This is a post-peer-review, pre-copyedit version of an article published in The Journal of Supercomputing. The final authenticated version is available online at: https://link.springer.com/article/10.1007/s11227-019-03100-4
https://doi.org/10.1007/s11227-019-03100-4

Iteration-Fusing Conjugate Gradient for Sparse Linear Systems with MPI+OmpSs

María Barreda · José I. Aliaga · Vicenç Beltran · Marc Casas

Received: date / Accepted: date

Abstract In this paper we target the parallel solution of sparse linear systems via iterative Krylov subspace-based method enhanced with a block-Jacobi preconditioner on a cluster of multicore processors. In order to tackle large-scale problems, we develop task-parallel implementations of the Preconditioned Conjugate Gradient (PCG) method that improve the interoperability between the MPI (message-pasing interface) and OmpSs programming models. Specifically, we progressively integrate several communication-reduction and iteration-fusing strategies into the initial code, obtaining more efficient versions of the 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 32 nodes with 24 cores each. The experimental analysis shows that the techniques described in the paper outperforms the classical method by a margin that varies between 6% and 48%, depending on the evaluation.

Keywords Sparse linear systems · Multicore processors · Distributed systems · Communication-reduction strategies · Iteration-fusing

1 Introduction

Linear systems appear in a myriad of applications, including fundamental numerical simulations of physical phenomena as well as in recent methods for data analytics [11]. When the coefficient matrix of the linear system is large and sparse, iterative methods based on Krylov subspaces, enhanced with some sort of preconditioner, often provide an efficient means to solve these type of linear algebra problems, avoiding the fill-in memory issues of their direct counterparts [15].

The conventional formulations of Krylov subspace methods (KSMs) such as conjugate gradient (CG), biconjugate gradient (BiCG), biconjugate gradient stabilized (BiCGStab) and generalized minimal residual method (GMRES) [15] present several "synchronization points" per iteration. In consequence, as technological advances widen the gap between computational performance and communication costs of computer architectures [8,12], the performance of KSMs suffers as the data dependencies and the memory-bound nature restrict the communication overhead that can be hidden [17]. In response, in the last years there has been an intensive research aimed to reduce the negative impact of synchronization bottlenecks in KSMs. The result has been a number of communication-reduction techniques, hierarchical and "enlarged" reformulations of the solvers, iteration-fusing variants, s-step methods ("communication-avoiding"), pipelined schemes ("communication-hiding"), etc. An up-to-date source of references to these variants can be found in [6].

Our work builds upon a number of previous papers that address the task-parallel implementation of KSMs on multicore architectures and clusters of multicore processors. First, the authors of [1]; proposed a parallel implemen-

María Barreda, José I. Aliaga
Dpto. de Ingeniería y Ciencia de Computadores, Universidad Jaime I, 12.071–Castellón, Spain
E-mail: {mvaya,aliaga}@uji.es

Vicenç Beltran, Marc Casas
08034–Barcelona Supercomputing Center, Barcelona, Spain.
E-mail: {vbeltran,casas}@bsc.es

tation of a CG solver, enhanced with a sophisticated ILUPACK preconditioner, that leverages MPI and OmpSs [13, 14] to improve the performance of a pure MPI-based solution; and this approach was then generalized to other types of incomplete LU (ILU)-based preconditioners and communication-reduced variants of CG in [2]. Independently, the authors of [18] presented an iteration-fusing variant of the pipelined CG [9], for multicore processors, that combines a task-parallel re-formulation of the method with a relaxation of the convergence test in order to break the strict barrier between consecutive iterations of the method.

In this paper we extend the iteration-fusing techniques to carry over to an MPI+OmpSs implementation of the pipelined CG solver for clusters of multicore processors. Specifically, our work makes the following contributions:

- We develop message-passing, task-parallel implementations of the CG method, preconditioned with a block-Jacobi (BJ) strategy, that exploit a "hybrid" programming solution which combines MPI and OmpSs. (We will refer to this method as BJCG hereafter.)
- We integrate several strategies into the classical BJCG to reduce the number of synchronization points, increase task asynchronism, and diminish the communication overhead. In particular, our message-passing task-parallel implementation of the pipelined BJCG solver reduces the number of synchronization points to only two per iteration. Furthermore, it overlaps communications and computations, making possible to execute in paralel tasks from two consecutive iterations.
- We examine in detail the communication and synchronization patterns, as well as the dependencies between tasks, appearing in these parallel implementations.
- For all these implementations, we perform a comparative analysis of their weak scalability on a cluster consisting of 32 nodes, using a 24-core socket per node.

The rest of the paper is structured as follows. Section 2 offers a brief review of the BJCG solver, and Section 3 presents an initial message-passing, task-parallel parallelization version of BJCG. In Section 4 we describe a number of variants of BJCG that increase the task asynchronism of the initial implementation. In Section 5, we analyze the performance and scalability of these different parallel versions. Finally, Section 6 summarizes our work and offers a few concluding remarks.

2 Preconditioned CG Solver

Let us consider the linear system $Ax = b$, where the coefficient matrix $A \in \mathbb{R}^{n \times n}$ is symmetric positive definite (s.p.d.) and sparse, with n_z nonzero entries; $b \in \mathbb{R}^n$ is the right-hand side vector; and $x \in \mathbb{R}^n$ is the sought-after solution vector. Figure 1 offers an algorithmic description of the iterative preconditioned CG method. The most time-consuming operations in this algorithm are the computation of the preconditioner (before the iteration commences), the sparse matrix-vector product (SpMV) (at each iteration), and the application of the preconditioner (M) (also once per iteration). The remaining operations are scalar computations or simple vector kernels such as the dot product (DOT) and AXPY-type vector updates [4].

For simplicity, in our implementation of the CG method we integrate the Jacobi preconditioner [15]. In particular, consider a decomposition of the coefficient matrix A that extracts its diagonal blocks into $D = (\hat{D}_1, \hat{D}_2, \ldots, \hat{D}_S)$, where $\hat{D}_s \in \mathbb{R}^{m_s \times m_s}$, $s = 1, 2, \ldots, S$, and $n = \sum_{s=1}^{S} m_s$. (In the classical Jacobi preconditioner, $m_s = 1$ for all $s = 1, 2, \ldots, S$, so that D only contains the diagonal entries of A.) The block-Jacobi preconditioner defined by the block-diagonal matrix $M = \text{diag}(D) \in \mathbb{R}^{n \times n}$ is particularly effective if the *structural blocks* $\hat{D}_s$ reflect the nonzero block-structure appearing in the coefficient matrix A, as it is the case for many linear systems that arise from a finite element discretization of a partial differential equation (PDE) [3]. To achieve this, we can leverage a supervariable agglomeration heuristic to determine the block diagonal structure of the block-Jacobi preconditioner [5]. This procedure captures the block structure of the coefficient matrix in the blocks of the preconditioner, with a given upper bound on the blocksize.

An additional appealing property of the block-Jacobi preconditioner is that its computation can be realized via a straight-forward extraction of the structural blocks from the coefficient matrix A, followed by an explicitly independent inversion of each one of these blocks: $M^{-1} = \text{diag}(\hat{M}_1, \hat{M}_2, \ldots, \hat{M}_S) = \text{diag}(\hat{D}_1^{-1}, \hat{D}_2^{-1}, \ldots, \hat{D}_S^{-1})$. The application of the preconditioner then boils down to a simple (dense) matrix-vector multiplication per block $\hat{M}_s$. Alternatively, one can extract the structural blocks and then decompose each into the product of two triangular factors using, for example, the Cholesky factorization [10]: $\hat{D}_s = L_s L_s^T$, where $L_s \in \mathbb{R}^{m_s \times m_s}$ is lower triangular. The application of the preconditioner then requires the solution of two triangular linear systems per block, with the factor L_s and its transpose L_s^T.

Compute preconditioner for $A \rightarrow M$
Set starting guess $x^{(0)}$
Initialize $z^{(0)}, d^{(0)}, \beta^{(0)}, \tau^{(0)}, l := 0$ (iteration count)

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

Step	Operation		Kernel
S1 :	$w^{(l)}$	$:= Ad^{(l)}$	SpMV
S2 :	$\rho^{(l)}$	$:= \beta^{(l)} / < d^{(l)}, w^{(l)} >$	DOT product
S3 :	$x^{(l+1)}$	$:= x^{(l)} + \rho^{(l)} d^{(l)}$	AXPY
S4 :	$r^{(l+1)}$	$:= r^{(l)} - \rho^{(l)} w^{(l)}$	AXPY
S5 :	$z^{(l+1)}$	$:= M^{-1} r^{(l+1)}$	Apply preconditioner
S6 :	$\beta^{(l+1)}$	$:= < z^{(l+1)}, r^{(l+1)} >$	DOT product
S7 :	$d^{(l+1)}$	$:= (\beta^{(l+1)} / \beta^{(l)}) d^{(l)} + z^{(l+1)}$	AXPY-like
S8 :	$\tau^{(l+1)}$	$:= < r^{(l+1)}, r^{(l+1)} >$	DOT product
	l	$:= l + 1$	

end while

Fig. 1 Formulation of the preconditioned CG solver annotated with 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.

For simplicity, hereafter we will drop the superindices that denote the iteration count in the variable names. Thus, for example, $x^{(l)}$ becomes x, where the latter stands for the storage space employed to keep the sequence of approximations $x^{(0)}, x^{(1)}, x^{(2)}, \ldots$, computed during the iterations.

3 Parallel BJCG Solver

3.1 Message-passing BJCG solver

In this subsection we perform a communication analysis of a basic message-passing implementation of the BJCG solver with the following considerations:

1. The parallel platform consists of np processes, denoted as P_1, P_2, ..., P_{np}. Without loss of generality, we will assume a message-passing algorithm that operates in a distributed-memory platform. (In the MPI context, the term process refers to an MPI rank.)
2. The coefficient matrix A is decoupled (i.e., partitioned) into np blocks of rows (i.e., one row-block per process):

$$A = \left[\begin{array}{c} A_1 \\ \hline A_2 \\ \hline \vdots \\ \hline A_{np} \end{array}\right],$$

   with the k-th *distribution block* $A_k \in \mathbb{R}^{n_k \times n}$ stored in P_k, and $\sum_{k=1}^{np} n_k = n$. This block-row distribution is particularly appealing when the sparse matrix is maintained in CSR (compressed sparse row) format [15].
3. Unless otherwise stated, vectors are partitioned and distributed conformally with the block-row distribution of A. For example, the residual vector r is partitioned as

$$r = [\, r_1 | r_2 | \ldots | r_{np} \,],$$

   and $r_k \in \mathbb{R}^{n_k}$ is stored in P_k.
4. The partitioning of A into np row-blocks also induces a conformal partitioning by blocks on the preconditioner:

$$M = \left[\begin{array}{c|c|c|c} M_1 & & & \\ \hline & M_2 & & \\ \hline & & \ddots & \\ \hline & & & M_{np} \end{array}\right],$$

where $M_k \in \mathbb{R}^{n_k \times n_k}$ is stored in P_k. For simplicity, we assume that each one of the distribution blocks M_k contains an integer number of structural blocks of the preconditioner $M = \text{diag}(\hat{D}_1, \hat{D}_2, \ldots, \hat{D}_S)$. In other words, the entries of the structural block $\hat{D}_s$ are all mapped to a single distribution block M_k. Note that this can always be achieved by slightly adjusting the values of the distribution block sizes n_k.

5. The scalars β, ρ, τ are replicated in all np processes.

Let us next analyze the individual operations (or computational kernels) S1–S8 comprised by the loop body of a single BJCG iteration, as shown in Figure 1.

Sparse matrix-vector product (S1) The input operands to this kernel are the coefficient matrix A, which is distributed by blocks of rows, and the vector d, which is partitioned and distributed conformally with A. Therefore, in order to compute this kernel, we first obtain a replicated copy of the distributed vector d in all processes, denoted as $d \rightarrow e$. (Vector e is the only array that is replicated in all processes. In contrast, the remaining vectors are all distributed.) After this communication stage, each process can then compute its local piece of the output vector w concurrently:

$$\mathsf{P}_k : w_k := A_k\, e.$$

This kernel thus requires collecting the distributed pieces of d into a single vector e that is replicated in all processes (in MPI, for example via an `MPI_Allgatherv`). The computation can then proceed in parallel, yielding the result w in the expected distributed state with no further communication involved. At the end, each MPI rank owns a piece of w.

DOT *products (S2, S6, S8)* The next kernel in the loop body is the DOT product S2 between the distributed vectors d and w. Here, it is easy to derive that a partial result can be first computed concurrently in each process:

$$\mathsf{P}_k : \rho_k := < d_k, w_k >,$$

after which, these intermediate values have to be reduced into a globally-replicated scalar $\rho := \beta/(\rho_1 + \rho_2 + \cdots + \rho_{np})$. The same idea applies to the DOT products in S6 and S8, yielding a total of three process synchronizations due to DOT products (in MPI, via `MPI_Allreduce`).

AXPY*(-type) vector updates (S3, S4, S7)* Consider now the AXPY kernel in S3, which involves the distributed vectors x, d and the globally-replicated scalar ρ. All processes can perform their local parts of this computation to obtain the result without any communication:

$$\mathsf{P}_k : x_k := x_k + \rho\, d_k.$$

The same applies to S4 and, given that β, β' are globally-replicated, also to S7.

Application of the preconditioner (S5) The kernel in S5, where the input preconditioner M is distributed by blocks of rows as A, and the input vector r is distributed conformally with the same matrix, can be performed concurrently in all processes, with

$$\mathsf{P}_k : z_k := M_k^{-1}\, r_k.$$

This kernel should exploit that M_k consists of multiple structural blocks. Thus, assuming for example that $M_1 = (\hat{D}_1, \hat{D}_2, \hat{D}_3, \hat{D}_4)$, with conformal partitionings $r_1 = (\hat{r}_1, \hat{r}_2, \hat{r}_3, \hat{r}_4)$, $z_1 = (\hat{z}_1, \hat{z}_2, \hat{z}_3, \hat{z}_4)$, the first process computes

$$\mathsf{P}_1 : z_1 := M_1^{-1}\, r_1 \quad \equiv \quad \hat{z}_s := \hat{D}_s^{-1}\, \hat{r}_s, \quad s = 1, 2, \ldots, 4.$$

Summary The previous elaboration exposes several insights for the message-passing BJCG solver:

1. Prior to each SpMV, we need to gather a copy of the distributed vector d within all nodes, implying a process synchronization at the beginning of each iteration.
2. Each DOT product requires a global reduction and, therefore, a process synchronization point. The loop body of the BJCG solve can be re-arranged so that S8 is pushed up next to S6. A simultaneous execution of these two reductions then decreases the number of process synchronizations, due to global reductions/DOT products, from 3 to 2 per iteration in the loop body of the BJCG solver.
3. The AXPY(-type) kernels and the application of the block-Jacobi preconditioner do not involve any sort of communication.
4. All (non-scalar) data is distributed among the processes, except for the replicated copy of vector d in e.

The reorganized algorithm and communications are summarized in Figure 2.

Compute preconditioner for $A \rightarrow M$
Set starting guess x
Initialize $z, d, \beta, \tau, l := 0$

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

Step	Operation	Kernel	Communication
	$\beta' := \beta$	–	
S1 :			
S1.1 :	$d \rightarrow e$	–	All gather
S1.2 :	$w := Ae$	SpMV	–
S2 :	$\rho := \beta / < d, w >$	DOT product	All reduce
S3 :	$x := x + \rho d$	AXPY	–
S4 :	$r := r - \rho w$	AXPY	–
S5 :	$z := M^{-1} r$	Apply preconditioner	–
S6 :	$\beta := < z, r >$	DOT product	Combined all reduce
S8 :	$\tau := < r, r >$	DOT product	
S7 :	$d := (\beta / \beta')d + z$	AXPY-like	–
	$l := l + 1$		

end while

Fig. 2 Message-passing formulation of the BJCG solver annotated with communication. Note the reorganization of the code to enable the merge of S6 and S8.

3.2 Task-parallel BJCG solver

When the target platform is a cluster of multicore processors, from the performance perspective it often pays off to expose an extra level of parallelism, which can then be exploited within each node of the cluster using, for example, OpenMP or OmpSs. This can be achieved by leveraging the *loop-parallelism* within the kernels S1–S8. Alternatively, the same goal can be pursued by explicitly dividing each kernel into a collection of finer-grain operations, or tasks, to then exploit this type of *task-parallelism*. This second option, analyzed in the *iteration-fusing CG scheme* presented in [18] and the ILU-preconditioned CG solver in [2], aims to avoid the introduction of a thread-synchronization point after each kernel, enabling a partially concurrent execution of two or more kernels, as explained next.

In the following analysis we will only consider kernels S1–S2 and S4–S7 in the loop body of the BJCG solver (see Figure 2). For simplicity, we merge the execution of S3 with that of S4; and S8 with S6. The operations in the solver are then intertwined by a series of data dependencies which, in principle, dictate a strict order of execution:

$$\underbrace{\cdots \rightarrow \mathsf{S6} \xrightarrow{\beta} \mathsf{S7}}_{\text{iteration } l-1} \xrightarrow{d} \underbrace{\mathsf{S1} \xrightarrow{w} \mathsf{S2} \xrightarrow{\rho} \mathsf{S4} \xrightarrow{r} \mathsf{S5} \xrightarrow{z} \mathsf{S6} \xrightarrow{\beta} \mathsf{S7}}_{\text{iteration } l} \xrightarrow{d} \underbrace{\mathsf{S1} \xrightarrow{w} \mathsf{S2} \xrightarrow{\rho} \cdots}_{\text{iteration } l+1}$$

(The name on top of each dependency arrow indicates the variable that generates the corresponding dependency.) However, as described next, exploiting task-parallelism enables a overlapped execution where some of these kernels can be (partially) computed concurrently (denoted with the symbol "$\|$"), breaking the strict inter-kernel barriers due to the dependencies; in particular, we aim to attain a parallel execution where S1 $\|$ S2 and S4 $\|$ S5 $\|$ S6.

Sparse matrix-vector product S1 $\|$ DOT *product* S2 For the SpMV we can partition the operands local to process P_k as

$$w_k := A_k\, e \quad \equiv \quad \left[\begin{array}{c} \tilde{w}_1 \\ \hline \tilde{w}_2 \\ \hline \vdots \\ \hline \tilde{w}_I \end{array}\right] := \left[\begin{array}{c|c|c|c} \tilde{A}_{1,1} & \tilde{A}_{1,2} & \cdots & \tilde{A}_{1,J} \\ \hline \tilde{A}_{2,1} & \tilde{A}_{2,2} & \cdots & \tilde{A}_{2,J} \\ \hline \vdots & \vdots & \ddots & \vdots \\ \hline \tilde{A}_{I,1} & \tilde{A}_{I,2} & \cdots & \tilde{A}_{I,J} \end{array}\right] \left[\begin{array}{c} e_1 \\ \hline e_2 \\ \hline \ddots \\ \hline e_J \end{array}\right].$$

Here, we can consider each one of the small SpMV operations $\tilde{A}_{i,j} e_j$ as a fine-grain task, where the specific sparsity pattern of A_k determines the dependencies between an output block $\tilde{w}_i$ and the input e. This yields a global *task dependency graph* (TDG) for SpMV that reflects the sparsity pattern of A. Consider, for example, the SpMV in Figure 3, where $I = 3$ and $J = 5$ and there are 7 blocks with nonzero entries in A_k. Now, due to the particular nonzero block structure of A_k, we note that, for example, $\tilde{w}_1 = \sum_{j=1}^{J} \tilde{A}_{1,j} e_j = \tilde{A}_{1,1} e_1 + \tilde{A}_{1,4} e_4$, yielding the dependencies $\{e_1, e_4\} \rightarrow \tilde{w}_1$. Here we consider that each one of the matrix-vector products $\tilde{A}_{1,1} e_1$ and $\tilde{A}_{1,4} e_4$ can be

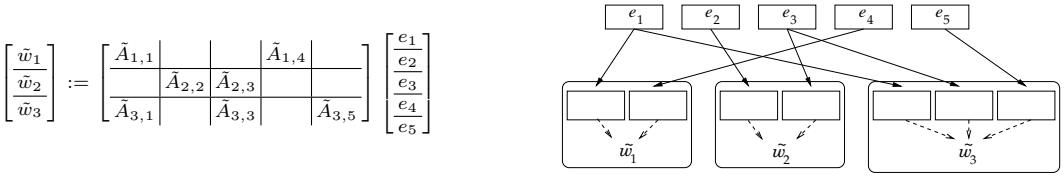


Fig. 3 Example of sparsity pattern and dependencies for SpMV. The solid lines indicate data dependencies while the dashed ones specify serialized updates.

annotated as a task, producing a partial result which finally have to be accumulated (in a serialized update) in order to obtain $\tilde{w}_1$.

For the DOT product S2, the computation local to P_k can be decomposed into I tasks, which can be calculated in parallel by partitioning the input operands d_k, w_k into I pieces, with each task producing a partial result $\tilde{\rho}_i$:

$$\rho_k := \ < d_k, w_k > \ \equiv \ \tilde{\rho}_i := \ < \tilde{d}_i, \tilde{w}_i >, \quad i = 1, 2, \ldots, I.$$

(Note the difference between the partitioning of the full replicated vector $e \in \mathbb{R}^n$ into J blocks, involved in the local SpMV, and that of the local piece of vector w as $w_k = (\tilde{w}_1 | \tilde{w}_2 | \ldots | \tilde{w}_I) \in \mathbb{R}^{n_k}$ appearing in the previous DOT product.) These partial results are then reduced to produce a single value local to the k-th node, $\rho_k := \sum_{i=1}^{I} \tilde{\rho}_i$, and these local values are next further reduced across all np nodes to produce the globally replicated scalar $\rho := \sum_{k=1}^{np} \rho_k$.

The key here is that, although there exists a strict dependency S1 $\rightarrow$ S2, by breaking these two operations into fine-grain tasks, the execution of some tasks of the second kernel can commence as soon as the corresponding results of the former one are available, yielding a partially-concurrent execution of these two tasks. Unfortunately, the global reduction required at the end of S2 imposes a task/process synchronization point that impedes to extend this idea beyond that point.

AXPY *vector update* S4 $\|$ *preconditioner application* S5 $\|$ DOT *product* S6 A similar division of the three kernels S4–S6 into fine-grain tasks permits their concurrent execution, but again a task/process synchronization appears right after S6.

AXPY *vector update* S7 *and* SpMV S1 *(subsequent iteration)* The convergence test together with the need to perform the replication $d \rightarrow e$, at the beginning of each iteration, introduces a process synchronization that prevents the parallel execution of the local tasks corresponding to these two kernels.

3.3 Implementation using MPI+OmpSs

Once we have analyzed the parallelism that the application offers, in this subsection we describe how to exploit it via a combination of two parallel programming interfaces: MPI and OmpSs. Following the OmpSs programming model [14], in our task-parallel implementations tasks are annotated with the directive `#pragma oss task`, using clauses to specify the operands' directionality as input (`in`), output (`out`) or both (`inout`). For example, the routine for a DOT product, calculating $\alpha := x^T y$, $x, y \in \mathbb{R}^q$, is annotated as

```
#pragma oss task in(x[0;n], y[0;n]) out(alpha)
ddot (int q, double *x, int incx, double *y, int incy, double alpha);
```

while, for a routine calculating the AXPY $y := y + \alpha x$, we have

```
#pragma oss task in(alpha, x[0;n]) inout(y[0;n])
daxpy (int q, double alpha, double *x, int incx, double *y, int incy);
```

As already hinted during the presentation of the message-passing task-parallel BJCG solver, the replication of vector d into e is performed across the processes using the MPI collective `MPI_Allgatherv`. To ensure that an updated version of d is fed into the MPI collective, we introduce a task barrier, using the OmpSs directive `#pragma`

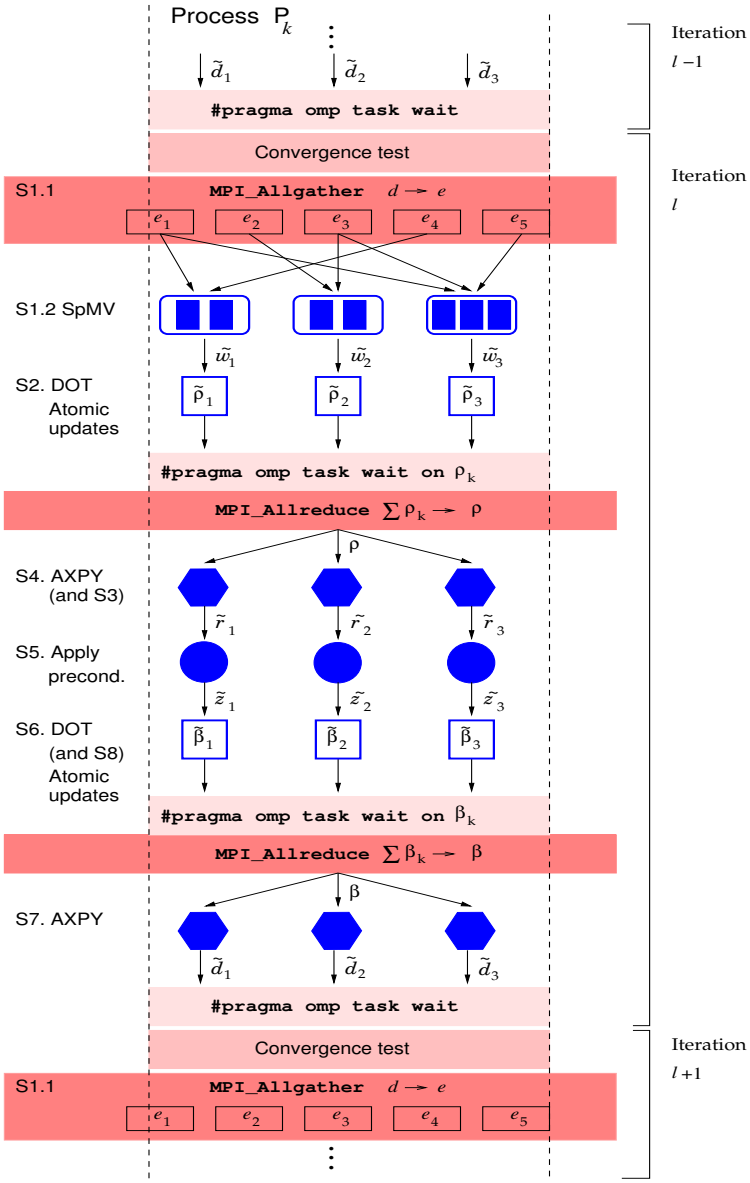


Fig. 4 Dependencies between kernels in the BJCG solver.

`oss taskwait`, before the invocation of the communication primitive. This enforces that all tasks up to that point are completed before the communication (in that process) is allowed to proceed, creating a task synchronization point which is leveraged to perform the convergence test ($\tau > \tau_{\max}$?) right after it. This is followed by an MPI synchronization across processes implicit in the MPI collective primitive.

The global reductions are realized via `MPI_Allreduce`. Similarly to the previous case, we insert a `#pragma oss taskwait on` the specific variable being reduced prior the invocation to the MPI collectives for the reduction, so that all tasks operating on that variable have been completed before the reduction across nodes can begin. Furthermore, the results from each reduction task (e.g., $\tilde{\beta}_i$ for `S8`) are accumulated into the local result (β_k) using atomic updates.

Figure 4 summarizes the previous elaboration focusing on the operations computed by process P_k, during iteration l, using a simple example with the sparse coefficient matrix defined as in Figure 3. Note the partitioning of the kernels into multiple tasks, with $I = 3$, $J = 5$.

Compute preconditioner for $A \rightarrow M$
Set starting guess x
Initialize $z, q, s, p, \alpha, \gamma, \tau, l := 0$

$r := b - Ax$, $u := M^{-1}r$, $w := Au$, $\gamma :=< r, u >$, $\delta :=< w, u >$, $\tau :=< r, r >$
while $(\tau > \tau_{\max})$

Step	Operation		Kernel	Communication
	if $(l > 0)$			
	β	$:= \gamma/\gamma'$		
	α'	$:= \alpha$		
	α	$:= \gamma/(\delta - \beta\gamma/\alpha')$		
	else			
	β	$:= 0$		
	α	$:= \gamma/\delta$		
	endif			
	γ'	$:= \gamma$		
S1 :	d	$:= M^{-1}w$	Apply preconditioner	–
S2 :				
S2.1 :	d	$\rightarrow e$	–	All gather
S2.2 :	w	$:= Ae$	SpMV	–
S3 :	z	$:= w + \beta z$	AXPY	–
S4 :	q	$:= d + \beta q$	AXPY	–
S5 :	s	$:= w + \beta s$	AXPY	–
S6 :	p	$:= u + \beta p$	AXPY	–
S7 :	x	$:= x + \alpha p$	AXPY	–
S8 :	r	$:= r - \alpha s$	AXPY	–
S9 :	u	$:= u - \alpha q$	AXPY	–
S10 :	w	$:= w - \alpha z$	AXPY	–
S11 :	γ	$:= < r, u >$	DOT product	
S12 :	δ	$:= < w, u >$	DOT product	Combined all reduce
S13 :	τ	$:= < r, r >$	DOT product	
	l	$:= l + 1$		

end while

Fig. 5 Message-passing formulation of the pipelined BJCG solver annotated with communication patterns.

4 Increasing Task Asynchronism in the BJCG Solver

The analysis of the parallel BJCG method in the previous section reveals three synchronization points per iteration. As the degree of hardware concurrency grows, these synchronizations can become more expensive than some of the computational kernels, impairing the parallel scalability of the method; see, e.g., [7,9]

In this section, we focus on the pipelined formulation of the CG solver proposed in [9] and its iteration-fusing task-parallel variant in [18], analyzing how to reduce synchronization points in a message-passing implementation.

4.1 Pipelining

Message-passing pipelined BJCG solver Figure 5 displays the pipelined BJCG solver. Let us make the same assumptions as those stated in the presentation of the message-passing BJCG solver in subsection 3.1; that is, A, M are both partitioned and distributed by blocks of rows, all vectors are partitioned/distributed conformally unless otherwise stated, and the scalars are globally replicated. Then, we can easily derive that the number of synchronizations have been reduced to only two:

- The gathering of the distributed vector y into the globally replicated copy e prior to the SpMV in S2.1 and S2.2, respectively.
- A combined global reduction for the DOT products in S11, S12, S13.

In this case, the convergence criterion can be tested right after the computation of τ, in S13.

Task-parallel pipelined BJCG solver Exploiting two levels of parallelism, by combining MPI with OpenMP or OmpSs, is also possible in the pipelined variant of the method. The dependency pattern in this algorithm can be expressed in the form of a (simplified) TDG as shown in Figure 6.

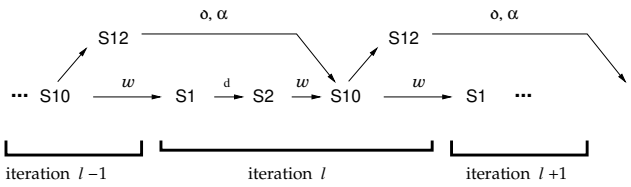


Fig. 6 TDG of the dependency pattern in the task-parallel pipelined BJCG solver.

For clarity, in this case we merge all AXPY(-type) updates into a single kernel (S10) and the three DOT products into a single one as well (S12). The purpose of dividing the kernels into fine-grain tasks is to overlap (execute partially in parallel) tasks from different kernels. In the pipelined algorithm, the preconditioner application (S1) is immediately followed by the gathering of vector d (S2.1), in preparation for the SpMV. This introduces a synchronization point which impedes a concurrent execution of these two kernels. In contrast, by dividing the SpMV itself (S2.2) into tasks, these can be executed in parallel with some of the tasks for the subsequent AXPY (S10), (as well as tasks for other AXPY vector updates). Furthermore, a task-parallelization also allows a partially-concurrent execution of the AXPY and the final DOT product (S12). In summary, S2.2 ∥ S10 ∥ S12. Unfortunately, after the combined reduction, there appears a new synchronization point implicit to the communication, which is leveraged to test the convergence criterion.

Figure 7 offer simplified TDGs of the classical and pipeline versions of the BJCG solver (left and right, respectively), exposing which kernels can proceed in parallel by dividing their execution into multiple tasks. Comparing both TDGs, we can conclude that the classical version comprisses less vector operations than the pipelined version, but the first one requires an additional synchronization step.

Implementation using MPI+OmpSs The codes for this scheme annotate tasks using OmpSs directive/clauses. Furthermore, they employ MPI collective primitives for the replication of vector d (`MPI_Allgatherv`), just before the SpMV; and the global reduction (`MPI_Allreduce`) of the scalars γ, δ, τ, at the end of the iteration. As in the task-parallel implementation of the plain CG method, an intra-process task-synchronization is enforced, via the introduction of a `#pragma oss taskwait`, before the `MPI_Allgatherv` collective. This enforces that all operations on d have been completed before its replication into e. Similarly, the insertion of directives `#pragma oss taskwait on` the three scalars being reduced before the `MPI_Allreduce` ensures the correct behaviour of these operations.

4.2 Iteration-fusing

In [18], the authors propose a variant of the pipelined scheme, named iteration-fusing CG (IFCG), that enables the concurrent execution of kernels from consecutive iterations on multicore processors. Specifically, their variant IFCG1 decomposes all kernels into fine-grain tasks, such as we described in previous sections, and relaxes the periodicity of the convergence test, to perform it only every t iterations. For those iterations where the test is not checked, on a shared-memory processor this allows that the DOT products at the end of one iteration proceed in parallel with the application of the preconditioner and SpMV at the beginning of the subsequent iteration, as there exist no dependencies between them and there is no need for MPI communication primitives; see Figure 5.

Our message-passing implementation of the pipelined CG scheme already decomposes the iteration kernels into tasks of fine granularity. At first sight, it could appear that relaxing the periodicity of the iteration test could then unleash the same "iteration-fusing effect" on a message-passing architecture. Unfortunately, the MPI primitive for the (combined) global reductions at the end of the iteration hinder this, as collective routines are blocking. Moreover, the same problem appears in the SpMV, as it is necessary to replicate vector d before this operation can commence.

An easy way to solve the previous issues would be to use tasks to implement both the computation and communication phases, relying on task dependencies to deal with inter-node and intra-node synchronizations. However, this approach cannot be efficiently implemented with the current MPI and OpenMP specifications. Concretely, MPI provides the `MPI_THREAD_MULTIPLE` mode, which supports the concurrent invocation of MPI calls from multiple threads, but this is not sufficient to efficiently support task-based programming models such as OmpMP or OmpSs-2. The main issue is that tasks are not aware of the synchronous MPI primitives, which might block not only the task but also the underlying hardware thread that runs it. Even if the MPI implementation does not rely on busy-waiting to check for operation completion and the hardware thread becomes idle, the task runtime has no means to discover

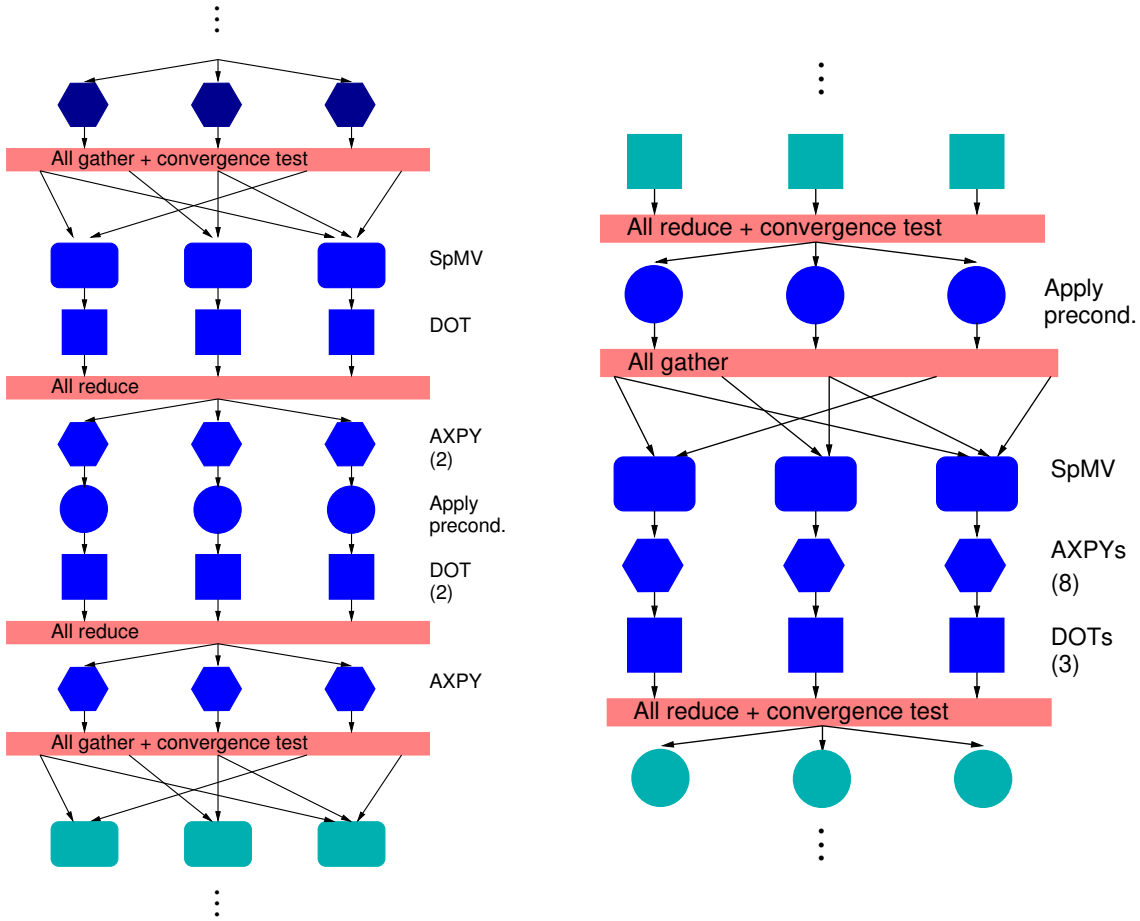
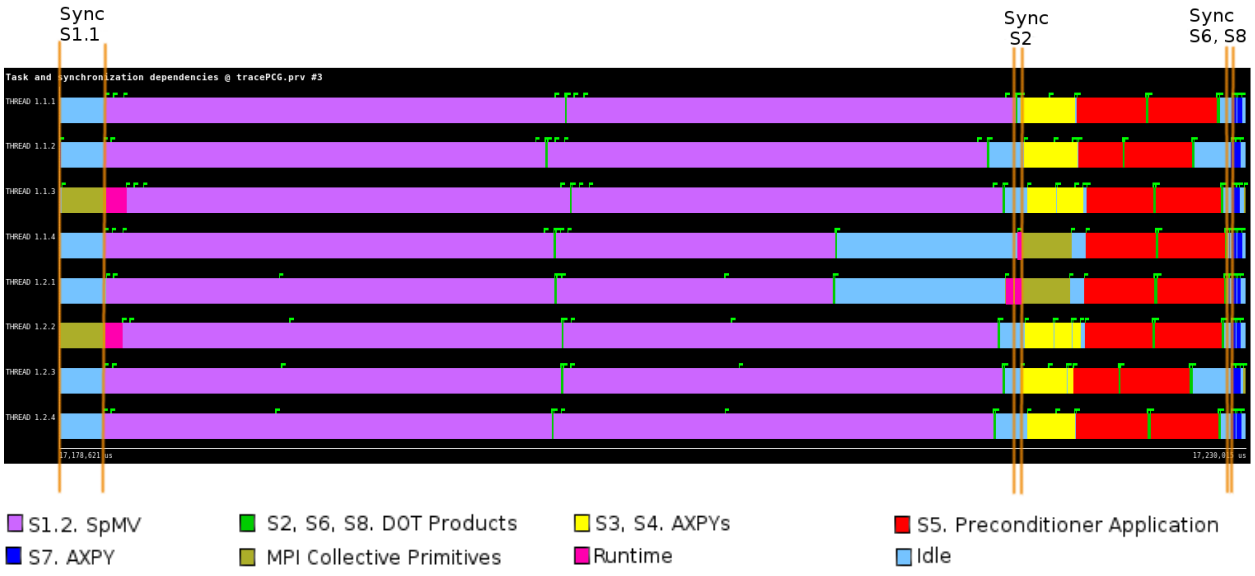


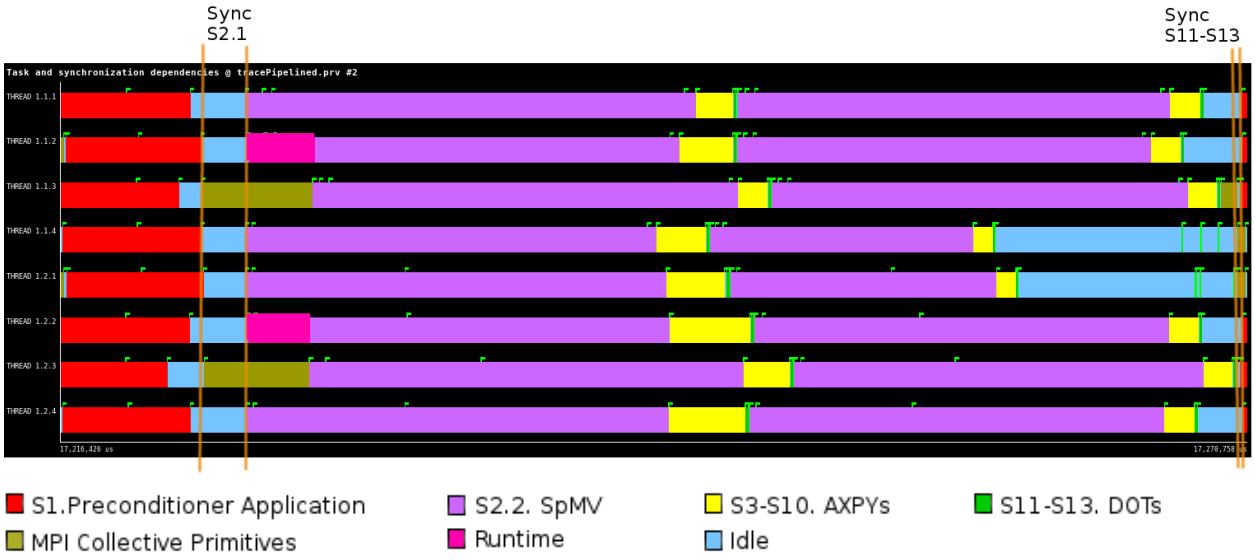
Fig. 7 Simplified dependencies between kernels in the classical BJCG solver and the pipelined variant (left and right, respectively). In both cases, a single iteration comprising the operations/kernels between two consecutive convergence tests is shown.

that the hardware thread is available without an explicit notification from the MPI side. Without this notification mechanism, if the number of blocked in-flight MPI operations reaches the number of hardware threads, the application will suffer a deadlock. With the current specification of MPI, it is the responsibility of the application developer to avoid this situation. However, this severely constrains the ability of application developers to fully benefit from task-based programming models.

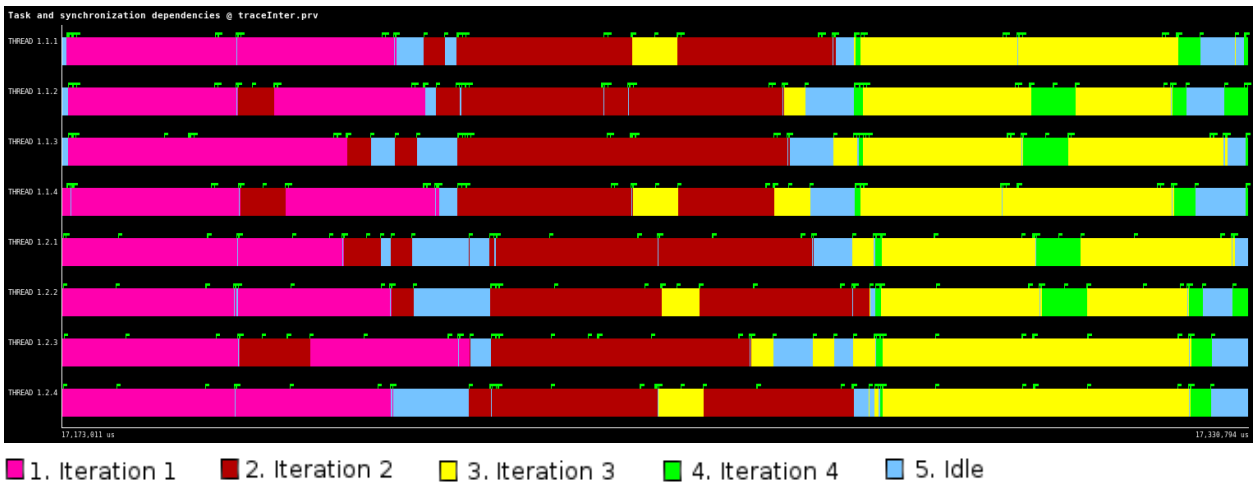
The Task-Aware MPI (TAMPI) library [16] overcomes these limitations by introducing a new MPI threading level called `MPI_TASK_MULTIPLE`. When MPI is initialized with this new threading model, each time a synchronous MPI operation blocks, instead of waiting until the operation completes, the TAMPI library notifies the runtime, so that it can schedule another task, avoiding any potential deadlock and improving CPU utilization. Using the TAMPI library, we can implement both computation and communication phases using tasks. This relies on fine-grained synchronizations to naturally overlap both phases and avoiding intra-node and inter-node global synchronization points. Therefore, using TAMPI we can break the `MPI_Allgather` routine into `MPI_Send` and `MPI_Recv` tasks, with the purpose of overlapping computation and communication. In this case, each MPI process sends its part of the vector to the others, and when an MPI process receives one part of the vector it can start the computation with that part, without the need of waiting for all the vector, as was the case without TAMPI. Furthermore, the global reductions can be also overlapped with some computations.



(a) Execution trace for **one iteration** of the **classical** task-parallel BJCG solver.



(b) Execution trace for **one iteration** of the **pipelined** task-parallel BJCG solver.



(c) Execution trace for **four iterations** of the **iteration-fusing** task-parallel BJCG solver.

Fig. 8 Execution trace of the three versions of task-parallel BJCG solver in two nodes, using 1 MPI rank per node and 4 OmpSs threads per MPI rank.

4.3 Analysis of the execution traces

Figure 8(a) provides a trace, obtained with the `Extrae+Paraver` tools,[1] that displays the execution of one iteration of the task-parallel BJCG solver using 2 MPI ranks and 4 OmpSs threads per MPI rank (Details on the parallel platform are given in section 5). We can distinguish there the sequence of operations detailed in Figure 2, consisting of the initial `MPI_Allgatherv` which replicates the contents of vector d (`S1.1`); followed by the (local part of) SpMV (`S1.2`), and a DOT product (`S2`); the `MPI_Allreduce` for the communication stage of the latter; and the final part composed of two combined AXPY(-type) vector updates (`S3` and `S4`), the preconditioner application (`S5`), two DOT products (`S6` and `S8`), with the corresponding communication, and an AXPY(-type) vector update (`S7`).

Figure 8(b) offers the trace of a single iteration of the task-parallel pipelined BJCG solver, employing the same configuration as that in Figure 8(a). In this case, the sequence commences with the application of the preconditioner (`S1`); this is directly followed by the `MPI_Allgatherv` (`S2.1`), the (local) SpMV (`S2.2`), and the AXPY(-type) vector updates (`S3`–`S10`); and finally the sequence is completed with the three combined DOT products (`S11`–`S13`) and the corresponding `MPI_Allreduce`.

These two traces expose that the SpMV and preconditioner application dominate the execution time of the iteration, but the cost of the vector operations is not negligible. Furthermore, the traces identify several synchronization points: one prior to the SpMV, to replicate vector d, and some additional ones imposed by the DOT products. Overall, this yields three synchronization points per iteration for the classical BJCG solver compared with only two for the pipelined variant. The negative impact of these synchronizations comes from the communication overhead and workload imbalance that leads to idle waits and waste of computational resources.

Figure 8(c) shows a trace for the execution of several iterations (each marked with a different color) of the iteration-fusing BJCG solver. This visual result confirms that the execution of some of the kernels of distinct iterations can be overlapped in time, confirming that the synchronization points that were visible in the classical and pipelined BJCG have been removed with this technique.

5 Experimental Evaluation

5.1 Setup

The experiments in this section employed IEEE754 double-precision arithmetic and were carried out on the *MareNostrum4* (MN4) supercomputer at *Barcelona Supercomputing Center*. This platform consists of SD530 Compute Racks with an Intel Omni-Path high performance network interconnect. Each node includes two 24-core Intel Xeon Platinum 8160 processors (2.10 GHz) and 96 Gbytes of DDR3 RAM. We performed our evaluation on up to 32 compute nodes of this infrastructure, using a single socket per node in the experimental evaluation (24 cores). The platform runs the SuSE Linux Enterprise Server operating system. The codes were compiled using Mercurium C/C++ (version 2.1.0), with MPICH (version 3.2.1). Other software included OmpSs-2 based on the Nanos 6 Runtime Library; MKL (version 2017.4) for the vector kernels and preconditioner application; and `Extrae+Paraver` (versions 3.5.1 + 4.7.2) to obtain and visualize execution traces.

For the experimental analysis, we leveraged a sparse s.p.d coefficient matrix arising from the finite-difference method for a 3D Poisson's equation with 27 stencil points. The fact that the vector involved in the SpMV kernel has to be replicated in all MPI ranks constrains the size of the largest problem that can be solved. Therefore, in order to analyze the weak scalability of the method, we permuted the original matrix, transforming it into a band matrix, where the size of the band depends on the number of nodes employed in the evaluation. Concretely, the size of the upper and lower bandwidth (BW) was set to 100×#nodes, taking values from 100 to 3,200 as the number of nodes increases from 1 to 32. With this adjustment we can maintain the size of the matrix to n=8,000,000 rows/columns, but still increase its bandwidth (and therefore, the computational workload) proportionally to the hardware resources. The sparsity degree of the coefficient matrix is not greatly affected by this modification because it still remains around 0.08%.

For the linear systems, each entry of the right-hand side vector b was initialized with the sum of the non-zero elements of the corresponding row in the matrix, in order to verify a correct solution with all entries set to ones, and the PCG iterate was started with the initial guess $x_0 \equiv 0$. The parameters that control the convergence and the FUSE steps of the iterative process (see [18]) were respectively set as `tolerance` = 1.0E-6 and `fuse` = 20.

[1] `https://tools.bsc.es/`.

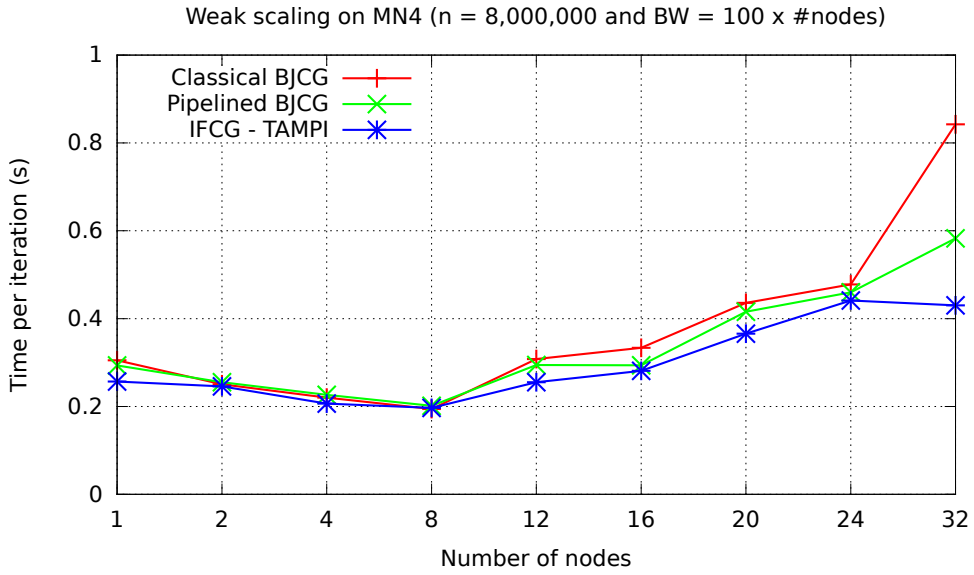


Fig. 9 Weak scaling of the message-passing task-parallel versions of the BJCG solver, using different number of nodes.

5.2 Performance analysis

We next evaluate the performance of the task-parallel BJCG solvers, focusing on the effects on performance of the synchronization-reduction techniques (pipelining and iteration-fusing), compared with the classical BJCG algorithm. Our complete experimentation included many configurations of MPI ranks and OmpSs threads. Concretely, given the node architecture, we employed 1, 2, 3, 4 or 6 MPI ranks per node, and a number of OmpSs threads that filled all the socket cores of the node; that is, 24, 12, 8, 6 or 4, respectively. For the sake of brevity, in the following we only present the results of the best configuration for each algorithm and number of nodes.

Figure 9 reports the execution time per iteration (averaged for 10 different executions) of the task-parallel BJCG solvers when we vary the number of nodes from 1 to 32, using a single socket per node. Specifically, we present a weak scaling evaluation, so that we set the number of non-zeros of the sparse matrix (n_z) to be roughly proportional to the number of cores, increasing the size of the band of the matrix, as discussed in subsection 5.1. At this point, we note that n_z only offers an estimation of the computational cost, while the communication, among other factors, may play a significant role. Therefore, although we can assume that the amount of data per node is the same, we cannot expect that the execution time will be maintained when we increase the resources proportionally to n_z. This is confirmed by the plot in Figure 9, which exhibits similar execution times for the algorithms executed in 1, 2, 4 and 8 nodes, but a considerable increase of this metric when the number of nodes augments to 12, 16, 20, 24 and 32, in principle due to the communication and synchronization overheads.

The comparison of the execution times of the distinct BJCG solvers reveals significant differences, especially as we increase the number of nodes. On the one hand, for 4 and 8 nodes (96 and 192 cores respectively), there appear small time variations between the methods, though we can observe that the application of the interoperability offers some small benefits (around 6%). On the other hand, for 12 to 24 nodes (288 to 576 cores), the difference is more notable (around 15%); and for 32 nodes (768 cores), the time difference between the classical and pipelined BJCG becomes remarkable, exposing the positive effects of exploiting the interoperability provided by the TAMPI library for this particular application. At that point, the IFCG algorithm outperforms the others by 48%.

We next analyze the impact of increasing the number of the MPI ranks, and consequently, the volume of communication, for the different algorithms. Concretely, Figure 10 displays the iteration time of the solvers when they are executed using 32 nodes, with 1, 2, 3, 4 or 6 MPI ranks per node, and adjusting the number of OmpSs threads in order to maintain the use of 24 cores per node. We observe that the execution time of the classical and pipelined BJCG grows with the number of MPI ranks while, in contrast, the execution time of IFCG is maintained or even decreases. Thus, in this case, the use of the interoperability layer of the TAMPI library counterbalances properly the increment of the communication volume when there are several MPI ranks per node.

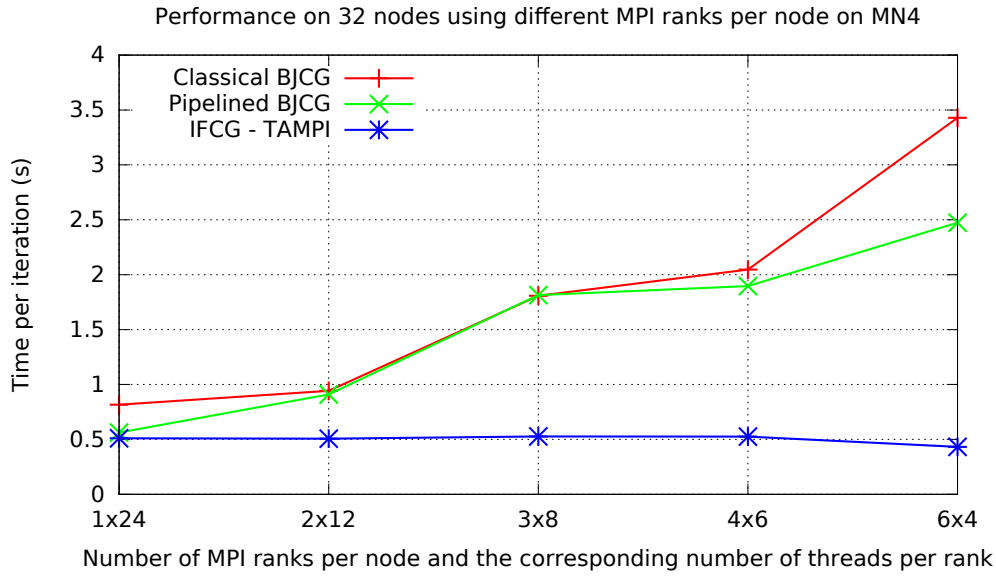


Fig. 10 Performance of the message passing task-parallel versions of the BJCG solver, using 32 nodes and different number of MPI ranks per node.

Finally, we evaluate the strong scaling of the task-parallel versions of the BJCG solver. In order to perform this analysis, the problem size and the bandwidth are fixed, respectively, to n=12,000,000 and bandwidth=400, whereas the resources are increased from 4 nodes/96 cores to 32 nodes/768 cores. The reason for starting setting the baseline to 4 nodes instead of a single node is that the maximum problem size that fits in one node is too small to obtain good scalability when the number of nodes is large, as the computation to communication ratio is rather small. Figure 11 shows the execution time per iteration of the solvers averaged for 10 different executions. In general, as expected, there is a decrease in the iteration time as the number of cores grows. If we compare the different versions, the results demonstrate that the IFCG variant consistently outperforms the other two versions, by a margin that is around 8-15%.

6 Concluding Remarks

In this paper, we have presented a message-passing task-parallel implementation of the pipelined BJCG that, in practice, reduces the number of synchronization points to only two (per iteration). To attain this goal, our iteration-fusing variant relaxes the periodicity test in the iteration while leveraging the TAMPI library to eliminate the negative blocking effects of collective communication primitive. The combination of this relaxation technique with TAMPI unleashes a concurrent execution of the tasks from two consecutive iterations which notably increases the parallel performance of the proposed message-passing task-parallel pipelined BJCG solver. Concretely, our experimental results using up to 32 nodes (and 768 cores) show a speed-up with respect to the classical BJCG and a plain pipelined version that consistently grows with the number of nodes (cores).

As part of future work, we plan to analyze alternative implementations of SpMV that avoid the replication of data and can be used to experiment with larger number of nodes or platform with a reduced amount of RAM per node.

Acknowledgements This research was partially supported by the H2020 EU FETHPC Project 671602 "INTERTWinE". The researchers from Universidad Jaume I were sponsored by project TIN2017-82972-R of the Spanish *Ministerio de Economía y Competitividad*. Maria Barreda was supported by the POSDOC-A/2017/11 project from the *Universitat Jaume I*.

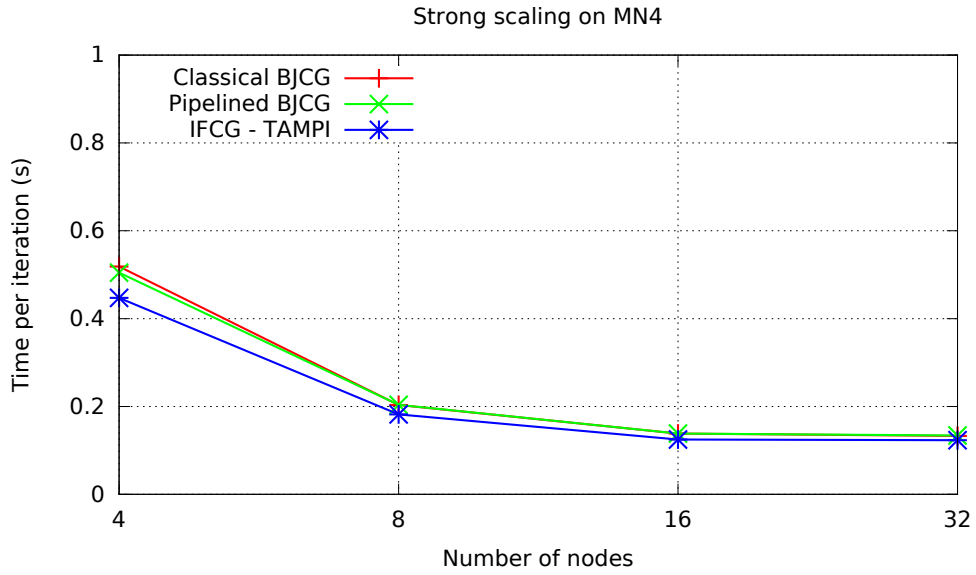


Fig. 11 Strong scaling of the message-passing task-parallel versions of the BJCG solver, using different number of nodes.

References

1. Aliaga, J.I., Barreda, M., Bollhöfer, M., Quintana-Ortí, E.S.: Exploiting task-parallelism in message-passing sparse linear system solvers using OmpSs. In: Euro-Par 2016: Parallel Processing: 22nd Int. Conf. Parallel and Distributed Computing, pp. 631–643. Springer (2016)
2. Aliaga, J.I., Barreda, M., Flegar, G., Bollhöfer, M., Quintana-Ortí, E.S.: Communication in task-parallel ILU-preconditioned CG solvers using MPI+OmpSs. Concurrency and Computation: Practice and Experience **29**(21), e4280 (2017)
3. Anzt, H., Dongarra, J., Flegar, G., Quintana-Ortí, E.S.: Batched Gauss-Jordan elimination for block-Jacobi preconditioner generation on GPUs. In: 8th Int. Workshop Programming Models & Appl. for Multicores & Manycores, PMAM, pp. 1–10 (2017). URL `http://doi.acm.org/10.1145/3026937.3026940`
4. Barrett, R., Berry, M., Chan, T.F., Demmel, J., Donato, J., Dongarra, J., Eijkhout, V., Pozo, R., Romine, C., der Vorst, H.V.: Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition. SIAM (1994)
5. Chow, E., Scott, J.: On the use of iterative methods and blocking for solving sparse triangular systems in incomplete factorization preconditioning. Tech. Rep. Technical Report RAL-P-2016-006, Rutherford Appleton Laboratory (2016)
6. Cools, S.: Numerical stability analysis of the class of communication hiding pipelined conjugate gradient methods. CoRR **abs/1804.02962** (2018). URL `http://arxiv.org/abs/1804.02962`
7. Cools, S., Vanroose, W., Yetkin, E.F., Agullo, E., Giraud, L.: On rounding error resilience, maximal attainable accuracy and parallel performance of the pipelined Conjugate Gradients method for large-scale linear systems in petsc. In: Proceedings of the Exascale Applications and Software Conference 2016, EASC '16, pp. 3:1–3:10. ACM, New York, NY, USA (2016). DOI 10.1145/2938615.2938621. URL `http://doi.acm.org/10.1145/2938615.2938621`
8. Duranton, M., De Bosschere, K., Coppens, B., Gamrat, C., Gray, M., Munk, H., Ozer, E., Vardanega, T., Zendraand, O.: The HiPEAC vision: High performance and embedded architecture and compilation (2019). URL `https://www.hipeac.net/vision/2019/`
9. Ghysels, P., Vanroose, W.: Hiding global synchronization latency in the preconditioned Conjugate Gradient algorithm. Parallel Comput. **40**(7), 224–238 (2014). DOI 10.1016/j.parco.2013.06.001. URL `http://dx.doi.org/10.1016/j.parco.2013.06.001`
10. Golub, G.H., Loan, C.F.V.: Matrix Computations, 3rd edn. The Johns Hopkins University Press, Baltimore (1996)
11. Kepner, J., Gilbert, J. (eds.): Graph Algorithms in the Language of Linear Algebra. SIAM (2011)
12. Liao, X., Lu, K., Yang, C., et al.: Moving from exascale to zettascale computing: challenges and techniques. Frontiers Inf. Technol. Electronic Eng. **19**(10), 1236–1244 (2018). DOI https://doi.org/10.1631/FITEE.1800494
13. MPI forum. `http://www.mpi-forum.org`
14. OmpSs project home page. `http://pm.bsc.es/ompss`
15. Saad, Y.: Iterative methods for sparse linear systems, 3rd edn. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA (2003)
16. Sala, K., Bellón, J., Farré, P., Teruel, X., Perez, J.M., Peña, A.J., Holmes, D., Beltran, V., J., L.: Improving the Interoperability between MPI and Task-Based Programming Models. In: EuroMPI Conference, Barcelona, Spain (2018)
17. Wulf, W.A., McKee, S.A.: Hitting the memory wall: Implications of the obvious. SIGARCH Comput. Archit. News **23**(1), 20–24 (1995). DOI 10.1145/216585.216588. URL `http://doi.acm.org/10.1145/216585.216588`

18. Zhuang, S., Casas, M.: Iteration-fusing Conjugate Gradient. In: Proceedings of the International Conference on Supercomputing, ICS '17, pp. 21:1–21:10. ACM, New York, NY, USA (2017). DOI 10.1145/3079079.3079091. URL http://doi.acm.org/10.1145/3079079.3079091