A Scalable Dense Linear System Solver for Multiple Right-Hand Sides in Data Analytics

Vassilis Kalantzis, A. Cristiano I. Malossi, Costas Bekas,
Alessandro Curioni, Efstratios Gallopoulos, and Yousef Saad

January 2018

EPrint ID: 2018.1

A scalable iterative dense linear system solver for multiple right-hand sides in data analytics

Vassilis Kalantzis[a,*], A. Cristiano I. Malossi[b], Costas Bekas[b], Alessandro Curioni[b], Efstratios Gallopoulos[c], Yousef Saad[a]

[a] *Department of Computer Science and Engineering, University of Minnesota, Minneapolis, MN 55455, USA. ({kalan019,saad}@umn.edu)*
[b] *Foundations of Cognitive Solutions, IBM Research – Zürich, Switzerland. ({acm,bek,cur}@zurich.ibm.com)*
[c] *Department of Computer Engineering and Informatics, University of Patras, 26504 Patras, Greece. (stratis@ceid.upatras.gr)*

Abstract

We describe Parallel-Projection Block Conjugate Gradient (PP-BCG), a distributed iterative solver for the solution of dense and symmetric positive definite linear systems with multiple right-hand sides. In particular, we focus on linear systems appearing in the context of stochastic estimation of the diagonal of the matrix inverse in Uncertainty Quantification. PP-BCG is based on the block Conjugate Gradient algorithm combined with Galerkin projections to accelerate the convergence rate of the solution process of the linear systems. Numerical experiments on massively parallel architectures illustrate the performance of the proposed scheme in terms of efficiency and convergence rate, as well as its effectiveness relative to the (block) Conjugate Gradient and the Cholesky-based ScaLAPACK solver. In particular, on a 4 rack BG/Q with up to 65,536 processor cores using dense matrices of order as high as 524,288 and 800 right-hand sides, PP-BCG can be 2x-3x faster than the aforementioned techniques.

1. Introduction

Several applications necessitate the solution of large linear systems with multiple right-hand sides by means of Krylov subspace iterative methods that attempt to exploit the fact that there is a single shared coefficient matrix. Indeed, an equivalent way to describe the problem is as a matrix equation. The design of iterative solvers for this problem has been an active research area since the early 1980s. The first contributions were block Conjugate Gradient-type methods [39, 40] (BCG) that exploited the convergence rate improvements offered by the generated block Krylov subspace, see [27] for a survey. A key idea that led to further improvements of Conjugate Gradient (CG) and block Krylov methods was to apply and exploit some form of "information sharing", e.g. approximating all the solutions from "seed" Krylov subspaces built from one or few right-hand sides [14, 44, 46, 47, 48]. These "seed methods" were later further refined and combined with block methods into hybrid schemes that also applied techniques such as subspace recycling, augmentation, deflation and smoothing, as well as special "global methods" that attempt to alleviate some of the computationally and memory intensive aspects of block methods, see e.g. [1, 20, 34, 41, 49].

[*] Corresponding author

Overall, methods have been designed to explore such possibilities, see e.g. [44, 48], and more recently [1, 10, 12, 19, 26, 37], from applications whose computational kernel is the solution of linear systems with multiple right-hand sides, specifically lattice QCD, electromagnetic and structures.

In this paper we consider applications in Data Analytics, and more specifically Uncertainty Quantification (UQ), where the linear systems with multiple right-hand sides result from the application of a stochastic estimator [9, 29] to approximate the diagonal of the inverse of the data covariance matrix. The latter quantity is of importance when measuring the degree of confidence in the quality of data at hand [4, 8, 45, 50, 51]. Other applications of multiple right-hand sides problems in data analytics, although not considered in this paper, can be found in statistical analysis [5, 50].

Mathematically, the problem of interest can be formulated as a sequence of linear systems (batches), each batch with multiple right-hand sides and the same symmetric positive definite (SPD) coefficient matrix $A \in \mathbb{R}^{n\times n}$, namely

$$AX^{(j)} = Z^{(j)},\ j = 1, 2, \dots, \delta, \text{ where } \quad Z^{(j)} \in \mathbb{R}^{n\times p}. \tag{1}$$

The total number of batches, denoted by δ, is typically not known a priori, and at each given moment, only a single batch $AX^{(j)} = Z^{(j)},\ 1 \le j \le \delta$, is available. For simplicity, all batches are assumed to be of equal size $p \ge 1$, i.e., each batch requires the solution of p right-hand sides. Some distinguishing features of the problem under consideration in this paper are: The matrices *i*) are very large, *ii*) dense and *iii*) symmetric positive definite and generally well-conditioned, possibly after some preprocessing, *iv*) the right-hand sides consist of Gaussian, Rademacher or standard unit vectors ([6, 43, 54]) and the total number of right-hand sides, δp, is typically much smaller than the size of the problem n, and, finally, *v*) neither the diagonal entries of A^{-1} nor the linear system solutions in (1) are sought to high accuracy [17]. Features $(i) - (v)$ will serve as the working assumptions of our discussion. Because of (i), it is essential to use distributed memory computing environments. Because of (ii) solvers can reap the advantages of BLAS-3 primitives. Because of $(iii) - (v)$, the direct $O(n^3)$ solution approaches in ScaLAPACK [11] become less practical (in certain cases A is not even explicitly available), and our interest turns in solving the linear systems in (1) by exploiting distributed memory Conjugate Gradient-type iterative linear system solvers [28].

When considering the solution of (1) in distributed computing environments, a standard approach, as proposed in [8], is to solve for each batch $AX^{(j)} = Z^{(j)},\ 1 \le j \le \delta$, by applying a "pseudo-block" CG approach (in the terminology of [7]), i.e., that is deploying standard CG but organizing the computation so as to solve for all right-hand sides simultaneously. This can improve efficiency and runtime since simultaneous Matrix-Vector products (MATVECs) are replaced with more effective Matrix-Multivector products (MATMULs). Recognizing that for diagonal estimators with large variance many batches of multiple right-hand sides might be required (that is $\delta \gg 1$), the method proposed in [33] was the first to consider the use of specialized multiple right-hand sides solvers for the solution of (1) in the context of UQ.

As problems increase in size, it becomes imperative to develop multiple right-hand sides solvers that ensure both fast convergence when solving for each batch in (1), and good scalability when implemented in massively parallel architectures, i.e., distributed computing environments with thousands of processors. The latter represents a challenging task, especially given the increasing gap between the cost of elementary arithmetic and communication operations. Therefore, it is not surprising that in spite of the sizeable literature on multiple right-hand side solvers, there is relatively little on parallel methods ([1, 7, 18, 21, 31, 36, 38, 40]) and even less ([8, 18, 33]) for problems that satisfy the stated assumptions (in particular dense and SPD matrices) holding in the application under consideration in this paper. The literature on parallel solvers is more extensive for the nonsymmetric case, see e.g. [23, 32, 35, 42, 52].

In this paper, we describe and analyze a new scheme, abbreviated as Parallel-Projection Block Conjugate Gradient (PP-BCG), for the distributed iterative solution of large and dense linear SPD systems in UQ. PP-BCG combines the block Conjugate Gradient scheme with partial recycling of Krylov subspaces using Galerkin projections [14, 44]. In particular, PP-BCG relies on the availability of a good approximation of the invariant subspace associated with the extremal eigenvalues of A generated during the application of BCG on the first batch of right-hand sides. PP-BCG shares similarities with MOD-INIT-BCG, a scheme proposed by some of the authors of this paper in [33]. The main differences of PP-BCG with MOD-INIT-BCG are the sequencing of the Galerkin projections and the fact that MOD-INIT-BCG uses a multi-sweep approach, each sweep consisting of a diminishing number of Galerkin projections. Because MOD-INIT-BCG stores no parts of the Krylov subspace, MATMULs are required to regenerate the projection subspace. On cluster computing environments, we found that the multi-sweep approach did not scale satisfactorily, and was a bottleneck. PP-BCG stores the generated Krylov subspace and projects exactly once. The Galerkin projections scheme considered in this paper reduces the cost of the projections by a factor of two relative to MOD-INIT-BCG and also appears to be less penalized by roundoff.

Specific contributions of our paper include: *i*) The development of the PP-BCG parallel numerical scheme for the solution of linear systems of the form in (1), and its parallel implementation on a message-passing environment. *ii*) A performance model for the solver and an analysis of tradeoffs in communication overheads, memory requirements, and convergence rates. *iii*) Experiments on a massively parallel architecture with up to 65,536 cores, using dense matrices of order as high as n=524,288, and 800 right-hand sides along with performance comparisons against other approaches such as the CG, BCG and the Cholesky-based ScaLAPACK solvers. We note that in the application of interest the matrices are relatively well conditioned, therefore we do not consider the use of preconditioning.

The paper is organized as follows. In Section 2 we present the proposed solver, abbreviated as PP-BCG. In Section 3 we describe its parallel implementation and give details on various design aspects as well as a cost-benefit analysis. In Section 4 we present experiments on a massively parallel architecture platform and comparisons with other solvers. Concluding remarks are provided in Section 5.

2. The PP-BCG method

This section outlines the PP-BCG scheme. The key idea is to recycle the Krylov subspace built while solving $AX^{(1)} = Z^{(1)}$ in order to improve the convergence rate of the subsequent batches $AX^{(j)} = Z^{(j)},\ j = 2, \ldots, \delta$.

2.1. Solving for the first batch of right-hand sides

Since we make no structural assumptions other than that matrix A is dense and SPD, we consider BCG to be the best choice for solving the first batch, $AX^{(1)} = Z^{(1)}$. Algorithm 1 lists the BCG algorithm for the solution of a linear system of the form $AX = Z$ where $Z \in \mathbb{R}^{n\times p}$. Matrices $X_0 \in \mathbb{R}^{n\times p}$, $R_0 = Z - AX_0$, and $P_0 = R_0$, denote the initial approximation, initial residual, and initial direction block, respectively. Throughout this section we assume that matrix A as well as multivectors X_i, R_i, T_i and P_{i-1} are distributed row-wise among the available processors. Variables

α_i and β_i denote matrices of size $p \times p$ that (in the absence of roundoff) are calculated to enforce the orthogonality conditions in BCG. Moreover, $r_i^{(k)}$ denotes the kth column of multivector R_i.

Algorithm 1 Block Conjugate Gradient (BCG).

```
 1: input : $A,\ Z,\ X_0, \text{tol},\ p$
 2: output : $X_i,\ \zeta \equiv i$
 3: $R_0 = Z - AX_0$, $P_0 = R_0$, compute $R_0^\top R_0$
 4: $i = 1$
 5: repeat
 6:     $T_i = AP_{i-1}$
 7:     $\alpha_i = (P_{i-1}^\top T_i)^{-1}(R_{i-1}^\top R_{i-1})$
 8:     $X_i = X_{i-1} + P_{i-1}\alpha_i$
 9:     $R_i = R_{i-1} - T_i\alpha_i$
10:     $\beta_i = (R_{i-1}^\top R_{i-1})^{-1}(R_i^\top R_i)$
11:     $P_i = R_i + P_{i-1}\beta_i$
12:     $i = i + 1$
13: until $\left(\max\left\{\|r_i^{(1)}\|, \ldots, \|r_i^{(p)}\|\right\} \leq \text{tol}\right)$
```

Computing the MATMUL $T_i = AP_{i-1}$ (line 6) demands communication among the processors to exchange their local sections of P_{i-1}, while computing the matrix products $P_{i-1}^\top T_i$ and $R_i^\top R_i$ in lines 7 and 10 require a reduction operation, each of size p^2. For reasons of numerical stability, the $p \times p$ matrix inverses $(R_{i-1}^\top R_{i-1})^{-1}$ and $(P_{i-1}^\top T_i)^{-1}$ are computed using the SVD decomposition (via LAPACK's `DGESVD`) [3]. The latter allows the (automatic) use of pseudoinverses in case of rank deficiency of R_{i-1}, e.g. when the right-hand sides converge at different rates. Another option, would be to use breakdown-free techniques similar to those in [2, 12, 30].

By partitioning $X_i = [x_i^{(1)}, \ldots, x_i^{(p)}]$ and $X = [x^{(1)}, \ldots, x^{(p)}]$, we have that $x_i^{(j)} \in x_0^{(j)} + \mathcal{K}_i,\ j = 1, \ldots, p$, where

$$\mathcal{K}_i \equiv \{R_0, AR_0, \ldots, A^{i-1}R_0\} \equiv \{P_0, \ldots, P_{i-1}\},$$

is the block Krylov subspace of dimension up to ip. If we order the eigenvalues of A as $0 < \lambda_1 \leq \ldots \leq \lambda_n$, the A-norm of the error of the jth right-hand side after $i-1$ BCG iterations satisfies the inequality $\dfrac{\|x_i^{(j)} - x^{(j)}\|_A}{\|x_0^{(j)} - x^{(j)}\|_A} \leq 2\left(\dfrac{\sqrt{\kappa_A} - 1}{\sqrt{\kappa_A} + 1}\right)^i$, where $\kappa_A = \lambda_n/\lambda_p$ [39]. Numerically, therefore, BCG has faster convergence than CG, since the corresponding value of κ_A for the latter is larger, namely $\kappa_A = \lambda_n/\lambda_1$.

The BCG algorithm can be applied to the solution of any subsequent batch $AX^{(j)} = Z^{(j)},\ j = 2, \ldots, \delta$. However, this does not entail any information exchange between the batch solves and therefore, any computational effort that is invested in solving $AX^{(1)} = Z^{(1)}$ is not reused. We next describe a mechanism to enable information reuse across batches.

2.2. Initialization by modified Galerkin projections

Let ζ denote the total number of iterations made by BCG during the solution process of the "seed" system $AX^{(1)} = Z^{(1)}$, and let matrices $P_{i-1},\ T_i,\ i = 1, \ldots, \hat{\zeta}\ (\hat{\zeta} \leq \zeta)$ be explicitly stored and distributed without overlap among the available set of processors. In addition, let matrices $\alpha_i,\ \beta_i$, and $P_{i-1}^\top T_i,\ i = 1, \ldots, \hat{\zeta}$, be replicated in all available processors.

The Krylov subspace generated during the solution process of $AX^{(1)} = Z^{(1)}$ can be exploited so as to improve the convergence rate of BCG applied on any subsequent batch $AX^{(j)} = Z^{(j)},\ j \geq 2$, by obtaining a non-trivial initial approximation of $X^{(j)}$. This initial approximation is obtained by means of Galerkin projections [14, 15, 46, 48], i.e., by projecting (deflating) the initial residual of $AX^{(j)} = Z^{(j)}$ to the orthogonal complement of the subspace defined by direction blocks $P_{i-1},\ i = 1, \ldots, \hat{\zeta}$ (one at a time) using oblique projections. In exact arithmetic, this approach essentially generates an initial approximation of $X^{(j)}$ such that the corresponding residual is orthogonal to the Krylov subspace $\mathcal{K}_{\hat{\zeta}}$ generated while solving $AX^{(1)} = Z^{(1)}$. If the Krylov subspace $\mathcal{K}_{\hat{\zeta}}$ has developed good approximations of the eigenvectors associated with the few extremal eigenvalues of A, then faster convergence is expected when applying BCG to $AX^{(j)} = Z^{(j)}$ due to the reduced effective condition number.

The practical difficulty with the above approach is that it is based on the A-orthogonality of direction blocks $P_{i-1},\ i = 1, \ldots, \hat{\zeta}$, a property that does not hold in practice due to finite precision arithmetic. This was recognized in [1] where it was proposed to use periodic orthogonalization of the projection subspace formed by a Lanczos-type procedure, and [33] where it was proposed to perform the Galerkin projections twice or more, each time applied on a diminishing number of direction blocks.

We propose a modified Galerkin projection scheme that tries to alleviate the effects of finite precision arithmetic in Galerkin projections without deploying orthogonalization or deflating each direction block $P_{i-1},\ i = 1, \ldots, \hat{\zeta}$, more than once. More specifically, we consider enhancing the convergence rate of BCG applied on any subsequent batch $AX^{(j)} = Z^{(j)},\ j \geq 2$ by computing a non-trivial initial approximation through a series of modified (reverse) Galerkin projections:

$$\hat{X}^{(j)}_{\hat{\zeta}-i+1} = \hat{X}^{(j)}_{\hat{\zeta}-i} + P_{i-1}(P_{i-1}^\top T_i)^{-1} P_{i-1}^\top \hat{R}^{(j)}_{\hat{\zeta}-i},\ i = \hat{\zeta} : -1 : 1, \tag{2}$$

$$\hat{R}^{(j)}_{\hat{\zeta}-i+1} = (I - T_i(P_{i-1}^\top T_i)^{-1} P_{i-1}^\top)\hat{R}^{(j)}_{\hat{\zeta}-i}, \quad i = \hat{\zeta} : -1 : 1, \tag{3}$$

where initially $\hat{X}^{(j)}_0 = 0$ and $\hat{R}^{(j)}_0 = Z^{(j)}$.

Algorithm 2 (GALPROJ) sketches the modified Galerkin projections procedure. Since each processor has access only to a certain part of P_{i-1} and $T_i,\ i = 1, \ldots, \hat{\zeta}$, a reduction operation of size $p \times p$ is necessary to form $P_{i-1}^\top \hat{R}^{(j)}_{\hat{\zeta}-i}$ (line 5 of GALPROJ). After computing $H \in \mathbb{R}^{p\times p}$, the update of $\hat{X}^{(j)}_{\hat{\zeta}-i}$ and $\hat{R}^{(j)}_{\hat{\zeta}-i}$ is performed locally in each processor. Overall, each iteration of GALPROJ requires $O(np^2)$ floating-point arithmetic operations.

From a distributed memory implementation viewpoint, small values of p lead to reductions that are dominated by latency. To improve performance, it is possible to deflate τ >1 direction blocks $P_i, \ldots, P_{i-\tau+1}$ simultaneously, this way increasing the granularity of each reduction step while reducing the initialization time of multiple reductions into a single one of size τp^2. This approach, however, requires the explicit formation of the matrix product $\mathcal{Z}{=}[P_i, \ldots, P_{i-\tau+1}]^\top A[P_i, \ldots, P_{i-\tau+1}]$. For sufficiently small values of τ, this matrix product can be approximated by its on-diagonal block part $P_i^\top T_i, \ldots, P_{i-\tau+1}^\top T_{i-\tau+1}$ which is already computed by Algorithm 1. Indeed, in exact arithmetic the direction blocks are A-orthogonal, and in finite precision arithmetic this property might also hold locally for moderate values of τ.

Algorithm 2 The Galerkin projections scheme (GalProj).

```
1: input : $\{T_i\}_{i=1}^{i=\hat{\zeta}}$, $\{P_{i-1}\}_{i=1}^{i=\hat{\zeta}}$, $\hat{X}_0^{(j)}$, $\hat{R}_0^{(j)}$, $\hat{\zeta}$
2: output : $\hat{X}_{\hat{\zeta}}^{(j)}$
3: $i = \hat{\zeta}$
4: repeat
5:     $H = (P_{i-1}^\top T_i)^{-1}(P_{i-1}^\top \hat{R}_{\hat{\zeta}-i}^{(j)})$
6:     $\hat{X}_{\hat{\zeta}-i+1}^{(j)} = \hat{X}_{\hat{\zeta}-i}^{(j)} + P_{i-1}H$
7:     $\hat{R}_{\hat{\zeta}-i+1}^{(j)} = \hat{R}_{\hat{\zeta}-i}^{(j)} - T_i H$
8:     $i = i - 1$
9: until $(i == 0)$
```

2.3. Analysis of deflation by modified Galerkin projections

Consider the deflation of the direction blocks $P_0, \ldots, P_{\hat{\zeta}-1}$ (in this order) from the initial residual $\hat{R}_0^{(j)}$ of some batch $AX^{(j)} = Z^{(j)},\ j \geq 2$. During the ith projection step,

$$\Phi_i = I - AP_{i-1}(P_{i-1}^\top AP_{i-1})^{-1}P_{i-1}^\top \tag{4}$$

projects on the orthogonal complement of the subspace defined by P_{i-1}. If we let

$$\Phi = \Pi_{i=0}^{\hat{\zeta}-1}\Phi_i, \tag{5}$$

then, since in exact arithmetic $P_0, \ldots, P_{\hat{\zeta}-1}$ are A-orthogonal, we get:

$$\Phi\hat{R}_0^{(j)} \perp \{P_0, \ldots, P_{\hat{\zeta}-1}\}, \tag{6}$$

and $\hat{R}_{\hat{\zeta}}^{(j)} = \Phi\hat{R}_0^{(j)}$ becomes orthogonal to any invariant subspace captured in the Krylov subspace $\mathcal{K}_{\hat{\zeta}} = \{P_0, \ldots, P_{\hat{\zeta}-1}\}$.

In finite precision arithmetic, the order of deflation of the direction blocks $P_0, \ldots, P_{\hat{\zeta}-1}$ can have a large impact. To see this, let $\hat{R}_i^{(j)}$ satisfy $\hat{R}_i^{(j)} \perp \{P_0, \ldots, P_{i-1}\}$. At the next step we compute $\hat{R}_{i+1}^{(j)} = \Phi_i \hat{R}_i^{(j)}$ and thus

$$\hat{R}_{i+1}^{(j)} \perp \Big\{P_i\Big\},\ \hat{R}_{i+1}^{(j)} \in \Big\{\hat{R}_i^{(j)}\Big\} + \Big\{AP_i\Big\}.$$

If $\hat{R}_{i+1}^{(j)}$ is to remain orthogonal to $\{P_0, \ldots, P_{i-1}\}$, we also need $P_\xi^\top AP_i = 0$ for all $\xi = 0, \ldots, i-1$, which does not generally hold in finite precision arithmetic. As a result, each time we deflate P_i, components from $P_0, \ldots, P_{i-1}$ re-emerge in $\hat{R}_{i+1}^{(j)}$. To ease this effect, we choose to deflate the direction blocks $P_0, \ldots, P_{\hat{\zeta}-1}$ in a reverse manner, i.e., as in (2) and (3). While reverse deflation does not eliminate the finite precision effects, in practice leads to initial approximations whose corresponding residual is typically weaker in the directions associated with the previously deflated direction blocks.

Figure 1 illustrates a numerical comparison between the two above deflation strategies for a diagonal test matrix with elements $A_{kk} = k/10^4,\ k = 1, 2, \ldots, 10^4$, where we set $p = 1$ and $\delta = 2$.

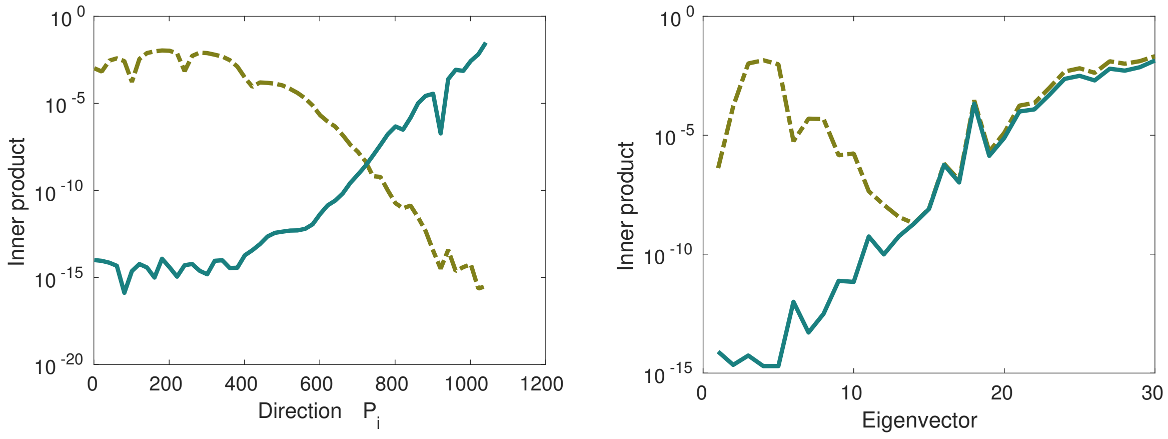

Figure 1: A comparison of two different deflation orderings for a diagonal matrix with elements $A_{kk} = k/10^4,\ k = 1, 2, \dots, 10^4$, where $p = 1$ and $\delta = 2$. Galerkin projections are performed in the natural order (dashed line) and reverse order (solid line). Left: Value of $\|P_{i-1}(P_{i-1}^\top P_{i-1})^{-1}P_{i-1}^\top \hat{R}_\zeta^{(2)}\|$ for $i = 1, \dots, \zeta$. Right: Inner product between $\hat{R}_\zeta^{(2)}$ and the eigenvectors associated with the thirty algebraically smallest eigenvalues of A.

We let CG applied to $AX^{(1)} = Z^{(1)}$ perform $\zeta = 1048$ iterations and then call GalProj to compute an initial approximation for $AX^{(2)} = Z^{(2)}$ by setting $\hat{\zeta}=\zeta$. The left subfigure plots the norm of the orthogonal projection of $\hat{R}_{\hat{\zeta}}^{(2)}$ along each direction block $P_{i-1},\ i = 1, \dots, \hat{\zeta}$, after the Galerkin projections procedure is implemented in the natural order (dashed line) and reverse order (solid line). In contrast with the reverse order implementation, the standard order of deflation leads to a residual $\hat{R}_{\hat{\zeta}}^{(2)}$ in which contributions from the earlier direction blocks have re-emerged. The right subfigure shows the inner product between $\hat{R}_{\hat{\zeta}}^{(2)}$ and the eigenvectors associated with the thirty lowest eigenvalues of A for the natural order and reverse order. Implementing the Galerkin projections in a reverse order, as in GalProj, leads to an initial approximation $\hat{X}_{\hat{\zeta}}^{(2)}$ for which $\hat{R}_{\hat{\zeta}}^{(2)}$ is closer to being orthogonal to the eigenvectors associated with the few smallest eigenvalues of A. As a result, faster convergence is expected when the associated initial approximation $\hat{X}_{\hat{\zeta}}^{(2)}$ is used by CG to solve $AX^{(2)} = Z^{(2)}$. We will return to this topic in Section 4.

The overhead involved in the deflation scheme presented in this section requires the storage of two sequences of multivectors, T_i and $P_{i-1},\ i = 1, \dots, \hat{\zeta}$. Variable $\hat{\zeta}$ can be set before-hand according to the amount of system memory being available to the application.

2.4. Galerkin projections under limited memory scenarios

We now consider a modification of GalProj to reduce the memory requirements of PP-BCG. In particular, we show that it is possible to trade the necessity to store the direction blocks $P_0, \dots, P_{\hat{\zeta}-1}$ with an additional number of floating-point operations.

More specifically, assume that during the application of BCG to $AX^{(1)} = Z^{(1)}$ we only store matrices $T_i,\ \alpha_i,\ \beta_i,\ i = 1, \dots, \hat{\zeta}$, as well as matrices $R_{\hat{\zeta}},\ P_{\hat{\zeta}}$. Then, by exploiting equation $P_{\hat{\zeta}} = R_{\hat{\zeta}} + P_{\hat{\zeta}-1}\beta_{\hat{\zeta}}$ in BCG we can recover $P_{\hat{\zeta}-1}$ as $P_{\hat{\zeta}-1} = (P_{\hat{\zeta}} - R_{\hat{\zeta}})\beta_{\hat{\zeta}}^{-1}$. Generalizing the above concept, each direction block $P_{i-1},\ i=\hat{\zeta},\dots,1$ can be generated on-the-fly by the following set of equations:

$$P_i = (P_{i+1}-R_{i+1})\beta_{i+1}^{-1},\ i=\hat{\zeta}-1,\dots,0, \tag{7}$$

$$R_i = R_{i+1}+T_{i+1}\alpha_{i+1},\quad i=\hat{\zeta}-1,\dots,0. \tag{8}$$

At each step i we only need access to matrices R_{i+1} and P_{i+1}, that have been already generated and temporarily stored from the previous step. Moreover, all computations in (7) and (8) are trivially parallel since a local copy of the $p\times p$ matrices $\alpha_i,\ \beta_i,\ i=1,\dots,\hat{\zeta}$, is already available at each processor. Overall, the necessity to store $P_0,\dots,P_{\hat{\zeta}-1}$ is replaced by the need to perform an additional $O(n\hat{\zeta}p^2)$ floating-point arithmetic operations. This computational cost will be distributed among the available set of processors.

In addition, we note that (7) is derived by assuming exact arithmetic during the application of BCG to $AX^{(1)}=Z^{(1)}$. In practice, β_{i+1} can become singular or nearly so and pseudoinverses might have to be used instead.

2.5. The complete distributed scheme

Algorithm 3 PP-BCG.

```
1: input : $A,\ \mathrm{tol}_1,\ \mathrm{tol}_2,\ p,\ \hat{\zeta}$
2:                                                          ▷ Solve $AX^{(1)}=Z^{(1)}$
3: $X^{(1)}=$ BCG$(A,\ Z_0^{(1)},\ X_0^{(1)},\ p,\ \mathrm{tol}_1)$
4:                                                          ▷ For any $AX^{(j)}=Z^{(j)}$
5: for $j=2,\dots,\delta$ do
6:     $\hat{X}_{\hat{\zeta}}^{(j)}=$ GalProj$(A,\ \hat{X}_0^{(j)}\equiv 0,\ Z^{(j)},\ \hat{\zeta})$
7:     $X^{(j)}=$ BCG$(A,\ Z^{(j)},\ \hat{X}_{\hat{\zeta}}^{(j)},\ p,\ \mathrm{tol}_2)$
8: end for
```

The complete PP-BCG scheme is described in Algorithm 3. For each batch $AX^{(j)}=Z^{(j)},\ j\geq 2$, PP-BCG generates an initial approximation by exploiting the information stored when solving $AX^{(1)}=Z^{(1)}$ by calling GalProj (line 6). This initial approximation is then passed to BCG (line 7). Note that tol_1, the tolerance set for the solution of $AX^{(1)}=Z^{(1)}$, can be different from the general tolerance tol_2 set for the solution of the subsequent batches of right-hand sides. This allows us to generate a larger and thus richer Krylov subspace which could lead to a more efficient initial approximation for the unsolved batches $AX^{(j)}=Z^{(j)},\ j\geq 2$.

3. Distributed memory implementation

3.1. Implementation in 2-D processor grids

We next describe the implementation of PP-BCG on one of the dominant parallel processing paradigms, that is a distributed memory message passing system using the Message Passing Interface (MPI) standard [53]. We assume a 2-D grid of $G=M\times K$ processors, where each processor is labeled by its row and column position on the grid. We can then write matrices A and $Z^{(j)},\ j=1,\dots,\delta$, as

$$A = \begin{bmatrix} A_{11} & A_{12} & \cdots & A_{1K} \\ A_{21} & A_{22} & \cdots & A_{2K} \\ \vdots & \vdots & \ddots & \vdots \\ A_{M1} & A_{M2} & \cdots & A_{MK} \end{bmatrix}, \quad Z^{(j)} = \begin{bmatrix} Z_1^{(j)} \\ Z_2^{(j)} \\ \vdots \\ Z_K^{(j)} \end{bmatrix}$$

where submatrix A_{IJ} is assigned to processor (I, J) and $Z_J^{(j)}$ is assigned to the Jth processor of the first row of the 2-D processor grid (equivalently, processor $(1, J)$). Here we assume that $I = 1, \dots, M$ and $J = 1, \dots, K$.

Let each processor belong to a row group and to a column group on the 2-D processor grid. This means that now, except for the global communicator, there are $M + K$ additional communicators which correspond to each different row and column of processors of the 2-D processor grid. Let matrix $P \in \mathbb{R}^{n \times p}$ be distributed among the K processors of the first row of the 2-D processor grid. Then, the MATMUL $T = AP$ can be accomplished in parallel as follows (see also Figure 2):

1. Each processor in first row holding block P_J broadcasts it along its column using the column communicator.
2. Each processor performs the product $T_{IJ} = A_{IJ} P_J$.
3. Each processor in a row performs a reduction operation using the row communicator.
4. Each processor of the first column (root processor of the corresponding row communicator) distributes its local result to the corresponding processor of the first row of the processor grid.

The communication pattern of the distributed MATMUL enables collective operations on 1-D processor grids of size at most $\max\{M, K\}$ plus some point-to-point communication. Except A, all multivectors are distributed row-wise among the K processors of the first row of the 2-D processor grid and thus for all operations of PP-BCG except the MATMUL, e.g., block inner products or block AXPY operations, only this subset of processors of the 2-D processor grid is active.

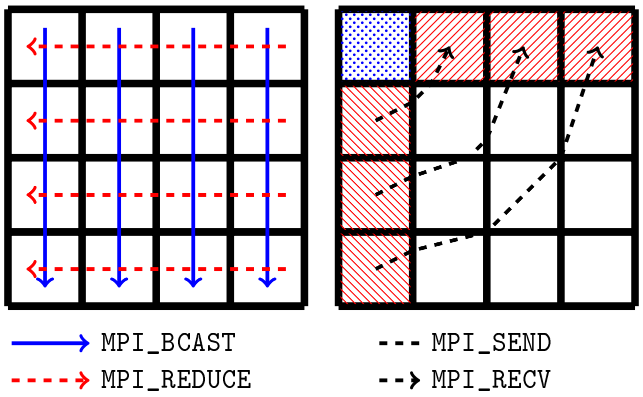

Figure 2: Schematic sketch of the communication pattern of MATMUL for a 4×4 2-D processor grid. Left: Broadcast of local P_J along the columns and reduction of the local products along the rows. Right: Point-to-point communication of the distributed MATMUL to the processors lying on the first row.

Table 1 shows the total number of floating-point scalars that need to be stored in the system memory of each processor during each phase of PP-BCG. The quantity mem(A_{IJ}) denotes the number of floating-point scalars required by the (I,J) processor to store its local portion of A. For dense unstructured matrices, this quantity runs at $\text{mem}(A_{IJ}) = n_I n_J$, where n_I and n_J denote the number of rows and columns of matrix A assigned to the (I,J) processor. When memory resources are limited, we have the option to store matrices T_i only and locally generate P_{i-1} on-the-fly (as described in subsection 2.4). This approach is denoted by the '*' superscript.

Table 1: Memory complexity per processor and phase. GalProj* denotes the limited memory version of GalProj described in Section 2.4.

Phase	**Memory requirements**
BCG (Alg. 1)	$\text{mem}(A_{IJ}) + 4n_J p$
GalProj (Alg. 2)	$2n_J p + 2n_J \hat{\zeta} p$
GalProj*	$4n_J p + n_J \hat{\zeta} p$

Table 2: Computational complexity per processor and iteration. GalProj* denotes the limited memory version of GalProj described in Section 2.4.

Phase	**Computational complexity**
BCG (Alg.1)	$\text{comp}(AP) + 6n_J p^2 + 3n_J p + 4n_J p^2$
GalProj (Alg. 2)	$2(2n_J p^2 + n_J p) + 2n_J p^2$
GalProj*	$4(2n_J p^2 + n_J p) + 2n_J p^2$

Table 2 shows the per processor computational complexity (per iteration) for all different phases of PP-BCG. The quantity $\text{comp}(AP)$ denotes the computational complexity associated with the computation of the MATMUL $T_{IJ} = A_{IJ} P_J$. For dense unstructured SPD matrices, this quantity runs at $\text{comp}(AP) = n_I p(2n_J - 1)$ floating-point arithmetic operations. In contrast, the computational cost of all other operations at any other phase of PP-BCG runs at $O(n_J p^2)$ floating-point arithmetic operations.

3.2. Communication cost of information sharing

We next model the communication cost of GalProj. For the purposes of our analysis, the communication cost between any two MPI processes is approximated by the following linear model (see e.g [13, 22])

$$t_{\text{comm}} = \ell + \mu q, \tag{9}$$

where ℓ is the startup cost (latency), μ is the message size (measured in terms of floating-point scalars) and q is the per-scalar transmission rate (bandwidth). Typically, ℓ is several orders of magnitude larger than q. We will assume that each processor can send/receive data only to/from one other processor at any given moment. Since each row of the 2-D processor grid has its own communicator, the K processors lying on the first row communicate independently from the rest processors of the 2-D grid of processors.

Each time we deflate one (or more) of the direction block(s) $P_0, \ldots, P_{\hat{\zeta}-1}$ from the initial residual of a new batch of p right-hand sides $AX^{(j)} = Z^{(j)}$, we must perform an `MPI_Allreduce` collective operation (line 5 in GalProj). The `MPI_Allreduce` operation can be viewed as a `MPI_Reduce`-`MPI_Broadcast` pair, and, assuming that the processors are positioned as in a binary tree network

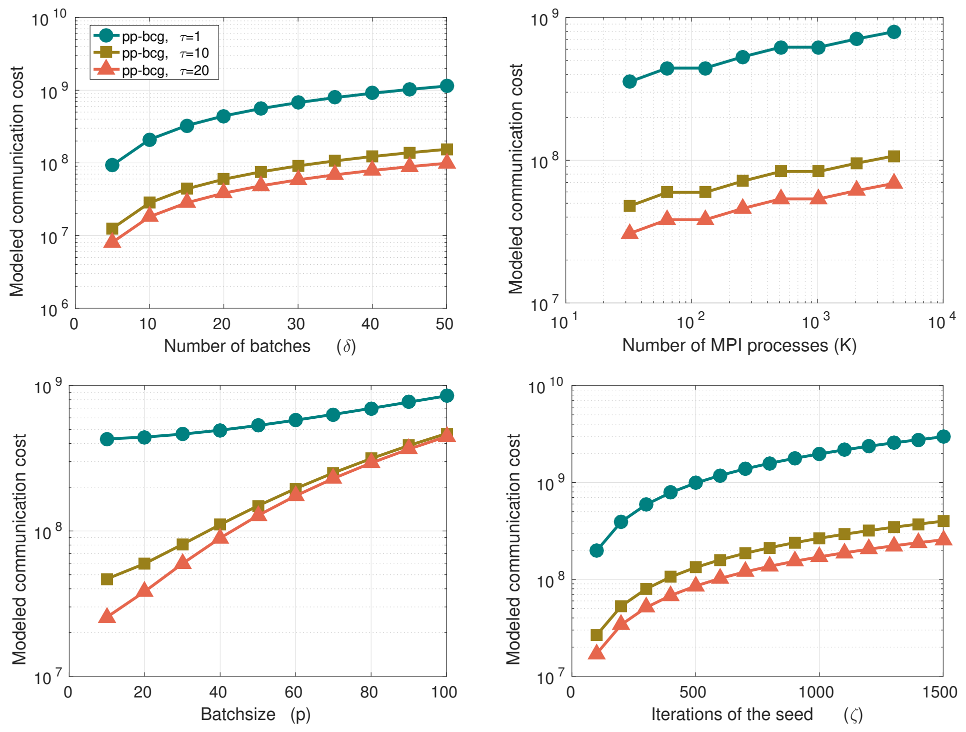

Figure 3: Modeled communication cost to obtain an initial approximation for a batch of p right-hand sides in PP-BCG. "●", "■", and "▲" PP-BCG using $\tau = 1$, $\tau = 10$, and $\tau = 20$, respectively.

topology, a single step of GalProj will introduce a communication overhead equal to

$$t_\tau = \lceil \log(K) \rceil (2\ell + 2\tau p^2 q + \tau p^2 \gamma), \tag{10}$$

where τ denotes the number of direction blocks that are simultaneously deflated, τp^2 is the message size, and γ is the cost per arithmetic operation [13]. The total communication overhead introduced by the Galerkin projections in GalProj for all $\delta - 1$ subsequent batches of p right-hand sides thus is

$$t_{\text{ppbcg}} = \frac{\hat{\zeta}(\delta - 1)t_\tau}{\tau}. \tag{11}$$

We quantify the above discussion on a synthetic experiment setting $\ell = 10^4 q$, i.e., the latency (startup) cost is 10^4 times larger than the cost to transmit a floating-point scalar, which is typical for current architectures. To account for different network architectures, we did not set an actual value for q. Figure 3 plots the modeled communication cost t_{ppbcg} as the numerical value of each one of the variables K, p, δ and $\hat{\zeta}$ is varied, leaving the rest in their default value. The default values used were $K = 128$, $p = 20$, $\delta = 20$ and $\hat{\zeta} = 250$. Deflating one direction block at a time (τ=1) leads to larger communication costs compared to $\tau > 1$, with larger values of τ leading to better performance. More specifically, as τ increases, the communication pattern of PP-BCG shifts from latency-dominated to bandwidth-dominated. On the other hand, larger values of p already to bandwidth-dominated communication pattern, thus undercutting the effect of τ (see Figure 3 (c)).

4. Experiments and performance evaluation

In this section we present numerical experiments performed in distributed memory computing environments. The proposed method, PP-BCG, was implemented in Fortran 90, and all local MATMUL were performed using the BLAS-3 `DGEMM` routine in IBM's ESSL. For the rest of this section we set the parameters in PP-BCG as $\text{tol}_1 = 10^{-12}$, $\text{tol}_2 = 10^{-6}$, and $\hat{\zeta} = 200$. While setting $\tau > 1$ can lead to an improved performance in PP-BCG, its choice enables a non-trivial numerical study which requires a lengthier exposition. All of our experiments were performed using $\tau = 1$. Moreover, we only considered 2-D processor grids of size $M \times K$ where M and K were related either as $K = M$ or $K = 2M$. All computations were performed in 64-bit arithmetic and all times are shown in seconds.

4.1. Computational system

The experiments were performed on a massively parallel architecture consisting of up to 4 racks of an IBM BlueGene/Q[1] (BG/Q) supercomputer [24]. Each BG/Q rack consists of 1024 compute nodes, each node hosting an 18 core A2 chip that runs at 1.6 GHz, and 16 GBytes of system memory. Sixteen of the 18 cores are for computation, one for the lightweight O/S kernel, and one for redundancy. Every core supports 4 hardware threads, thus, in total a rack has 16,384 cores and can support up to 65,536 threads. BG/Q nodes are connected by a 5-dimensional bidirectional network, with a network bandwidth of 2 GBytes/s for sending and receiving data. Each BG/Q rack features dedicated I/O nodes with 4 GBytes/s I/O bandwidth. The system implements optimized collective communication and allows specialized tuning of point-to-point communication. The PP-BCG source files were compiled using the IBM XL F compiler[2] version 14.1.13.

4.2. Test matrices

As in [4, 9, 33], we used a synthetic dataset consisting of model covariance matrices, generated to simulate real-case scenarios in UQ. In particular, each different matrix A was formed as

$$A_{ii} = 1 + i^{\theta}, \quad A_{ij} = 1/|i-j|^{\kappa} \ (if\ i \neq j), \quad i,j = 1,\ldots,n, \tag{12}$$

for real θ and fixed $\kappa = 2$. The progressive decay away from the main diagonal simulates the fact that features are locally-only correlated, and the condition number of these matrices is known to scale like n^{θ}. In addition, we also experimented with a more general dataset, generated by imposing additional symmetric perturbations (maintaining the SPDness) at random positions of (12), that is $A_{\hat{i}\hat{j}} = A_{ij} + \delta_{\hat{i}\hat{j}}$, $\delta_{\hat{i}\hat{j}} = 100/|\hat{i} - \hat{j} + 1|$, and $\hat{i},\ \hat{j} \in \mathcal{I}$ where $\mathcal{I}$ represents a random subset of the integers in $[1, n]$ and $|\mathcal{I}| = n/100$. Even though, if explicitly available, the matrices in (12) would be amenable to special fast methods, for the reasons explained earlier and as in prior literature, we do not make use of such techniques. Finally, each batch $Z^{(j)}$ was formed by p n-dimensional vectors, each vector having entries ± 1 with equal probability (Rademacher variables).

[1]IBM and Blue Gene/Q are trademarks of International Business Machines Corporation, registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies.

[2]Flags used: `-O5 -qnosave -qdebug=recipf:forcesqrt -qmaxmem=-1 -qipa=level=2 -qhot=level=2 -qarch=qp -qtune=qp -qsmp=omp:noauto -qthreaded -qsimd=auto`

4.3. Numerical demonstration

The application of interest is the approximation of the diagonal of a matrix that is only available via MATVEC's (MATMUL's) using the so called Hutchinson estimator. The diagonal of A^{-1}, $\mathcal{D}(A^{-1})$, can be approximated by the following stochastic estimator [9]:

$$\mathcal{D}_\delta(A^{-1}) := \left[\sum_{j=1}^{\delta}\sum_{k=1}^{p} Z^{(j,k)} \odot X^{(j,k)}\right] \oslash \left[\sum_{j=1}^{\delta}\sum_{k=1}^{p} Z^{(j,k)} \odot Z^{(j,k)}\right], \tag{13}$$

where $Z^{(j,k)}$ denotes the kth column of $Z^{(j)}$, $X^{(j,k)} = A^{-1}Z^{(j,k)}$, and the symbols $\odot, \oslash$ denote element-wise multiplication and division, respectively.

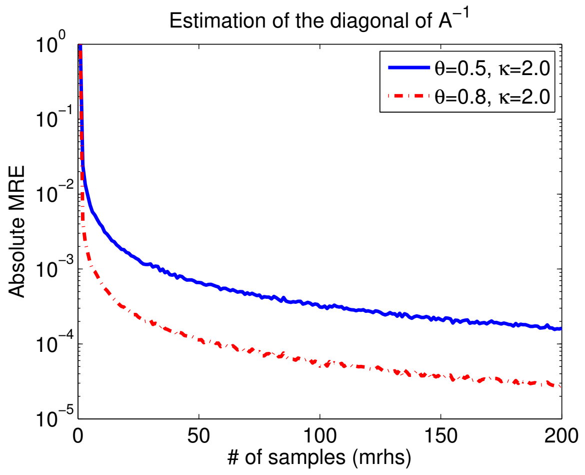

Figure 4: MRE for Monte Carlo stochastic estimator in (13) for a model covariance matrix of size n=8,192 and $\theta = 0.5$, $\theta = 0.8$.

Figure 4 plots the mean relative error (MRE) of the estimator in (13) for a small-scale model covariance matrix of size $n = 8,192$ as the number of samples (right-hand sides) increases. As was extensively discussed in [33], the fast solution of (1) is critical for the success of stochastic diagonal estimation since for estimators with large variance the number of right-hand sides that must be solved might be of the order $\mathcal{O}(10^3)$.

We now consider the application of PP-BCG on two batches $AX^{(1)} = Z^{(1)}$ and $AX^{(2)} = Z^{(2)}$, each with $p = 5$ right-hand sides, for the same model covariance matrices as in Figure 4. Figure 5 plots the inner product between the initial residual of the first right-hand side of the yet unsolved batch $AX^{(2)} = Z^{(2)}$ after we apply GALPROJ, and the eigenvectors associated with the thirty algebraically smallest eigenvalues of A. The curve associated with the legend "Standard" refers to applying the Galerkin projections in the standard (non-reverse) order as in [14, 48]. Similarly to the results shown in Section 2.3, exploiting the GALPROJ algorithm, as PP-BCG does, leads to an initial approximation $\hat{X}^{(2)}_{\hat{\varsigma}}$ for which the associated residual $\hat{R}^{(2)}_{\hat{\varsigma}}$ is nearly orthogonal to the eigenvectors associated with the few smallest eigenvalues of A.

4.4. Runtimes and efficiency of PP-BCG

We now consider the performance of PP-BCG on distributed memory computing environments. We used three model covariance matrices generated as in (12), each of size n=131,072, n=262,144

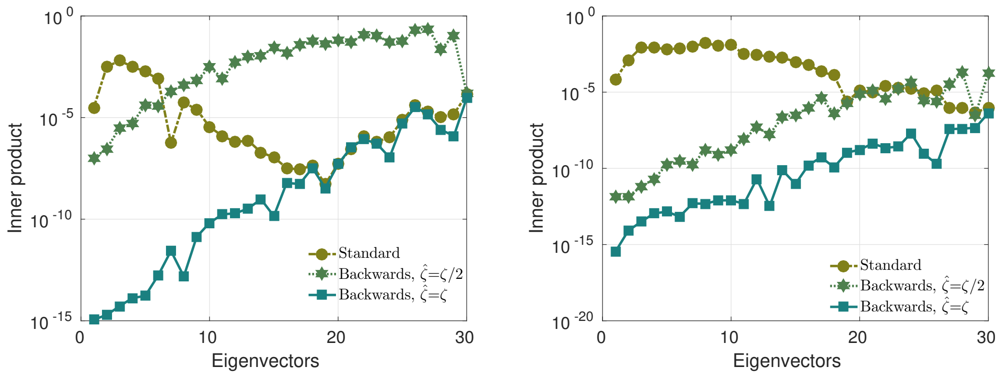

Figure 5: Inner product of $\hat{R}_{\zeta}^{(2)}$ with the thirty lowest eigenvectors of the model covariance matrices A in Figure 4. Galerkin projections are performed in the natural order ("●") and reverse order ("■","∗"). Left: $\theta = 0.5$. Right: $\theta = 0.8$.

and n=524,288. For each value of n, we considered two different values of θ, $\theta = 0.6$ and $\theta = 0.8$, and solved for a total of $s = 800$ right-hand sides, dividing the latter in batches of size $p = 20$, 40 and $p = 80$. For n=131,072 we used 2^{ν}, $\nu = 4, \ldots, 10$ BG/Q compute nodes (i.e. up to one BG/Q rack). For n=262,144 the values of ν were $\nu = 6, \ldots, 12$, while for n=524,288 the latter was set to $\nu = 8, \ldots, 12$ (i.e. up to four BG/Q racks). Since each BG/Q node features exactly 16 processor cores devoted to computation, the total number of cores used were 16,384 (for n=131,072) and 65,536 for the two larger matrices. The number of MPI processes was always equal to the number of cores and each MPI process utilized 2 hardware threads to provide maximum bandwidth.

Figure 6 illustrates the strong scalability of PP-BCG for all different combinations of n, p and θ (log-log scale). The runtimes include all different phases of PP-BCG, i.e., the time required to obtain the initial approximations and solve for all batches. Observe that larger values of p lead to reduced runtimes for all matrix sizes and condition numbers. We will verify in Section 4.6 that this behavior is due to the faster convergence of PP-BCG when higher values of p are used.

Figures 7-9 plot the efficiencies of the PP-BCG solver and the distributed MATMUL for all different values of n, p and θ. Assuming a reference baseline execution time t_r on a baseline number of G_r MPI processes, the parallel efficiency is computed as $E_c = \dfrac{G_r t_r}{G_c t_c}$, where t_c denotes the execution time on $G_c > G_r$ MPI processes. The efficiencies observed for PP-BCG follow two different regimes. For smaller numbers of compute nodes, the runtime of PP-BCG is mostly spent on MATMULs and thus runs at very high efficiencies, commensurate with those achieved by MATMUL, which run at almost perfect efficiency. As the number of compute nodes grows, non-MATMUL operations, i.e., block AXPY and block inner products, account for a larger portion of the total computational profile since they scale only along the second dimension of the 2-D processor grid (this is also illustrated in Figure 10 where we plot the computational profile of the smallest matrix n=131,072 for all different values of p). As a consequence, PP-BCG transitions to a regime where its efficiency is mostly determined by the second dimension of the 2-D processor grid. Overall, as the compute nodes double for the regimes we have explored, the average observed efficiencies range from 85 to 90%.

One way to increase the efficiency of PP-BCG when non-MATMUL operations dominate the computational profile is to use an $M \times K$ processor grid where $K \gg M$. However, in this case

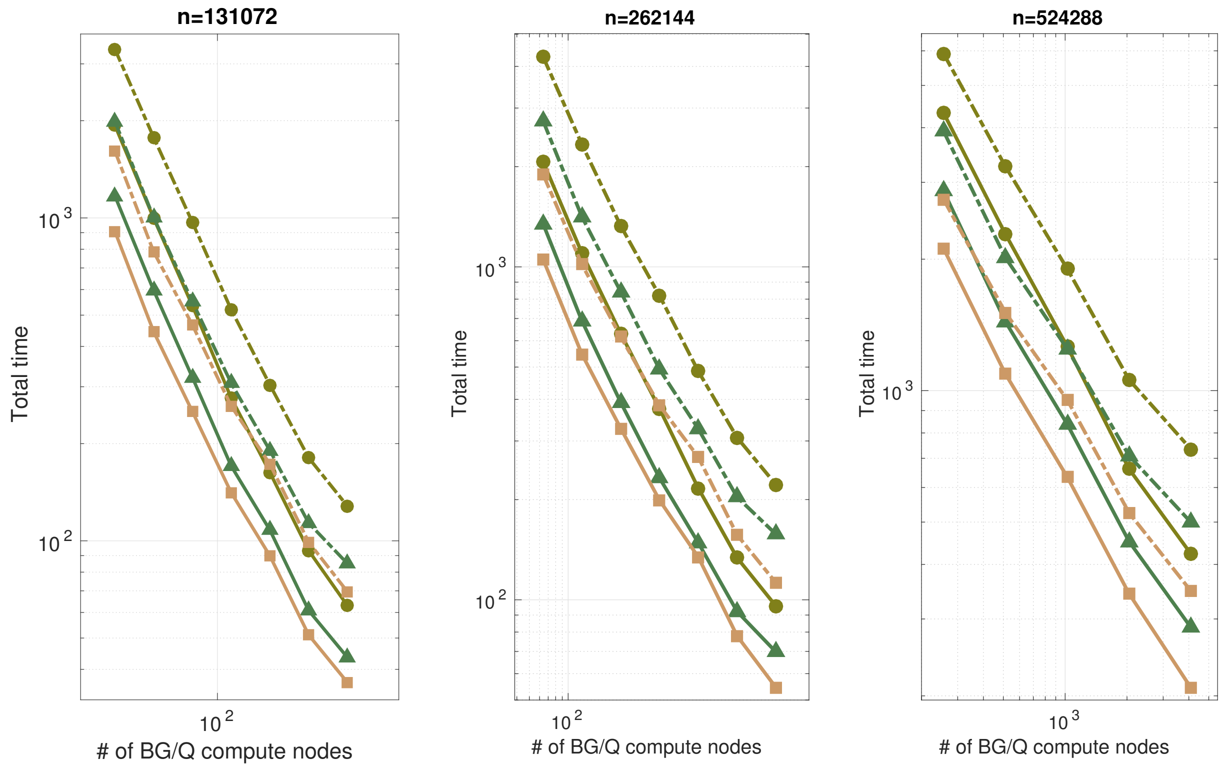

Figure 6: Runtimes of the PP-BCG solver. ●: $p = 20$, ▲ : $p = 40$, ■ : $p = 80$. Solid lines: $\theta = 0.6$. Dashed lines: $\theta = 0.8$.

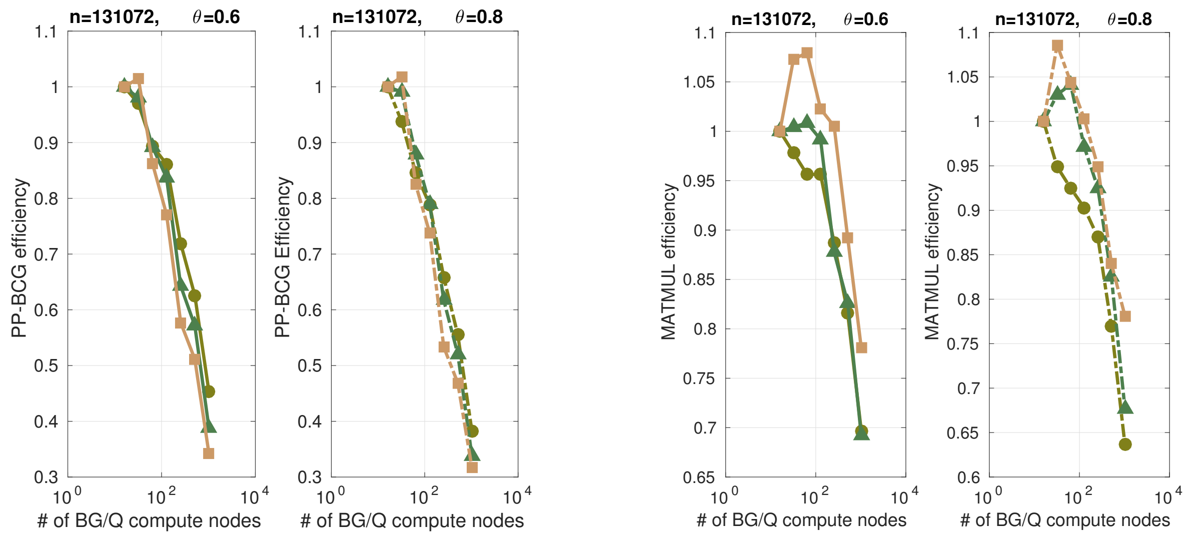

Figure 7: Efficiency of the PP-BCG solver and the MATMUL for n=131,072: ●: $p = 20$, ▲ : $p = 40$, ■ : $p = 80$. Solid lines: $\theta = 0.6$. Dashed lines: $\theta = 0.8$.

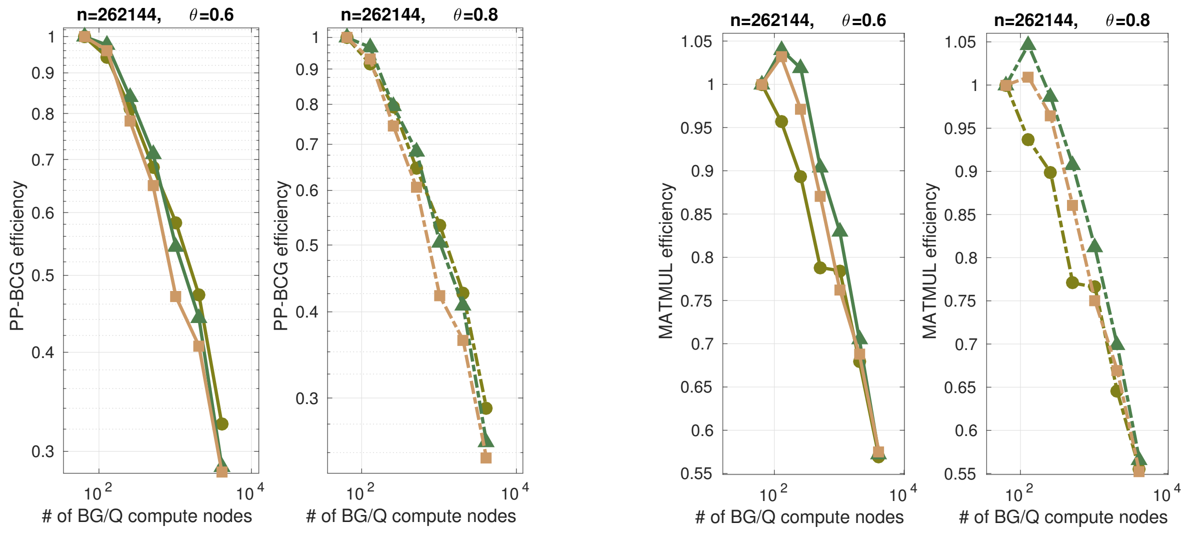

Figure 8: Efficiency of the PP-BCG solver and the MATMUL for n=262,144: ●: $p = 20$, ▲ : $p = 40$, ■ : $p = 80$. Solid lines: $\theta = 0.6$. Dashed lines: $\theta = 0.8$.

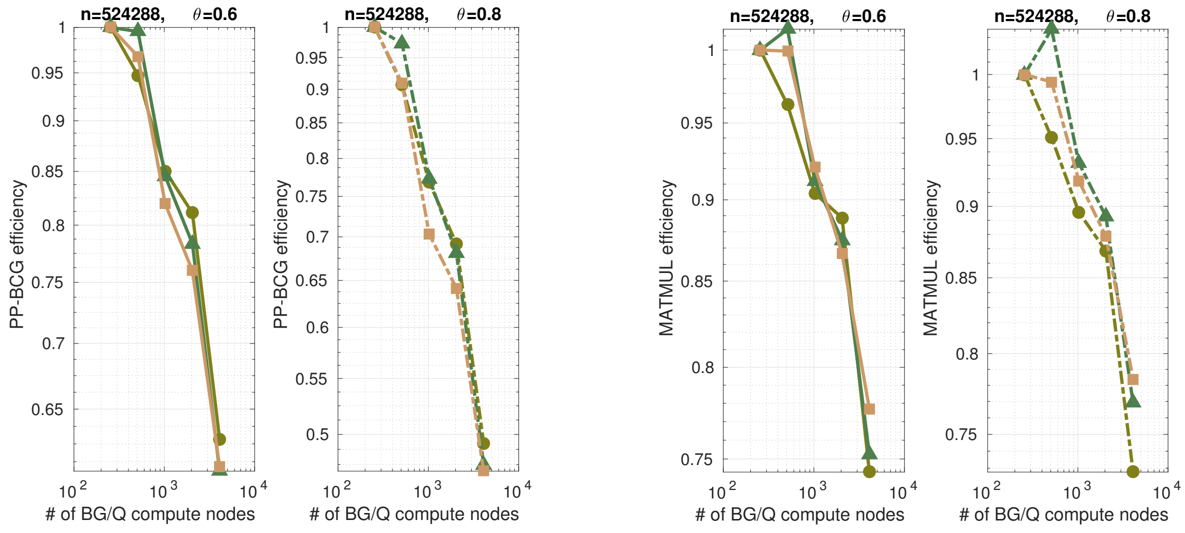

Figure 9: Efficiency of the PP-BCG solver and the MATMUL for n=524,288: ●: $p = 20$, ▲ : $p = 40$, ■ : $p = 80$. Solid lines: $\theta = 0.6$. Dashed lines: $\theta = 0.8$.

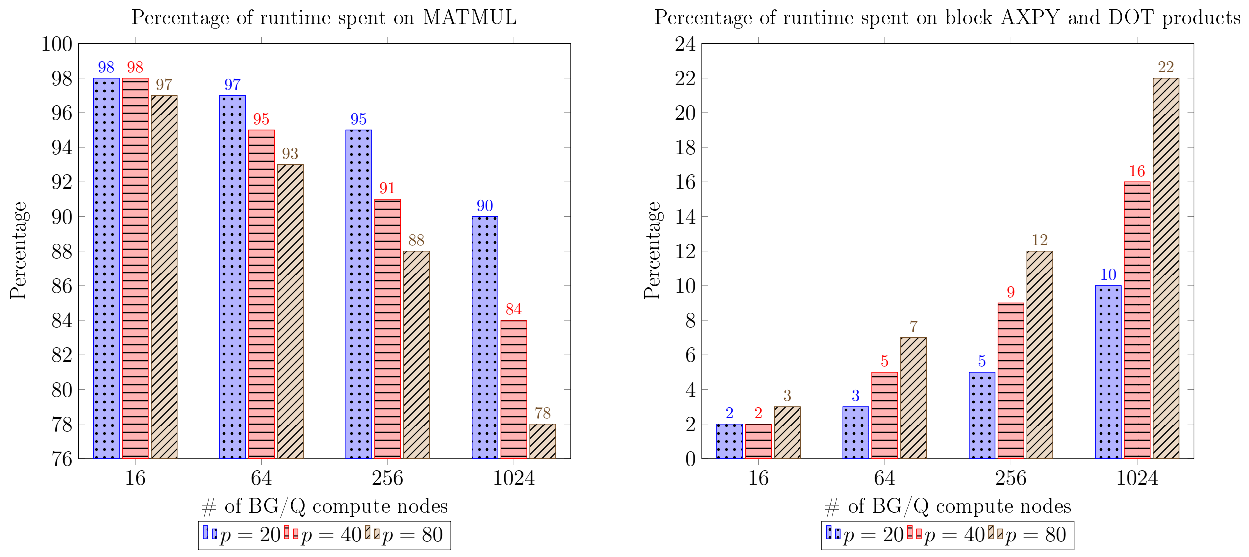

Figure 10: Performance profile of PP-BCG in terms of compute primitives for the case where n=131,072 and $\theta = 0.6$. The x-axis denotes the number of BG/Q nodes. For each ensemble of BG/Q nodes, the leftmost, middle and rightmost bars correspond to $p = 20$, $p = 40$ and $p = 80$, respectively.

the efficiency of the MATMUL operation would not remain the same since the shape of the 2-D processor grid would become more similar to that of a 1-D topology. From extensive experiments we performed (not reported due to space limitations), we determined that the optimal topologies were obtained when $K = \rho M$ with a small $\rho > 1$.

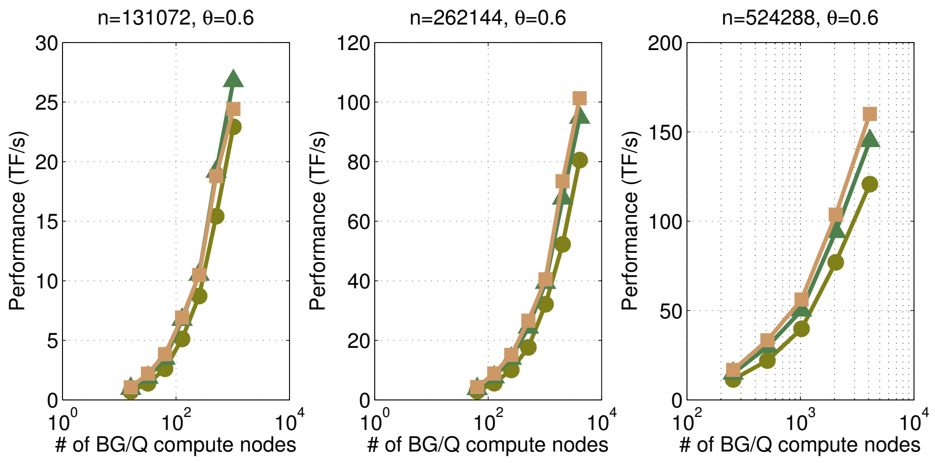

Figure 11: Performance (TF/s). ●: $p = 20$, ▲ : $p = 40$, ■ : $p = 80$.

Finally, Figure 11 illustrates the performance of the PP-BCG scheme in terms of Tera Flops per second (TF/s). Results shown are for $\theta = 0.6$.

4.5. Memory-runtime trade-offs in PP-BCG

Although we did not experience any breakdowns due to memory limitations, it is important to consider the performance trend of PP-BCG in scenarios where the direction blocks $P_0, P_1, \ldots, P_{\hat{\zeta}-1}$

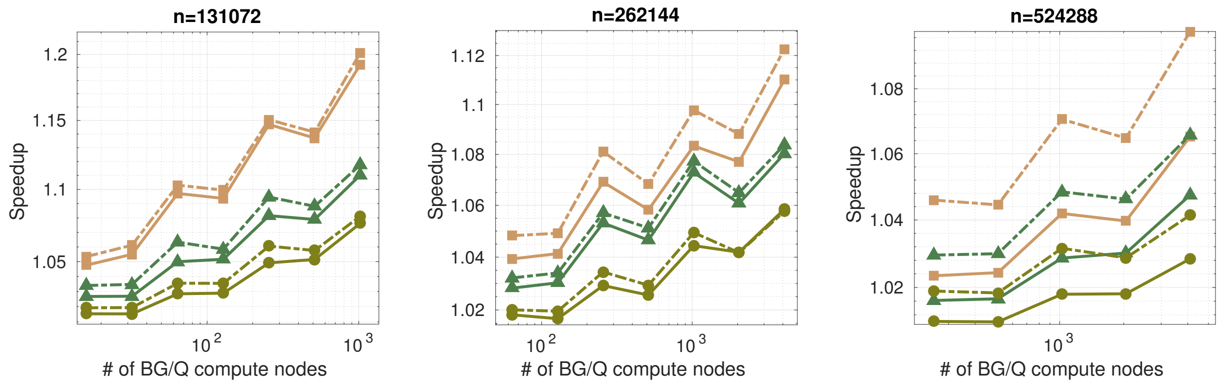

Figure 12: Speedup of the "unlimited" memory version of PP-BCG versus the limited memory version. ●: $p = 20$, ▲ : $p = 40$, ■ : $p = 80$. Solid lines: $\theta = 0.6$. Dashed lines: $\theta = 0.8$.

can not be stored explicitly, and must thus be re-computed by GALPROJ on-the-fly (see the discussion in Section 2.4).

Figure 12 shows the speedup of PP-BCG using the "unlimited" memory version (the one for which we reported results so far) versus the limited memory version for the test matrices considered in Section 4.4. The numerical behavior of the two different versions of PP-BCG was very similar (the total number of iterations performed by the memory limited version of PP-BCG was slightly bigger). When the distributed MATMUL dominates the runtimes, the computational profile of the two versions of PP-BCG is quite similar. However, when the total amount of time spent on MATMULs reduces, the block-AXPY operations in (6) and (7) become noticeable, and the memory limited version of PP-BCG becomes slower than the "unlimited" memory version.

4.6. Comparisons with CG, BCG *and deflated* BCG

In this section we compare the performance of PP-BCG relative to other schemes such as the "pseudo"-block CG, the BCG, as well as the deflated version of BCG, abbreviated as (D-BCG) [16]. Similarly to PP-BCG, all schemes were implemented in Fortran 90 and compiled using the IBM XL F compiler.

To perform the experiments with D-BCG we followed a two-stage procedure. First, we called block Lanczos[3] [25], with a block-size equal to p and performed a number of iterations which was equal to the number of iterations performed by the "seed" system $AX^{(1)} = Z^{(1)}$ in PP-BCG. We then retained all approximate eigenpairs of A for which the corresponding residual norm was less or equal to 10^{-4}. The second phase of D-BCG consisted of obtaining an initial approximation $X_0^{(j)}$ for each batch $AX^{(j)} = Z^{(j)}$, $j = 1, \ldots, \delta$, by exploiting the previously computed approximate eigenvectors of A, and then passing this approximation to BCG. The initial approximation $X_0^{(j)}$ was obtained as

$$X_0^{(j)} = U(U^T AU)^{-1} U^T Z^{(j)}, \tag{14}$$

where $U \in \mathbb{R}^{n \times w}$ denotes the matrix associated with the $1 \leq w \leq n$ eigenvectors of A retained in the first stage. In particular, if U holds the eigenvectors of A associated with its $1 \leq w \leq n$ smallest eigenvalues, D-BCG converges with an effective condition number $\kappa_{cn} = \lambda_n / \lambda_{p+w}$ [16].

[3]We used full orthogonalization

Table 3: Average number of iterations per batch of p right-hand sides during the application of CG, BCG, D-BCG and PP-BCG to the matrices in Section 4.4.

	n=131,072			**n=262,144**			**n=524,288**		
Batch size (p)	20	40	80	20	40	80	20	40	80
$\theta = 0.6$									
CG		156			192			242	
BCG	85	71	62	137	111	87	176	155	141
D-BCG	63	52	46	104	89	78	133	117	104
PP-BCG	50	40	35	82	61	55	113	98	83
$\theta = 0.8$									
CG		471			625			781	
BCG	189	154	122	258	208	149	374	322	246
D-BCG	154	112	81	182	155	136	260	224	193
PP-BCG	87	67	60	111	86	64	141	119	92

4.6.1. Convergence rate per right-hand side

Table 3 shows the average number of iterations per batch of p right-hand sides (equivalently, the average number of iterations per right-hand side) obtained when solving all $\delta = 800/p$ batches of right-hand-sides by CG, BCG, D-BCG and PP-BCG for the model covariance matrices in (4.4). For PP-BCG, the results listed include the amortization of the extra iterations performed when solving $AX^{(1)} = Z^{(1)}$ to the higher accuracy $\text{tol}_1 = 10^{-12}$.

Summarizing the results, larger values of p lead to fewer iterations per batch for BCG, D-BCG and PP-BCG since the effective condition number of each right-hand side was at most λ_n/λ_p. Moreover, D-BCG and PP-BCG always converged faster than BCG because of their non-trivial initial approximation. In particular, PP-BCG and D-BCG offered greater speedups for problems that were less well-conditioned (as in $\theta = 0.8$). As the value of p increases, the speedup ratio of D-BCG and PP-BCG over BCG starts declining, since the convergence rate is now affected more by the blocksize p and less by the efficiency of the non-trivial initial approximation. On the other hand, the convergence rate of CG is oblivious to the value of p since the solution procedure for each right-hand side is independent from the others within each batch.

A similar behavior was also observed for the perturbed model covariance matrices in Table 4.

4.6.2. Runtime comparisons

The results discussed in Section 4.6.1 demonstrated the efficiency of PP-BCG in terms of faster convergence rate per batch. However, when considering massively parallel architectures, the above does not necessarily imply that PP-BCG will be the fastest scheme also in terms of runtime.

Figure 13 plots the average runtime required to solve for a batch of p right-hand sides by CG, BCG and PP-BCG, for the values of θ and p shown in Table 3 and the model covariance matrices of size n=131,072 and n=524,288. Results for D-BCG are omitted since its runtime was always longer than that of PP-BCG. PP-BCG was the fastest scheme overall, while CG and BCG became

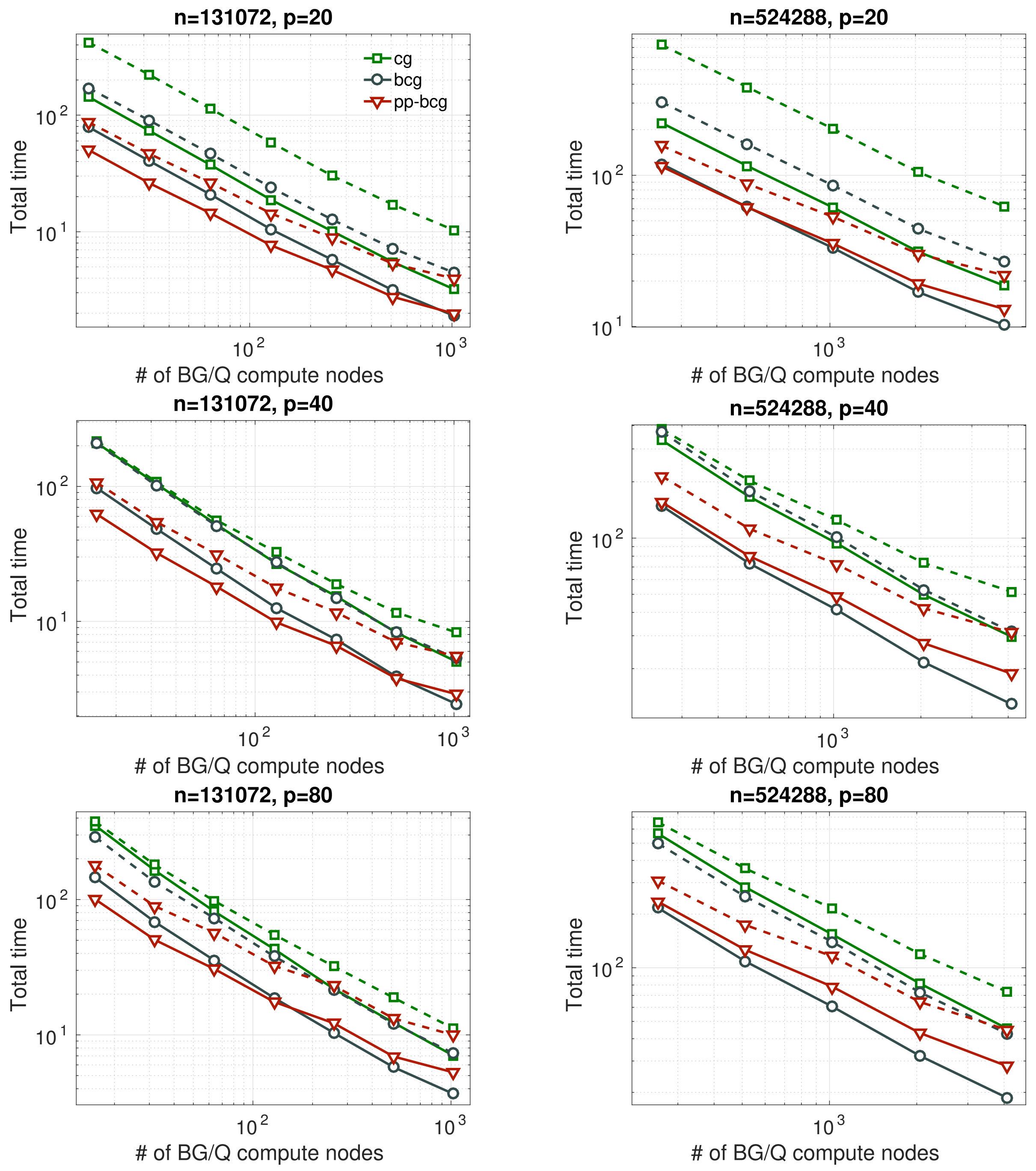

Figure 13: Runtime per batch solved by CG ("□"), BCG ("◯") and PP-BCG ("▽") for the values of θ and p shown in Table 3 for $n = 131,072$ (left column) and $n = 524,288$ (right column). Solid lines: $\theta = 0.6$. Dashed lines: $\theta = 0.8$

Table 4: Average number of iterations per batch of p right-hand sides during CG, BCG, D-BCG and PP-BCG for symmetrically perturbed model covariance matrices.

	n=131,072			**n=262,144**			**n=524,288**		
Batch size (p)	20	40	80	20	40	80	20	40	80
$\theta = 0.6$									
CG		170			182			218	
BCG	102	88	73	132	102	83	148	136	128
D-BCG	67	59	50	86	70	59	118	108	102
PP-BCG	58	44	36	68	54	42	82	78	70
$\theta = 0.8$									
CG		495			588			703	
BCG	168	138	116	224	176	122	295	239	196
D-BCG	128	91	70	177	143	103	217	180	148
PP-BCG	76	59	48	104	78	58	130	117	102

more competitive for higher values of p. In particular, PP-BCG consistently outperformed both CG and BCG when the biggest portion of the runtimes was spent on MATMUL (as in $\theta = 0.8$). On the other hand, because CG and BCG perform less non-MATMUL operations than PP-BCG, and non-MATMUL operations scale only along the second dimension of the 2-D processor grid, their efficiency tends to be higher.

4.7. Comparing with ScaLAPACK

In this last experiment we compare PP-BCG against the distributed memory Cholesky-based solver in ScaLAPACK [11]. For ScaLAPACK, the covariance matrix A was distributed in a two-dimensional block-cyclic manner among the processors of the 2-D processor grid. The Cholesky factorization of A was obtained by `PDPOTRF` and the linear system solution for each batch of p right-hand sides was performed by `PDPOTRS`. The computational blocksize in `PDPOTRF` was held the same for both dimensions of the 2-D processor grid and, after some experimental tuning, was set to $m_b = 128$.

Figure 14 presents comparisons for the model covariance matrix of size n= 262,144 using a fixed number of 16,384 MPI processes. The left subfigure shows the runtime of `PDPOTRF` in ScaLAPACK, as well as the runtimes of PP-BCG for all different combinations of $p = 40$, $p = 80$, and $\theta = 0.6$, $\theta = 0.8$, as δ varies. Depending on the condition number of A, as well as the batch size p, there is a value of δ after which it is more efficient to use the direct solver instead of PP-BCG. This can be also verified by the right subfigure of Figure 14 where we plot the speedup ratio of PP-BCG over ScaLAPACK. As a general remark, we found PP-BCG a better alternative than ScaLAPACK in all cases where the total number of MATVEC products were less than n. The main advantage of the direct solver is that once the Cholesky factorization of A becomes available, each batch of p right-hand sides can be solved at the cost of a single MATMUL product. In practice, however, an iterative approach like the one in PP-BCG offers several advantages over direct methods. The solver

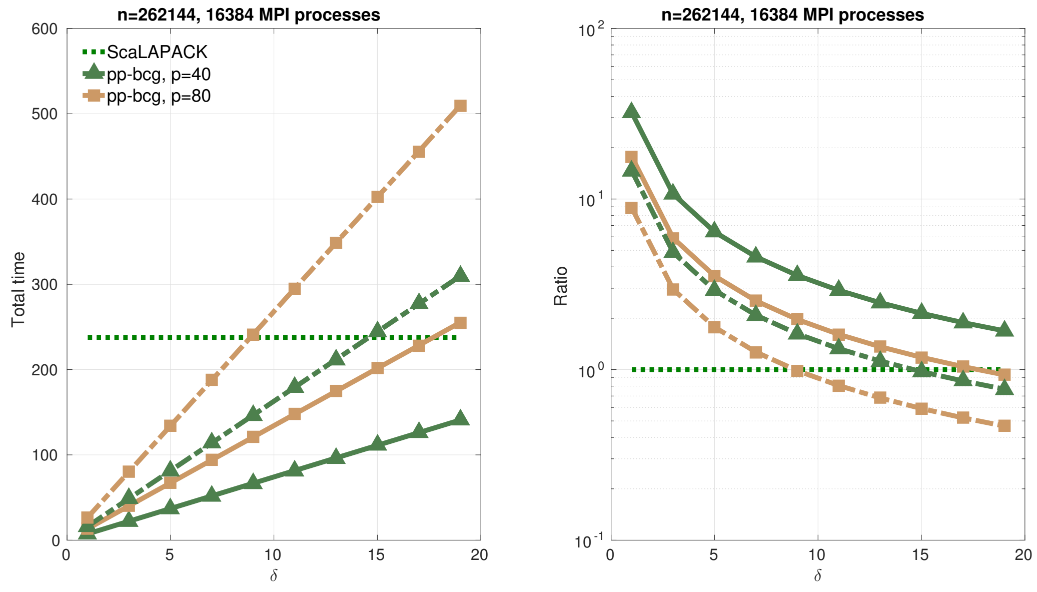

Figure 14: Comparison of ScaLAPACK (dotted line) and PP-BCG (▲ : $p = 40$, ■ : $p = 80$) as δ varies (solid lines: $\theta = 0.6$, dashed lines: $\theta = 0.8$). Left: runtimes. Right: speedup of PP-BCG over ScaLAPACK.

can be tuned to the accuracy demanded by the user, which in the application discussed here and other areas in Data Analytics is lower than that obtained by standard direct solvers operating in one precision throughout. Moreover, PP-BCG is matrix-free and therefore applicable when A is implicitly available, which is not the case with direct methods. Finally, in the course of the MATMUL's, one can also exploit any structure in matrix A and right-hand sides.

5. Conclusion

In this paper we proposed PP-BCG, a distributed numerical scheme for the solution of dense, symmetric and positive-definite linear systems. PP-BCG combines recycling of Krylov subspaces by Galerkin projections with BCG in order to accelerate its convergence. The key idea in PP-BCG is to take advantage of the availability of a good approximation of the invariant subspace associated with the extremal eigenvalues of A generated during the linear system solution procedure for the first batch of right-hand sides. To ease the effects of finite precision arithmetic, PP-BCG applies the Galerkin projections in a reverse manner. We discussed the numerical aspects of PP-BCG and described its parallel implementation and a performance model for distributed memory 2-D processor grids. Experiments performed on a few BG/Q racks illustrated the performance of PP-BCG compared to several other approaches for solving large linear systems with model covariance matrices and multiple right-hand sides in an Uncertainty Quantification application. More specifically, PP-BCG can be particularly helpful when the biggest portion of the computational runtimes is devoted to the MATMULs.

As part of our future work, we plan to pursue a study of PP-BCG on co-processor systems in combination with mixed precision arithmetic and iterative refinement. Another interesting path of research would be to consider the extension of PP-BCG to applications in other areas, e.g., Quantum

Chromodynamics, and the study of the implications of special matrix structure such as sparsity on the overall performance. Indeed, while in this paper we focused on dense matrices, the proposed technique can be applied to sparse matrices and/or dense structured matrices on an "as is" basis without any need for modifications except for an implementation of an efficient MATVEC (MATMUL) routine adjusted to the problem of interest.

6. Acknowledgments

Vassilis Kalantzis was partially supported by a Gerondelis Foundation Fellowship. This work was completed while E. Gallopoulos was on sabbatical leave at the Dept. CSE, University of Minnesota. We thank the University of Minnesota Supercomputing Institute for providing the computational resources needed in the initial stage of this work. We also thank the anonymous referees for their comments which greatly improved the clarity and presentation of this paper.

References

[1] A. M. Abdel-Rehim, R. B. Morgan, and W. Wilcox, *Improved seed methods for symmetric positive definite linear equations with multiple right-hand sides*, Numer. Linear Algebra Appl., 21 (2014), pp. 453–471.

[2] E. Agullo, L. Giraud, and Y.-F. Jing, *Block GMRES method with inexact breakdowns and deflated restarting*, SIAM J. Matrix Anal. Appl., 35 (2014), pp. 1625–1651.

[3] E. Anderson et al., *LAPACK Users' Guide*, SIAM, 3d ed., 1999.

[4] C. Angerer et al., *A Fast, Hybrid, Power-Efficient High-Precision Solver for Large Linear Systems Based on Low-Precision Hardware*, Sustainable Computing: Informatics and Systems, (2015), pp. 1–27.

[5] M. Anitescu, J. Chen, and M. L. Stein, *An inversion-free estimating equations approach for Gaussian process models*, J. Comput. Graph. Stat., 26 (2017), pp. 98–107.

[6] H. Avron and S. Toledo, *Randomized algorithms for estimating the trace of an implicit symmetric positive semi-definite matrix*, J. ACM, 58 (2011), pp. 1–34.

[7] E. Bavier, M. Hoemmen, S. Rajamanickam, and H. Thornquist, *Amesos2 and Belos - Direct and iterative solvers for large sparse linear systems.*, Sci. Program., (2012).

[8] C. Bekas, A. Curioni, and I. Fedulova, *Low-cost data uncertainty quantification*, Concur. Comput.: Pract. Exper., 24 (2012), pp. 908–920.

[9] C. Bekas, E. Kokiopoulou, and Y. Saad, *An estimator for the diagonal of a matrix*, Appl. Numer. Math., 57 (2007), pp. 1214 – 1229.

[10] S. Birk and A. Frommer, *A deflated Conjugate Gradient method for multiple right hand sides and multiple shifts*, Numer. Algorithms, 67 (2014), pp. 507–529.

[11] L. Blackford et al., *ScaLAPACK Users' Guide*, SIAM, 1997.

[12] H. Calandra, S. Gratton, R. Lago, X. Vasseur, and L. M. Carvalho, *A modified block flexible GMRES method with deflation at each iteration for the solution of non-hermitian linear systems with multiple right-hand sides*, SIAM J. Sci. Comput., 35 (2013), pp. S345–S367.

[13] E. Chan, M. Heimlich, A. Purkayastha, and R. van der Geijn, *Collective communication: Theory, practice, and experience: Research articles*, Concurr. Comput. : Pract. Exper., 19 (2007), pp. 1749–1783.

[14] T. Chan and W. Wan, *Analysis of projection methods for solving linear systems with multiple right-hand sides*, SIAM J. Sci. Comput., 18 (1997), pp. 1698–1721.

[15] T. F. Chan and M. K. Ng, *Galerkin projection methods for solving multiple linear systems*, SIAM J. Sci. Comput., 21 (1999), pp. 836–850.

[16] J. Chen, *A deflated version of the block Conjugate Gradient algorithm with an application to Gaussian process maximum likelihood estimation*, Tech. Rep. ANL/MCS-P1927-0811, Argonne Nat'l. Lab., 2011.

[17] J. Chen, *How accurately should I compute implicit matrix-vector products when applying the Hutchinson trace estimator?*, SIAM J. Sci. Comput., 38 (2016), pp. A3515–A3539.

[18] J. Chen, T. Li, and M. Anitescu, *A parallel linear solver for multilevel Toeplitz systems with possibly several right-hand sides*, Parallel Comput., 40 (2014), pp. 408 – 424.

[19] J. Dutiné, M. Clemens, and S. Schöps, *Multiple Right-Hand Side Techniques in Semi-Explicit Time Integration Methods for Transient Eddy Current Problems.*, CoRR, (2016).

[20] A. El Guennouni, K. Jbilou, and H. Sadok, *The block Lanczos method for linear systems with multiple right-hand sides*, Appl. Numer. Math., 51 (2004), pp. 243–256.

[21] C. Farhat, L. Crivelli, and F. X. Roux, *Extending substructure based iterative solvers to multiple load and repeated analyses*, Comput. Methods in Appl. Mech. Eng., 117 (1994), pp. 195–209.

[22] E. Gallopoulos, B. Philippe, and A. Sameh, *Parallelism in Matrix Computations*, Springer, 2016.

[23] E. Gallopoulos and V. Simoncini, *Iterative solution of multiple linear systems: Theory, practice, parallelism, and applications*, in Proc. 2nd Int'l. Conf. Comput.Structures Tech., B. Topping and M. Papadrakakis, eds., Civil-Comp Press, Edinburgh, 1994, pp. 47–51.

[24] M. Gilge, *IBM System Blue Gene Solution: Blue Gene/Q Application Development*, IBM Int'l. Tech. Supp. Org., 2nd ed., June 2013.

[25] G. H. Golub and R. Underwood, *The block Lanczos method for computing elgenvalues*, In Mathematical Software III, J.R. Rice (Ed.), Academic Press, New York, (1977), pp. 361–377.

[26] P. Gosselet, D. Rixen, F.-X. Roux, and N. Spillane, *Simultaneous FETI and block FETI: Robust domain decomposition with multiple search directions*, Int'l. J. Numer. Meth. Engrng., 104 (2015), pp. 905–927.

[27] M. Gutknecht, *Block Krylov space methods for linear systems with multiple right-hand sides: An introduction*, In Modern Mathematical Models, Methods and Algorithms for Real World Systems, Siddiqi AH, Duff IS, Christensen O (eds), Anamaya, New Delhi, 2007., pp. 420–447.

[28] M. Hestenes and E. Stiefel, *Methods of Conjugate Gradients for solving linear systems*, J. Res. NBS, 49 (1952), pp. 409–436.

[29] M. Hutchinson, *A stochastic estimator of the trace of the influence matrix for Laplacian smoothing splines*, Commun. Stat. Simul. Comput., 19 (1990), pp. 433–450.

[30] H. Ji and Y. Li, *A breakdown-free block Conjugate Gradient method*, BIT Numerical Mathematics, (2016), pp. 1–25.

[31] H. Ji, M. Sosonkina, and Y. Li, *An implementation of block conjugate gradient algorithm on cpu-gpu processors*, in 2014 Hardware-Software Co-Design for High Performance Computing, Nov 2014, pp. 72–77.

[32] P. Jolivet and P.-H. Tournier, *Block iterative methods and recycling for improved scalability of linear solvers*, in Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '16, Piscataway, NJ, USA, 2016, IEEE Press, pp. 17:1–17:14.

[33] V. Kalantzis, C. Bekas, A. Curioni, and E. Gallopoulos, *Accelerating data uncertainty quantification by solving linear systems with multiple right-hand sides*, Numer. Algorithms, 62 (2013), pp. 637–653.

[34] M. E. Kilmer and E. de Sturler, *Recycling Subspace Information for Diffuse Optical Tomography*, SIAM J. Sci. Comput., 27 (2006), pp. 2140–2166.

[35] G. Li, *A block variant of the GMRES method on massively parallel processors.*, Parallel Comput., 23 (1997), pp. 1005–1019.

[36] X. Liu, E. Chow, K. Vaidyanathan, and M. Smelyanskiy, *Improving the performance of dynamical simulations via multiple right-hand sides*, in 2012 IEEE 26th International Parallel and Distributed Processing Symposium, May 2012, pp. 36–47.

[37] P. Lötstedt and M. Nilsson, *A Minimal Residual Interpolation Method for Linear Equations with Multiple Right-Hand Sides*, SIAM J. Sci. Comput., 25 (2004), pp. 2126–2144.

[38] A. Murli et al., *A multi-grained distributed implementation of the parallel block Conjugate Gradient algorithm*, Concur. Comput.: Pract. Exper., 22 (2010), pp. 2053–2072.

[39] D. O'Leary, *The block Conjugate Gradient algorithm and related methods*, Lin. Alg. Appl., 29 (1980), pp. 293 – 322.

[40] ——, *Parallel implementation of the block Conjugate Gradient algorithm*, Parallel Comput., 5 (1987), pp. 127 – 139.

[41] M. Parks, E. de Sturler, G. Mackey, D. Johnson, and S. Maiti, *Recycling Krylov subspaces for sequences of linear systems*, SIAM J. Sci. Comput., 28 (2006), pp. 1651–1674.

[42] M. Parks, K. Soodhalter, and D. Szyld, *A block recycled GMRES method with investigations into aspects of solver performance*, Tech. Report. 16-04-04, Temple U., April 2016.

[43] F. Roosta-Khorasani and U. Ascher, *Improved Bounds on Sample Size for Implicit Matrix Trace Estimators*, Found. Comput. Math., 15 (2015), pp. 1187–1212.

[44] Y. Saad, *On the Lanczos method for solving symmetric systems with several right hand sides*, Math. Comp., 48 (Apr. 1987), pp. 651–662.

[45] M. Selig, N. Oppermann, and T. Ensslin, *Improving stochastic estimates with inference methods: Calculating matrix diagonals*, Physical Rev. E, 85 (2012), pp. 021134–9.

[46] V. Simoncini and E. Gallopoulos, *An iterative method for nonsymmetric systems with multiple right-hand sides*, SIAM J. Sci. Comput., 16 (1995), pp. 917–933.

[47] V. Simoncini and E. Gallopoulos, *A hybrid block GMRES method for nonsymmetric systems with multiple right-hand sides*, J. Comput. Appl. Math., 66 (1996), pp. 457–469.

[48] C. Smith, A. Peterson, and R. Mittra, *A Conjugate Gradient algorithm for the treatment of multiple incident electromagnetic fields*, IEEE Trans. Ant. & Propag., 37 (1989), pp. 1490–1493.

[49] A. Stathopoulos and K. Orginos, *Computing and deflating eigenvalues while solving multiple right-hand side linear systems with an application to quantum chromodynamics*, SIAM J. Sci. Comput., 32 (2010), pp. 439–462.

[50] M. Stein, J. Chen, and M. Anitescu, *Stochastic approximation of score functions for Gaussian processes*, Ann. Appl. Stat., 7 (2013), pp. 1162–1191.

[51] G. V. G. Stevens, *On the Inverse of the Covariance Matrix in Portfolio Analysis*, The Journal of Finance, 53 (1998), pp. 1821–1827.

[52] B. Vital, *Etude de quelques méthodes de résolution de problèmes linéaires de grande taille sur multiprocesseur*, PhD thesis, Université de Rennes I, Rennes, Nov. 1990.

[53] W. Gropp, et al., *MPI - The Complete Reference, The MPI Extensions*, vol. 2, MIT Press, 1998.

[54] L. Wu, J. Laeuchli, V. Kalantzis, A. Stathopoulos, and E. Gallopoulos, *Estimating the trace of the matrix inverse by interpolating from the diagonal of an approximate inverse*, J. Comput. Phys., 326 (2016), pp. 828 – 844.