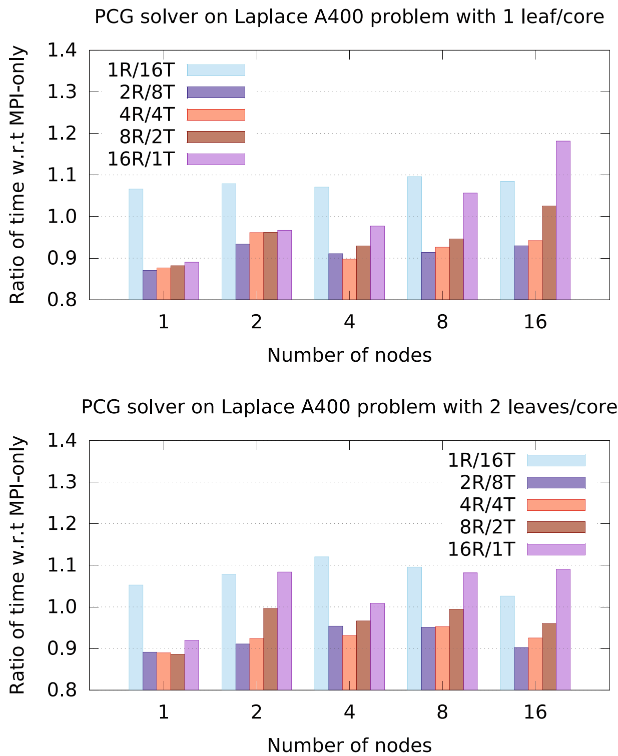

Fig. 4. Execution time per PCG iteration for the Laplace A400 problem for different configurations, using 1 leaf per core (top) and 2 leaves per core (bottom). (Color figure online)

latter is in general smaller and we observed no significant performance differences between the MPI-only and MPI+OmpSs parallel versions of this procedure. In addition, several previous experiments (for brevity, not shown here) revealed that the best performance was obtained when splitting the sparse matrix via nested dissection to generate a TDG with a number of leaves that equals or doubles the number of cores. Therefore, for simplicity, in the following we analyze only these two cases.

4.2 Analysis of Configurations

In order to assess the performance of the parallel MPI+OmpSs version of ILUPACK, we first evaluate different combinations of MPI ranks and OmpSs threads per node (configurations). Given the node target architecture, with 2 sockets/8 cores per socket, we employ 1, 2, 4, 8 or 16 MPI ranks per node and the corresponding number OmpSs threads that fill all cores per node: 16, 8, 4, 2 or 1, respectively. We will denote these configurations as 1R/16T, 2R/8T, 4T/4T, 8R/2T, and 16R/1T (#Ranks/#Threads). Figure 4 reports the ratio of execution

time of these configurations normalized with respect to the MPI-only implementation for the A400 problem, splitting the problem to obtain one leaf per core and two leaves per core. Both graphs reveal that, for almost all cases, the best option is 2R/8T, which mimics the internal socket/core architecture of the servers. Furthermore, we also notice that the extreme configurations, 1R/16T and 16R/1T, deliver the lowest performance. In the case that employs 1 rank and 16 OmpSs threads, this is due to the intersocket implicit communications. In the alternative with 16 MPI ranks and 1 thread per rank the reason is the overhead introduced by the OmpSs runtime system. In order to avoid this, when exploiting the hardware concurrency using MPI ranks only, we will not employ the OmpSs runtime system in the following.

4.3 Analysis of Scalability

We first evaluate the strong scalability of the parallel solvers. Figure 5 shows the execution time per iteration of the PCG solve for the A400 problem as the resources are increased from 16 cores/1 node to 256 cores/16 nodes. In general, as expected, there is a decrease in the iteration time as the number of cores grows. If we compare the two versions, the results demonstrate that the MPI+OmpSs variant consistently outperforms the MPI version (with no underlying OmpSs runtime system), by a margin that is around 5–10 %. Moreover, there is a slight difference between the cases with one or two leaves per core that is enlarged with the number of cores, revealing the TDG with one leaf per core as the best choice for 32 or more cores. The reason is that, as the amount of computational resources grows, the additional concurrency explicitly exposed by further splitting the computational load (sparse matrix/adjacency graph) does not compensate the overhead that is introduced for this particular (moderate) problem dimension.

The next experiment aims to provide an evaluation of weak scaling for the parallel solvers. Unfortunately, for ILUPACK's PCG solve it is not possible to

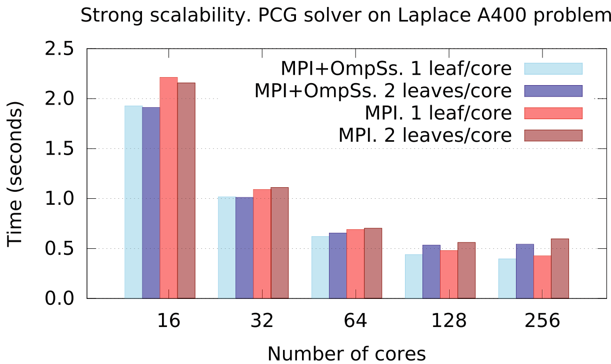

Fig. 5. Execution time per PCG iteration for the Laplace A400 problem. (Color figure online)

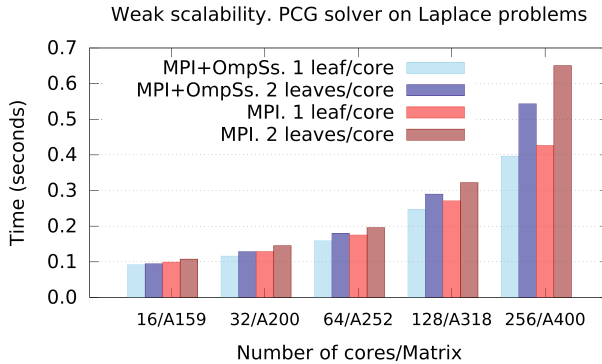

Fig. 6. Execution time per PCG iteration for different Laplace problems. (Color figure online)

generate an instance of the Laplace problem with a computational complexity that grows exactly in proportion to the number of resources. To approximate this scenario, we set the number of non-zeros of the sparse matrix (n_z) to be roughly proportional to the number of cores. However, we emphasize that n_z only offers an estimation of the computational cost, as other factors such as the fill-in/quality of the preconditioner may play a relevant role. Figure 6 reports the performance of the parallel implementations of the PCG solve (per iteration) for the different matrices in Table 1. These results show that the execution times grow with the number of cores/problem dimension. The reason is that the number of actual floating-point arithmetic operations per iteration increases faster than n_z. Comparing both implementations, the MPI+OmpSs version outperforms the MPI variant; and the difference between the cases with one or two leaves per core also grows with the number of cores.

5 Concluding Remarks

We have presented a new parallel version of the complete method underlying ILUPACK for solving symmetric positive definite linear systems on clusters of multicore processors. The approach extracts task-parallelism by splitting the sparse matrix into multiple levels, yielding a directed acyclic graph, with the form of a binary tree, where the nodes represent tasks, the arrows indicate data dependencies, and most computational work is performed in the leaf tasks. This graph is then traversed from bottom-up for the computation of the preconditioner and one of the triangular solves during its application, and top-down for the second triangular solve. In principle, the tree can be expanded into further levels to expose any number of tasks and, therefore, degree of concurrency. However, doing so yields different preconditioners and, from a certain depth, incurs into a significant overhead. In general, the best compromise is to generate up to two leaves per core, to allow the OmpSs scheduler optimize the computation. The experimental results confirm this assert for configurations with a

reduced number of nodes, where the overhead is compensated by the OmpSs optimization. For unstructured matrices, the OmpSs runtime system accelerates the computation in most scenarios, due to the irregularity of the node sizes.

The solver combines the MPI and OmpSs programming models, with the best solution corresponding to a configuration that maps one MPI rank and eight OmpSs threads per socket, mimicking the internal architecture of the cluster nodes. With these parameters, the new MPI+OmpSs version of ILUPACK outperforms the initial implementation for clusters, which was based on MPI and could only process one leaf per rank.

Acknowledgements. This work was supported by the CICYT project TIN2014-53495-R of the MINECO and FEDER, 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. Aliaga, J.I., Badia, R.M., Barreda, M., Bollhöfer, M., Dufrechou, E., Ezzatti, P., Quintana-Ortí, E.S.: Exploiting task and data parallelism in ILUPACK's preconditioned CG solver on NUMA architectures and many-core accelerators. Parallel Comput. **54**, 97–107 (2016)
2. Aliaga, J.I., Badia, R.M., Barreda, M., Bollhöfer, M., Quintana-Ortí, E.S.: Leveraging task-parallelism with OmpSs in ILUPACK's preconditioned cg method. In: 26th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD 2014), pp. 262–269 (2014)
3. Aliaga, J.I., Bollhöfer, M., Martín, A.F., Quintana-Ortí, E.S.: Exploiting thread-level parallelism in the iterative solution of sparse linear systems. Parallel Comput. **37**(3), 183–202 (2011)
4. Aliaga, J.I., Bollhöfer, M., Martín, A.F., Quintana-Ortí, E.S.: Parallelization of multilevel ILU preconditioners on distributed-memory multiprocessors. In: Jónasson, K. (ed.) PARA 2010, Part I. LNCS, vol. 7133, pp. 162–172. Springer, Heidelberg (2012)
5. Bollhöfer, M., Grote, M.J., Schenk, O.: Algebraic multilevel preconditioner for the Helmholtz equation in heterogeneous media. SIAM J. Sci. Comput. **31**(5), 3781–3805 (2009)
6. Bollhöfer, M., Saad, Y.: Multilevel preconditioners constructed from inverse-based ILUs. SIAM J. Sci. Comput. **27**(5), 1627–1650 (2006). Special issue on the 8-th Copper Mountain Conference on Iterative Methods
7. Chow, E., Patel, A.: Fine-grained parallel incomplete *lu* factorization. SIAM J. Sci. Comput. **37**(2), C169–C193 (2015)
8. Duran, A., Ferrer, R., Ayguadé, E., Badia, R.M., Labarta, J.: A proposal to extend the OpenMP tasking model with dependent tasks. Int. J. Parallel Program. **37**(3), 292–305 (2009)
9. Gaidamour, J., Hénon, P.: A parallel direct/iterative solver based on a schur complement approach. In: 11th IEEE International Conference on Computational Science and Engineering, CSE 2008, pp. 98–105. IEEE (2008)

10. George, T., Gupta, A., Sarin, V.: An empirical analysis of the performance of preconditioners for SPD systems. ACM Trans. Math. Softw. **38**(4), 24:1–24:30 (2012)
11. Hénon, P., Ramet, P., Roman, J.: PaStiX: a high-performance parallel direct solver for sparse symmetric definite systems. Parallel Comput. **28**(2), 301–321 (2002)
12. Irony, D., Shklarski, G., Toledo, S.: Parallel, fully recursive multifrontal supernodal sparse Cholesky. Future Gener. Comput. Syst. **20**(3), 425–440 (2004). Special issue: Selected numerical algorithms archive
13. Kepner, J., Gilbert, J. (eds.) Graph Algorithms in the Language of Linear Algebra. SIAM (2011)
14. Li, X.S.: An overview of SuperLU: algorithms, implementation, and user interface. ACM Trans. Math. Software **31**(3), 302–325 (2005)
15. Li, Z., Saad, Y., Sosonkina, M.: pARMS: a parallel version of the algebraic recursive multilevel solver. Numerical Lin. Alg. W. Appl. **10**, 485–509 (2003)
16. The OmpSs programming model. http://pm.bsc.es/ompss
17. Amestoy, J.K.P.R., Duff, I.S., L'Excellent, J.-Y.: A fully asynchronous multifrontal solver using distributed dynamic scheduling. SIAM J. Matrix Anal. Appl. **23**(1), 15–41 (2001)
18. Saad, Y.: Iterative Methods for Sparse Linear Systems. SIAM (2003)
19. Schenk, O., Bollhöfer, M., Römer, R.A.: On large scale diagonalization techniques for the Anderson model of localization. SIAM Rev. **50**, 91–112 (2008)
20. Schenk, O., Gärtner, K.: On fast factorization pivoting methods for symmetric indefinite systems. Electr. Trans. Num. Anal. **23**(1), 158–179 (2006)
21. Schenk, O., Wächter, A., Weiser, M.: Inertia-revealing preconditioning for large-scale nonconvex constrained optimization. SIAM J. Sci. Comput. **31**(2), 939–960 (2009)