A highly 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‡, Yousef Saad*
*Department of Computer Science and Engineering, University of Minnesota, MN 55455, USA, {kalan019,saad}@umn.edu
†Foundations of Cognitive Solutions, IBM Research – Zurich, Switzerland, {acm,bek,cur}@zurich.ibm.com
‡CEID, University of Patras, 26504 Patras, Greece & Dept. CSE, University of Minnesota, MN 55455, USA, stratis@ceid.upatras.gr

***Abstract*—We describe PP-BCG, a parallel iterative solver for the solution of dense and symmetric positive-definite linear systems with multiple right-hand sides suitable for MPPs. Such linear systems appear in the context of stochastic estimation of the diagonal of the matrix inverse in Uncertainty Quantification and the trace of matrix products in statistical analysis. We propose a novel numerical scheme based on the block Conjugate Gradient algorithm combined with Galerkin projections and describe its implementation. We test the method on model covariance matrices from uncertainty quantification, where the solution of the linear systems is typically used to estimate the diagonal of the matrix inverse. Numerical experiments on an MPP illustrate the performance of the proposed scheme in terms of efficiency and convergence rate, as well as its effectiveness relative to block Conjugate Gradient and the Cholesky-based ScaLAPACK solver. In particular, on a 4 rack BG/Q with up to 65536 cores using dense matrices of order as high as 0.5×10^6 and 800 rhs, PP-BCG is 2-3 times faster.**

***Keywords*-Linear systems, multiple right-hand sides, block Conjugate Gradient, Galerkin projections, distributed memory environments, uncertainty quantification**

I. Introduction

We present a parallel solver for large and dense symmetric positive-definite (SPD) linear systems with multiple right-hand sides (mrhs). Our interest is in using this solver as a kernel in applications from data analytics on state-of-the-art massively parallel processors (MPPs). A distinguishing feature of some of these applications is that only moderate accuracy is sought, which makes viable the use of Monte Carlo type stochastic estimation methods based on matrix equations [2], [5]. In uncertainty quantification (UQ) for example, which is the application of interest here, one seeks the diagonal of the inverse of a covariance matrix that is SPD; the elements of the diagonal quantify the degree of confidence in the data collection and their estimation involves solving relatively well-conditioned SPD linear systems with mrhs [4], [25]. In a related application, the diagonal of the so called "hat matrix" provides the statistical leverage scores that are of interest in many applications and in the design of randomized numerical linear algebra algorithms [16]. Another case in data analytics where SPD systems with mrhs appear is when Gaussian process modeling is used in statistical analysis. In particular, the determination of covariance structure via maximum likelihood estimation leads to the need to estimate the trace of the product of the inverse of a matrix with a block of vectors [36]. Historically, of course, solving linear systems with mrhs attracted attention primarily because of their importance in large scale simulations from electromagnetics, electronic structures, physics, mechanics, etc. [7], [8], [15], [17], [34]; see also [3] for recent software in the Trilinos package.

A general form of the target problem is to solve

$$AX^{(j)} = Z^{(j)},\ j = 1, 2, \ldots, \delta, \tag{1}$$

where $A \in \mathbb{R}^{n \times n}$ is dense and SPD though not necessarily explicitly available, $Z^{(j)} \in \mathbb{R}^{n \times p}$ is the j'th batch of p right-hand sides (rhs), and δ is some unknown positive integer. The batch-like form of (1) arises from the fact that in several applications only a few rhs are available at a time; for example, in stochastic estimation, the least sufficient number of samples (rhs) is typically not known a priori. Thus, only a few samples, e.g., $p \ll n$, are generated and solved at each step, the procedure repeating until the approximation is considered accurate enough[1]. Moreover, the huge size of modern data collections in analytics, as well as the unprecedented rates by which new data is generated, lead to large problems for which it is imperative to develop low complexity, scalable solvers.

By and large, there are two major opportunities when designing solvers for (1). The *first*, we term "computational", is related to the underlying computational model. In particular, it refers to code restructuring in order to obtain better performance, for example using operations on blocks rather than single vectors in order to exploit the memory hierarchy, using bulk communication, etc.; see e.g. [18]. The *second* opportunity we refer to (somewhat generically) as "mathematical", because it is related to the extraction, sharing and reusing information (in the terminology of [33]). In direct methods, this is well known and part of standard software, since the expensive (in this case, Cholesky) factorization step is independent of the rhs and performed only once, and the triangular factors reused repeatedly or independently until all solutions are computed [6], which can amortize the cost of the factorization. On the other hand, in the cases of interest here this approach is no longer practical. Firstly, the matrices can be very large and the cubic cost of the factorization prohibitive and also not sufficiently amortized without enough batches of rhs. Secondly, in several applications the matrix

[1] In general, p could also vary, though we leave this for future research.

A is not explicitly available but only in action, via MVs. Thirdly, a direct method like Cholesky does not permit early stopping to relax the accuracy of the computed results and trade it for smaller time to solution, something that might be desirable in some data analytics applications [12]. In order to address these difficulties and reduce the complexity of the problem, it becomes necessary to deploy matrix-free iterative schemes, like Krylov subspace methods (KSMs) [31]. KSMs generate a sequence of approximate solutions by augmenting a subspace (in this case, the Krylov subspace) at each new iteration. Not surprisingly, information sharing becomes in general more complex (with few exceptions, e.g. when the rhs are linearly dependent or almost so, in which case one can easily approximate part of the solution in terms of the other part) involving the matrix eigenstructure, analysis on subspaces, etc.

In spite of the large and growing body of literature on KSMs for mrhs (see [20], [30] for recent reviews and bibliographical surveys) there have been very few studies on the interplay between these opportunities and a study of tradeoffs. Notable exceptions are [30] that explores nonsymmetric block methods and recycling together with data movement and cache efficiencies for serial architectures, and [8] that describes deflation for nonsymmetric solvers with experiments solving large sparse systems with up to 128 rhs on a 1024 core multiprocessor. We attribute this scarcity in contributions to the fact that (1) is readily parallelizable. This, however, misses entirely the opportunity for information sharing. As we will see and was noted early on in [19] by careful designing the parallel algorithm, the rewards are substantial.

In our case, we can solve each batch $AX^{(j)} = Z^{(j)},\ 1 \leq j \leq \delta$ by the Conjugate Gradient method (CG) [23], the best known KSM for SPD matrices. In the context of UQ, applying CG simultaneously to all available rhs, that is in a "pseudo"-block approach using a term from [3], and combining with iterative refinement was shown to scale well in MPPs [4]. Other CG-based parallel schemes, like the multistep CG [14], communication-avoiding CG [24], and pipelined CG [21], have also had efficient implementations, the focus however was on reducing communication in solving for a single rhs. While the cost of applying CG to each different batch of rhs can typically be much smaller than that of the direct solver (at each iteration the dominant cost comes from the MV products which for unstructured SPD systems runs at $O(n^2)$), the total computational cost might still be prohibitive, especially if many batches of rhs need to be solved. It is then natural to consider KSMs that take advantage of the computational effort invested when solving $AX^{(j)} = Z^{(j)},\ j = 1, \ldots, \overline{j}$, in order to accelerate the convergence of the method in subsequent batches $AX^{(j)} = Z^{(j)},\ j = \overline{j}+1, \ldots, \delta$. When p>1, information sharing in KSMs occurs naturally when deploying a block method such as block CG (BCG) on each batch $AX^{(j)} = Z^{(j)},\ j = 1, \ldots, \delta$, since, compared to standard CG, a p-times larger Krylov subspace is built for each individual solution, and thus faster convergence is obtained per batch. Moreover, each matrix-matrix product (MM) in BCG requires only one pass over A, thus reducing the memory references of standard CG; cf. the discussions in [13], [20], [26], [28], [30], [37] for block methods and recycling. However, BCG is designed to only share information among rhs within the same batch. Therefore, for small p is the performance of such a scheme for solving (1) would be similar to CG. Some proposals to address this issue and accelerate the overall performance of a BCG-based approach already exist. In [11] it is suggested to use deflation of invariant subspaces, and in [29], [35] to use spectral recycling. These techniques, however, do not consider the implications of massive parallelism and a study of their scalability remains an interesting open issue. Our goal, is to go beyond and consider the mathematical and computational opportunities for solving for mrhs in an MPP implementation.

Contributions of this paper

We present a solver suitable for large dense linear SPD systems with mrhs that is specifically designed to exploit MPPs and also makes use of information sharing ideas in order to accelerate the solution process. It combines partial recycling of Krylov subspaces with BCG and is related to earlier work in [32] and the block seed approach in [10]. The proposed algorithm also extends and improves prior work of the authors ([25]) as it explicitly addresses the needs of a parallel implementation together with more effective information sharing. In the absence of problem specific information (e.g. special structure as in [39]), this method returns better performance when solving large matrix equations like (1) compared to well-known baseline methods. The paper is organized as follows. In Section II we present the proposed solver, abbreviated as PP-BCG. In Section III we describe its parallel implementation and give details on various design aspects as well as a cost-benefit analysis. In Section IV we present experiments on an MPP platform and comparisons with other solvers. Concluding remarks are provided in Section V.

Specific contributions of our paper include: i) A novel 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) An implementation on an MPP and experiments with up to 65,536 cores using dense matrices of order as high as 0.5×10^6 and 800 rhs together with performance comparisons relative to BCG and ScaLAPACK. Results show that PP-BCG is 2-3 times faster for these large problems. We note here that due to the relatively good condition of the matrices in the application of interest, preconditioning was not considered.

II. The PP-BCG method

In this section, we outline the proposed method, abbreviated as PP-BCG. The key idea is to recycle the Krylov subspace built in solving $AX^{(1)} = Z^{(1)}$ in order to improve the convergence rate of batches $AX^{(j)} = Z^{(j)},\ j = 2, \ldots, \delta$.

A. Solving for the first batch of rhs

Since we make no structural assumptions other than that the matrix is dense and SPD, we consider BCG to be the best choice for solving the first batch, $AX^{(1)} = Z^{(1)}$ [27]. For the same reasons, we pick as initial $Z^{(1)}$ whatever is available and do not consider seed selection strategies (e.g. as done in [33]).

For future reference, this is listed as Algorithm 1. Matrices $X_0^{(1)} \in \mathbb{R}^{n\times p}$, $R_0^{(1)} = Z^{(1)} - AX_0^{(1)}$, and $P_0 = R_0^{(1)}$ denote the initial guess, initial residual, and initial direction block of BCG, respectively. Throughout this section we assume that matrix A as well as multivectors $X_i^{(1)},\ R_i^{(1)}$ and P_{i-1} are distributed row-wise among the available processors. Matrices

Algorithm 1 Block Conjugate-Gradient (BCG).

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

α_i and β_i are $p \times p$ that (in the absence of roundoff) are calculated to enforce the orthogonality conditions in BCG.

Computing the MM product $T_i = AP_{i-1}$ (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 $(R_i^{(1)})^\top R_i^{(1)}$ in lines 7 and 10 require a reduction operation, each of size p^2. For reasons of stability, $((R_{i-1}^{(1)})^\top R_{i-1}^{(1)})^{-1}$ and $(P_{i-1}^\top T_i)^{-1}$ are computed using the SVD (via LAPACK's DGESV) [1]. The latter allows the (automatic) use of pseudoinverses in case of rank deficiency of $R_{i-1}^{(1)}$, e.g. when the rhs converge at different rates. Another option, would be to use deflation techniques that have been developed primarily for block nonsymmetric solvers, see e.g. [8].

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

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

is the Krylov subspace of dimension i. Ordering the eigenvalues of A as $\lambda_n \ge \ldots \ge \lambda_1 > 0$, the A-norm of the error of the j'th right-hand side after $i-1$ BCG iterations satisfies $\dfrac{\|x_i^{(j)} - x^{(j)}\|_A}{\|x_0^{(j)} - x^{(j)}\|_A} \le 2\left(\dfrac{\sqrt{\kappa_A}-1}{\sqrt{\kappa_A}+1}\right)^i$, where $\kappa_A = \lambda_n/\lambda_p$ [27]. Numerically, therefore, BCG has faster convergence than CG, since the corresponding value of κ_A for the latter is larger, namely λ_n/λ_1. Moreover, BCG provides a computational opportunity for better performance as it can use MM rather than MVs.

The BCG algorithm can be applied to the solution of any subsequent batch $AX^{(j)} = Z^{(j)},\ j = 2, \ldots, \delta$ as in Algorithm 1. However, this does not entail any information exchange between the batch solves and therefore, any computational effort that was invested in solving $AX^{(1)} = Z^{(1)}$ is not reused. From an information exchange point of view, this is suboptimal especially when the blocksize p is small in which case BCG converges at a rate similar to CG. We next describe a mechanism to enable information reuse across batches.

B. Initialization by modified Galerkin projections

Let ζ denote the number of iterations made by BCG during the solution process of $AX^{(1)} = Z^{(1)}$, and let matrices $P_{i-1},\ T_i$ generated by Algorithm 1 be distributed without overlap among the available set of processors, while all of the small $p \times p$ matrices $\alpha_i,\ \beta_i$, and $P_{i-1}^\top T_i,\ i = 1, \ldots, \hat{\zeta}$, are replicated in each available processor. The parameter $\hat{\zeta} \le \zeta$ is user-provided and depends only on the amount of system memory being available to the application.

The convergence rate of BCG applied on any subsequent batch $AX^{(j)} = Z^{(j)},\ j \ge 2$, can be enhanced by computing a non-trivial initial guess $X_0^{(j)}$ through a series of Galerkin projections; see e.g. [10], [25]:

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

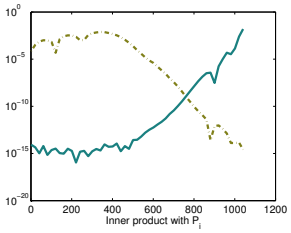
$$\hat{R}_{\hat{\zeta}-i+1}^{(j)} = (I - AP_{i-1}(P_{i-1}^\top T_i)^{-1} P_{i-1}^\top)\hat{R}_{\hat{\zeta}-i}^{(j)},\ i = \hat{\zeta} : -1 : 1. \quad (3)$$

Initially, $\hat{X}_0^{(j)} = 0$ and $\hat{R}_0^{(j)} = Z^{(j)}$. Eqs. (2) and (3) essentially capture the part of the solution of $X^{(j)},\ j \ge 2$, that lies in the Krylov subspace generated by $AX^{(1)} = Z^{(1)}$, under the constraint that $\hat{R}_0^{(j)}$ is made orthogonal to each direction block $P_i,\ i = \hat{\zeta} - 1, \ldots, 0$. Algorithm 2 (GALPROJ) sketches the 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 $P_{i-1}^\top \hat{R}_{\hat{\zeta}-i+1}^{(j)}$ (line 5 of GALPROJ). After computing $H \in \mathbb{R}^{p\times p}$, the update of $\hat{X}_{\hat{\zeta}-i+1}^{(j)}$ and $\hat{R}_{\hat{\zeta}-i+1}^{(j)}$ is performed locally in each processor. In total, each iteration of GALPROJ costs $O(np^2)$, for a combined cost of $O(n\hat{\zeta}p^2)$.

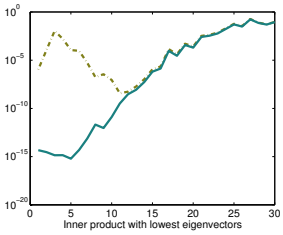
From an HPC viewpoint, small values of p lead to reductions which are dominated by latency. To improve performance, it is possible to deflate τ>1 direction blocks $P_i, \ldots, P_{i-\tau+1}$ simultaneously, this way increasing the granularity of each reduction step and reducing the initialization time of multiple reductions into a single one of size $\tau \times p^2$. For sufficiently small values of τ, the matrix product $\mathcal{Z}$=$[P_i, \ldots, P_{i-\tau+1}]^\top A[P_i, \ldots, P_{i-\tau+1}]$ 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, since in exact arithmetic the direction blocks are A-orthogonal, and in finite precision arithmetic we expect this property to hold locally for matrices $P_i, \ldots, P_{i-\tau+1}$.

Algorithm 2 The Galerkin projections scheme (GALPROJ).

input : $A,\ \hat{X}_0^{(j)},\ \hat{R}_0^{(j)},\ \hat{\zeta}$
output : $\hat{X}_{\hat{\zeta}}^{(j)}$
$i = \hat{\zeta}$
repeat
  $H = (P_{i-1}^\top T_i)^{-1}(P_{i-1}^\top \hat{R}_{\hat{\zeta}-i}^{(j)})$
  $\hat{X}_{\hat{\zeta}-i+1}^{(j)} = \hat{X}_{\hat{\zeta}-i}^{(j)} + P_{i-1}H$
  $\hat{R}_{\hat{\zeta}-i+1}^{(j)} = \hat{R}_{\hat{\zeta}-i}^{(j)} - T_i H$
  $i = i - 1$
until $i == 1$



(a) $\|P_i(P_i^\top P_i)^{-1}P_i^\top \hat{R}_\zeta^{(2)}\|$.



(b) Inner product of $\hat{R}_\zeta^{(2)}$ with the thirty lowest eigenvectors.

Fig. 1: A comparison of two different deflation orderings.

C. Analysis of deflation by modified Galerkin projections

Consider the deflation of the direction blocks $P_0, \ldots, P_{\hat{\zeta}-1}$ in (2) and (3). During the i'th 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 P_{i-1}. Since, in exact arithmetic, the direction blocks are A-orthogonal, we have:

$$\Phi = \Pi_{i=1}^{\hat{\zeta}}\Phi_i, \qquad \Phi\hat{R}_0^{(j)} \perp \{P_0, \ldots, P_{\hat{\zeta}-1}\}. \tag{5}$$

Thus, in exact arithmetic, $\hat{R}_0^{(j)}$ becomes orthogonal to any invariant subspace captured in the Krylov subspace $\mathcal{K}_{\hat{\zeta}} = \{P_0, \ldots, P_{\hat{\zeta}-1}\}$, which explains the enhanced convergence rate when $\hat{X}_{\hat{\zeta}}^{(j)}$ is passed as an initial guess in BCG when the latter is applied on $AX^{(j)} = Z^{(j)}$.

In exact arithmetic, the order of deflation of the direction blocks $P_0, \ldots, P_{\hat{\zeta}-1}$ does not make any difference. However, in finite precision arithmetic, the order of deflation has a large impact. As an example, consider for the moment that the direction blocks are deflated in the order they were generated, i.e., from P_0 to $P_{\hat{\zeta}-1}$. Now, let $\hat{R}_i^{(j)}$ satisfy $\hat{R}_i^{(j)} \perp \{P_0, \ldots, P_{i-1}\}$. At the next step we compute $\hat{R}_{i+1}^{(j)} = \Phi_{i+1}\hat{R}_i^{(j)}$ and thus

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

If $\hat{R}_{i+1}^{(j)}$ is to remain orthogonal to $\{P_0, \ldots, P_{i-1}\}$, we also need that $P_\xi^\top AP_i = 0$ for all $\xi = 0, \ldots, i-1$, which, in general, does not hold. Actually, in finite precision, every time we deflate P_i, components from $P_0, \ldots, P_{i-1}$ re-emerge in $\hat{R}_i^{(j)}$. To ease this effect, we choose to deflate the direction blocks $P_0, \ldots, P_{\hat{\zeta}-1}$ in a reversed manner, as in (2) and (3). While reverse deflation does not eliminate the finite precision effects, it leads to initial guesses of better quality since the leading direction blocks $P_0, P_1, \ldots$ remain better deflated. Fig. 1 illustrates a numerical comparison between the two different deflation strategies for a diagonal test matrix $A_{ii} = i/10^4$, $i = 1, \ldots, 10^4$, where $p = 1$, and $\delta = 2$. We perform $\zeta = 1048$ when solving the linear system $AX^{(1)} = Z^{(1)}$ and then call GALPROJ to compute an initial guess for $AX^{(2)} = Z^{(2)}$, using $\hat{\zeta}$=ζ. The left subfigure plots the orthogonal projection of $\hat{R}_{\hat{\zeta}}^{(2)}$ along each direction block P_i, $i = 0, \ldots, \hat{\zeta}-1$, when deflation is implemented in the natural order (dashed line) and reversed order (solid line). Notice how the direction blocks produced in the early iterations of Algorithm 1 have re-emerged in $\hat{R}_{\hat{\zeta}}^{(2)}$ under the standard ordering. 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 (dashed line) and reversed order (solid line). Since the term $\hat{R}_{\hat{\zeta}}^{(2)}$ obtained by reverse deflation is closer to being orthogonal to these extremal eigenvectors, faster convergence is expected when that initial guess is used in BCG when applied on $AX^{(2)} = Z^{(2)}$. The reverse deflation process that we described represents an important feature of PP-BCG and improves significantly on the work reported in [25].

The overhead involved in the aforementioned deflation scheme presented in this section requires the storage of two sequences of multivectors, T_i and P_{i-1}, $i = 1, \ldots, \hat{\zeta}$; see also Table I. However, this is not an issue for our approach since one can set $\hat{\zeta}$ before-hand based on the amount of system memory being available. While the efficiency of the scheme depends on the portion of the Krylov subspace being deflated, it is guaranteed to work even without any additional memory overhead (in which case the whole scheme would reduce to the standard BCG). In the following we consider a modification of GALPROJ for clusters with limited system memory.

D. Galerkin projections under limited memory scenarios

When memory space is limited, it is possible to perform the Galerkin projections via GALPROJ without storing the direction blocks $P_0, \ldots, P_{\hat{\zeta}-1}$. We next show that these matrix variables can be re-computed on-the-fly each time an initial guess for a new batch of rhs $AX^{(j)} = Z^{(j)}$, $j \geq 2$ is sought.

Assume that during Algorithm 1 we store only matrices $T_i,\ \alpha_i,\ \beta_i,\ i = 1, \ldots, \hat{\zeta}$, as well as matrices $R_{\hat{\zeta}},\ P_{\hat{\zeta}}$. Then using equation $P_{\hat{\zeta}} = R_{\hat{\zeta}}^{(1)} + P_{\hat{\zeta}-1}\beta_{\hat{\zeta}}$ from BCG, we can recover (the unknown) $P_{\hat{\zeta}-1} = (P_{\hat{\zeta}} - R_{\hat{\zeta}}^{(1)})\beta_{\hat{\zeta}}^{-1}$. Generalizing the above concept, each P_i, $i = \hat{\zeta}-1, \ldots, 0$ can be re-generated on-the-fly by using the following set of equations:

$$P_i = (P_{i+1} - R_{i+1}^{(1)})\beta_{i+1}^{-1},\ i = \hat{\zeta}-1, \ldots, 0 \tag{6}$$

$$R_i^{(1)} = R_{i+1}^{(1)} + T_{i+1}\alpha_{i+1},\ i = \hat{\zeta}-1, \ldots, 0. \tag{7}$$

At step i, we only need access to $R_{i+1}^{(1)}$ and P_{i+1}, which are already generated and temporarily stored from the previous step. Moreover, all computations in (6) and (7) are embarrassingly parallel since a local copy of the $p \times p$ matrices $\alpha_i,\ \beta_i,\ i = 1, \ldots, \hat{\zeta}$ is already available at each processor during execution of Algorithm 1. The necessity to store $P_0, \ldots, P_{\hat{\zeta}-1}$ is thus replaced by an additional $O(n\hat{\zeta}p^2)$ operations for each new batch.

E. 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: $X^{(1)} = \mathrm{BCG}(A,\ Z_0^{(1)},\ X_0^{(1)},\ p,\ \mathrm{tol}_1)$
4: ▷ For any $AX^{(j)} = Z^{(j)}$
5: **for** $j = 2, \ldots, \delta$ **do**
6: $\quad \hat{X}_{\hat{\zeta}}^{(j)} = \mathrm{GALPROJ}(A,\ \hat{X}_0^{(j)} \equiv 0,\ Z^{(j)},\ \hat{\zeta})$
7: $\quad X^{(j)} = \mathrm{BCG}(A,\ Z^{(j)},\ \hat{X}_{\hat{\zeta}}^{(j)},\ p,\ \mathrm{tol}_2)$
8: **end for**

PP-BCG scheme is described in Algorithm 3. In each iteration (lines 5-8) an initial guess is generated by exploiting the information stored when solving $AX^{(1)} = Z^{(1)}$ (line 6). This initial guess is then passed to BCG (line 7). Note that the tolerance, tol_1, for the solution of $AX^{(1)} = Z^{(1)}$ can be different than the general tolerance tol_2 used for the solution of the subsequent batches of rhs. This allows us to generate a larger and thus richer Krylov subspace which is essential for PP-BCG to be efficient.

III. Distributed implementation

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 [38]. We assume a 2-D grid of $G = M \times K$ processors, each processor labeled by its row and column position on the grid. We thus write matrices A and $Z^{(j)},\ j = 1, \ldots, \delta$ 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 J'th processor of the first row of the grid (equivalently, processor $(1, J)$).

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 in the 2-D grid. Let matrix $P \in \mathbb{R}^{n \times p}$ be distributed among the K processors of the first row of the 2-D grid. Then, the MM product $T = AP$ can be accomplished in parallel as follows (see also Fig. 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 MM product enables collective operations on 1-D 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 and thus for all operations of PP-BCG except the MM product, e.g., block inner products or block AXPY operations, only a subset of processors of the 2-D grid is active. *In our implementation, this subset is always formed by the K processors in the first row of the 2-D grid*. Thus, performing block inner products requires communication only among the K processors lying in the first row of the 2-D grid.

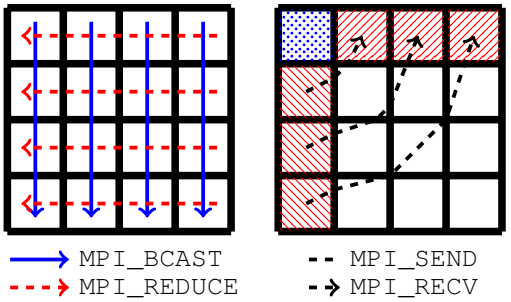


Fig. 2: Schematic sketch of the communication pattern for the MM product in a 4×4 2-D grid of processors.

We also note that for completeness PP-BCG was also implemented for 1-D processor grids. However, this leads to MM products which are less scalable than in the 2-D case, since having all $M \times K$ MPI processes under the same communicator increases the communication volume.

Table I shows the leading memory requirements of each phase of PP-BCG. Variables n_I and n_J denote the number of rows and columns of A assigned to the (I, J) processor, respectively. If memory resources are limited, we have the option to store matrices T_i only, and then locally generate P_{i-1} on the fly (as described in subsection II-D), an approach denoted by the '*' superscript.

Table II shows the per processor computational complexity (per iteration) for all different phases. The main computational cost is caused by the need to perform the local MM product $T = AP$ and runs at $O(n_I n_J)$. In contrast, all other operations at any other phase of PP-BCG cost $O(n_J)$.

TABLE I: Memory complexity per processor and phase. GALPROJ* denotes the limited memory version of GALPROJ described in Section II-D.

Phase	Memory requirements
BCG (Alg. 1)	$n_I n_J + 4n_J p$
GALPROJ (Alg. 2)	$2n_J p + 2n_J \hat{\zeta} p$
GALPROJ*	$4n_J p + n_J \hat{\zeta} p$

TABLE II: Computational complexity per processor and iteration. GALPROJ* denotes the limited memory version of GALPROJ described in Section II-D.

Phase	Computational complexity
BCG (Alg.1)	$n_I p(2n_J - 1) + 6n_J p^2 + 3n_J p + 4n_J p^2$
GALPROJ (Alg. 2)	$2(2n_J p^2 + n_J p) + 2n_J p^2$
GALPROJ*	$4(2n_J p^2 + n_J p) + 2n_J p^2$

A. Communication cost of information sharing

We next model the communication cost of GALPROJ which is the process which implements the information sharing via Eqs. (2) and (3). For the purposes of our analysis, the communication cost is approximated by a linear model, e.g [9], [18]

$$t_{\text{comm}} = \ell + \mu q, \tag{8}$$

where ℓ is the startup cost (latency), μ is the message size (measured in terms of double-precision 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 processors of the 2-D grid has its own communicator, the K processors lying in the first row of the grid communicate independently and in a binary tree network topology.

Each time we deflate one (or more) direction block(s) from the residual of each new batch of rhs, 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 thus a single deflation step of GALPROJ will introduce a communication overhead equal to

$$t_\tau = \lceil \log(K) \rceil (2\ell + 2\tau p^2 q + \tau p^2 \gamma), \tag{9}$$

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 [9]. Thus, the total communication overhead introduced by the Galerkin projections in GALPROJ for all $\delta - 1$ subsequent batches of p rhs is

$$t_{\text{comm}}^{(\delta,\tau)} = \frac{(\delta - 1)\hat{\zeta} t_\tau}{\tau}. \tag{10}$$

Fig. 3 quantifies the above discussion on a synthetic experiment, when $n = 64000$. We assume that $\ell = 10^4 q$, i.e., the latency (startup) cost is 10^4 larger than the cost to transmit a double-precision scalar, which is typical for current

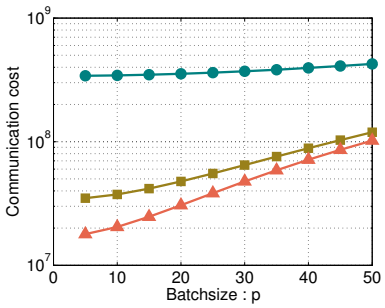


Fig. 3: Communication cost of GALPROJ, as modeled by Eq. (10). "●", "■" and "▲" represent PP-BCG using $\tau = 1$, $\tau = 10$ and $\tau = 20$, respectively.

architectures. To account for different network architectures, we did not set an actual value for q. The x-axis runs across different values of p, while the y-axis shows the modeled communication cost as determined by (10). The above analysis shows that deflating one direction block at a time (τ=1) leads to larger communication cost than when more direction blocks are deflated simultaneously, with larger values of τ leading to better performance. As τ increases, the communication pattern shifts from latency-dominated to bandwidth-dominated. Moreover, as p increases, the communication pattern also becomes bandwidth-dominated and the choice of τ will have reduced impact.

IV. EXPERIMENTS AND PERFORMANCE EVALUATION

In this section we present experiments performed in distributed computing environments. PP-BCG was implemented in Fortran 90. All local MM multiplications were performed using the BLAS-3 `DGEMM` routine in IBM's ESSL.

For the rest of this section we set $\text{tol}_1 = 10^{-12}$, $\text{tol}_2 = 10^{-6}$, and $\hat{\zeta} = 200$. While setting $\tau > 1$ can lead to an improved performance, its choice enables a non-trivial numerical study which requires a lengthier exposition. All of our experiments were performed using $\tau = 1$. Moreover, we consider 2-D grids of processors $G := (M, K)$ in which M and K are related as $K = M$ and $K = 2M$. All computations were performed in 64-bit arithmetic.

A. Computational system and test matrices

Experiments were performed on an MPP consisting of up to 4 racks of an IBM BlueGene/Q[2] (BG/Q) supercomputer [22]. A rack consists of 1024 compute nodes, each 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

[2]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.

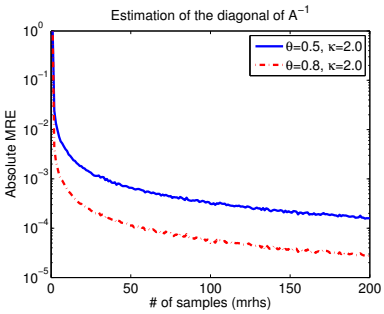


Fig. 4: MRE for Monte Carlo stochastic estimator in (12) for model covariance matrix with $n = 8192$, values $\theta = 0.5, 0.8$.

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[3] version 14.1.13. As in [2], [5], [25], we used a synthetic dataset consisting of model covariance matrices, generated to simulate real-case scenarios in data UQ, specifically

$$A_{ii} = 1 + i^{\theta}, \quad A_{ij} = 1/|i-j|^{\kappa} \ (if \ i \neq j), \quad i,j = 1,\ldots,n, \tag{11}$$

for real θ and $\kappa = 2$. The progressive decay away from the main diagonal simulates the fact that features are locally-only correlated. The condition number of these matrices is known to scale like n^{θ}. Each batch $Z^{(j)}$ was formed by p n-dimensional vectors, each vector having entries ± 1 with equal probability (Rademacher variables). We also experimented with a more general dataset, generated by imposing additional symmetric perturbations (maintaining the SPDness) at random positions of (11), that is $A_{\hat{i}\hat{j}} = A_{\hat{i}\hat{j}} + \delta_{\hat{i}\hat{j}}$, $\delta_{\hat{i}\hat{j}} = 100/|\hat{i}-\hat{j}+1|$, 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 (11) 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.

B. A case study and numerical considerations

The application of interest is the approximation of the diagonal of a matrix that is only available via MV's using the so called Hutchinson estimator. The diagonal estimation method was first proposed in [5] and consists of approximating the diagonal of A, say $\mathcal{D}(A^{-1})$, via

$$\mathcal{D}_{\delta}(A^{-1}) := [\sum_{j=1}^{\delta} Z^{(j)} \odot X^{(j)}] \oslash [\sum_{j=1}^{\delta} Z^{(j)} \odot Z^{(j)}], \tag{12}$$

where $X^{(j)} = A^{-1}Z^{(j)}$ and the symbols $\odot, \oslash$ denote element-wise multiplication and division, respectively. Fig. 4 plots the mean relative error (MRE) of the estimator in (12) for a small

[3]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`

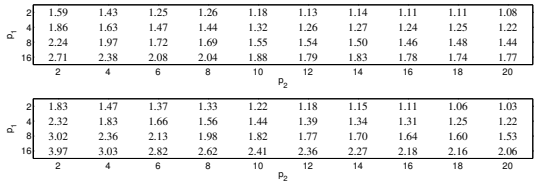


Fig. 5: Speedup in terms of iterations of BCG over PP-BCG for $AX^{(2)} = Z^{(2)}$ with p_2 rhs. Top: $\theta = 0.5$. Bottom: $\theta = 0.8$.

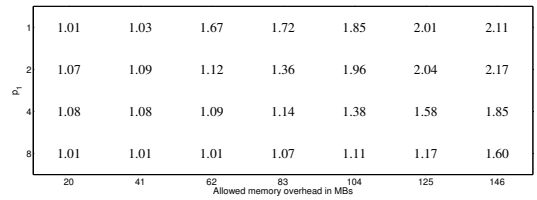


Fig. 6: Memory overhead in PP-BCG vs. speedup over BCG for $p_1 = 1,\ 2,\ 4$ and $p_1 = 8$ with $p_2 = 8$, $n = 32768$, $\theta = 0.8$.

scale model covariance matrix of size $n = 8192$ as the number of samples (rhs) increases. As was extensively discussed in [25], the fast solution of (1) is critical for the success of stochastic diagonal estimation since for estimators with large variance, the number of rhs that must be solved might be of the order $\mathcal{O}(10^3)$.

We next study how the PP-BCG and BCG schemes compare if different batch sizes were allowed, i.e., the i'th batch has p_i rhs. For simplicity consider only two batches, $AX^{(1)} = Z^{(1)}$ and $AX^{(2)} = Z^{(2)}$, each one with p_1 and p_2 rhs, respectively. Fig. 5 shows the speedup of PP-BCG over BCG (in terms of iterations to reach convergence) as both p_1 and p_2 vary. We used the same matrices as in Fig. 4. The top subfigure considers the case $\theta = 0.5$ while the bottom subfigure the case $\theta = 0.8$. The performance gap between the two methods increases as $p_1 \gg p_2$, since in that case the convergence rate acceleration offered by the Galerkin projections can be much higher that the acceleration offered by BCG. On the other hand, as p_2 becomes larger than p_1 the convergence rate of $AX^{(2)} = Z^{(2)}$ is affected more by the blocksize of BCG and less by the initial guess obtained by the Krylov subspace built by $AX^{(1)} = Z^{(1)}$. As a result, PP-BCG and BCG converge similarly.

In the previous example, the matrix equation $AX^{(1)} = Z^{(1)}$ was always solved for the same accuracy $\text{tol}_1 = 10^{-12}$ irrespectively of the batch size p_1. Thus, the amount of memory overhead used by PP-BCG was not fixed as p_1 varied. It is then of interest to consider how should the value of p_1 be chosen so that $AX^{(j)} = Z^{(j)},\ j = 2,\ldots,\delta$ converge as fast as possible when storage is limited. For this we prefer to generate a Krylov subspace of the highest possible dimension, i.e., building the Krylov subspace using $p_1 = 1$. Fig. 6 demonstrates the above observation using the model covariance matrix (11) with $n = 32768$ and $\theta = 0.8$. We used two batches of rhs with p_1 varying and $p_2 = 8$. For any fixed memory overhead allowed

in PP-BCG, the highest speedups over BCG were obtained when $p_1 = 1$ or $p_1 = 2$. This experiment reveals that with limited memory, one should store only the matrices T_i in Algorithm 1 and locally regenerate the P_{i-1} matrices. This is because storing only the T_is there is enough space to store a Krylov subspace whose dimension is approximately doubled.

C. Runtimes and efficiency of PP-BCG

We used three model covariance matrices as in (11), setting $n = 131072, 262144$ and $n = 524288$, for $\theta = 0.6$ and $\theta = 0.8$. For each n, we solved for a total of $s = 800$ rhs, divided in batches of size $p = 20,\ 40$ and $p = 80$. For $n = 131072$ we used $2^\nu, \nu = 4, ..., 10$ compute nodes (i.e. up to one rack). For $n = 262144$ the values of ν were $\nu = 6, ..., 12$, while for $n = 524288$ they were $\nu = 8, ..., 12$ (up to four BG/Q racks). Since each BG/Q node features 16 compute cores devoted to computation, the total number of cores used were 16384 (for $n = 13107$) and 65536 (for the larger matrices). The number of MPI processes was always equal to the number of compute cores and each MPI process utilized 2 hardware threads to provide maximum bandwidth.

Fig. 7 illustrates the strong scalability plotting runtimes for all combinations of $n,\ p$ and θ (log-log scale) as the compute nodes increase. The runtimes include all different phases of PP-BCG, i.e., obtaining initial guesses and solving for all batches. Observe that larger values of p lead to reduced runtimes for all matrix sizes and condition numbers. We will see in Section IV-D that this behavior is due to the faster convergence per rhs of PP-BCG when higher values of p are used. Fig. 8 plots the efficiency of PP-BCG scheme. Assuming a reference baseline execution time t_r on a baseline number of G_r MPI processes, 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 all runs follow two different regimes as can be observed by comparing Fig. 8 with Fig. 9 which reports the efficiencies achieved for MM. For smaller numbers of compute nodes, the runtime of PP-BCG is mostly spent on MM products and thus runs at very high efficiencies, commensurate with those achieved by MM, which run at almost perfect efficiency. As the number of compute nodes grows, non-MM 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 grid of processors (this is also illustrated in Fig. 10 where we plot the computational profile of the smallest matrix $n = 131072$ 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 grid.

Overall, as the compute nodes double for the regimes we have explored, the average observed efficiencies range from 85 to 90%. This demonstrates the high scalability of PP-BCG.

One way to increase the efficiency of PP-BCG when non-MM operations dominate the computational profile is to use an $M \times K$ grid of processors with $K \gg M$. However, in this case the efficiency of the MM operation would not remain the same since the 2-D grid would be "stretched" and become more similar to 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 $\rho > 1$. Under a limited memory regime, from a numerical perspective it is better to store only matrices T_i and regenerate the $P_{i-1},\ i = 1, \ldots, \hat{\zeta}$ on-the-fly (as shown in Fig. 6); from our HPC perspective, however, we opt to store these factors even at the price of deflating a smaller Krylov subspace, e.g. halving $\hat{\zeta}$. This is because regenerating the P_is utilizes only a subset of the available processors. Fig. 11 illustrates the performance of the PP-BCG scheme in terms of Tflops per second (TF/s). Results shown are for $\theta = 0.6$.

TABLE III: Average number of iterations per batch of p rhs during PP-BCG and speedup over BCG.

	n = 131072			**n = 262144**			**n = 524288**		
batch size (p)	20	40	80	20	40	80	20	40	80
PP-BCG iterations									
$\theta = 0.6$	50	40	35	87	67	60	113	98	83
$\theta = 0.8$	87	67	60	111	86	64	141	119	92
PP-BCG speedup over BCG									
$\theta = 0.6$	1.70	1.78	1.75	1.92	2.14	2.01	2.31	2.45	2.41
$\theta = 0.8$	2.18	2.25	2.14	2.32	2.42	2.31	2.65	2.71	2.67

D. Comparisons with BCG and ScaLAPACK

The only approach we are aware that has been tested in sizes and computing installations similar to ours is the mix of "pseudo"-block CG and iterative refinement presented in [4]. Given that in this paper we only consider double precision arithmetic, we found that BCG can be 1.5-2 times faster than the method in [4], especially as the condition number increases.

Table III shows the average number of iterations per batch of p rhs in PP-BCG, as well as the speedup of PP-BCG over BCG in terms of average number of iterations per batch of p rhs for the model covariance matrices in (11). Summarizing the results we can observe that larger values of p lead to fewer iterations per batch, since in the latter case the BCG part of PP-BCG converges faster. Moreover, PP-BCG offers greater speedup for problems that are less well-conditioned. Also, as the value of p increases after a certain value, the speedup ratio of PP-BCG over BCG starts declining (as is the case when we shift from $p = 40$ to $p = 80$). Similar behavior is observed for the perturbed model covariance matrices in Table IV with the difference that the perturbation leads to slightly more challenging linear systems.

We also compare with the ScaLAPACK distributed memory Cholesky-based solver [6]. In that case, the covariance matrix A was distributed along a two-dimensional block-cyclic manner among the processors of the 2-D grid. The computational blocksize in `PDPOTRF` was held the same for both dimensions of the 2-D grid and, after some experimental tuning, was set to $m_b = 128$. Fig. 12 presents comparisons for the matrix of size $n = 262144$ when using a fixed number of 16384 MPI

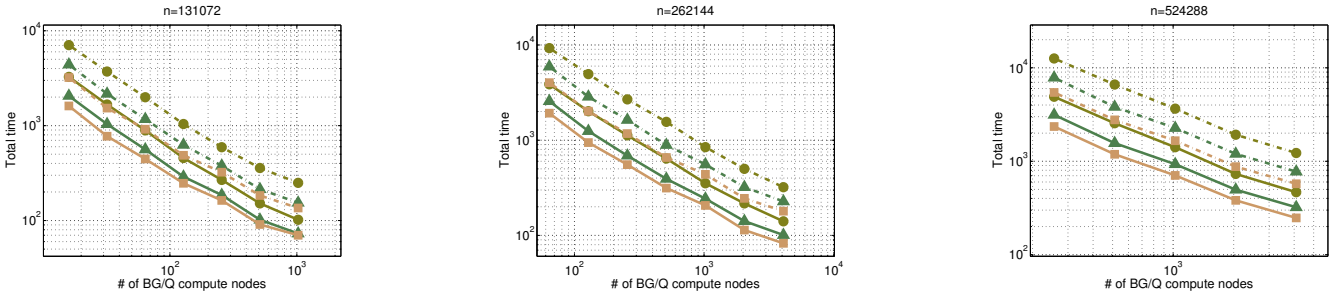


Fig. 7: Runtimes of the PP-BCG solver. • : $p = 20$, ▲ : $p = 40$, ■ : $p = 80$. Solid lines: $\theta = 0.6$. Dashed lines: $\theta = 0.8$.

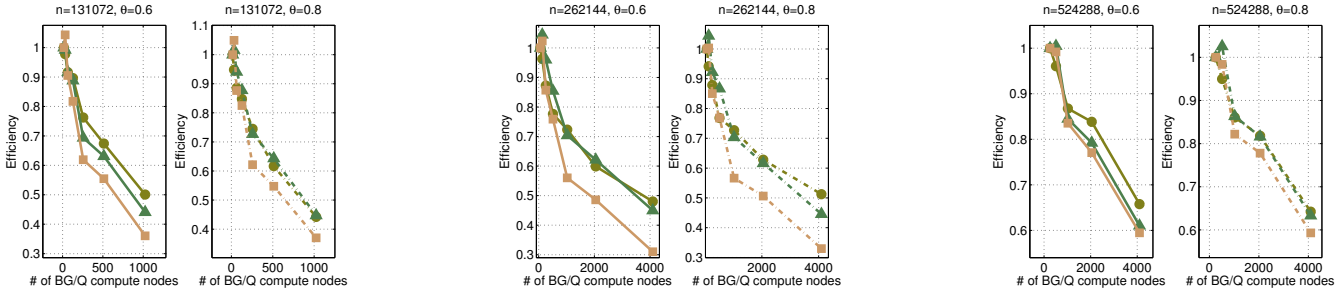


Fig. 8: Efficiency of the PP-BCG solver: • : $p = 20$, ▲ : $p = 40$, ■ : $p = 80$. Solid lines: $\theta = 0.6$. Dashed lines: $\theta = 0.8$.

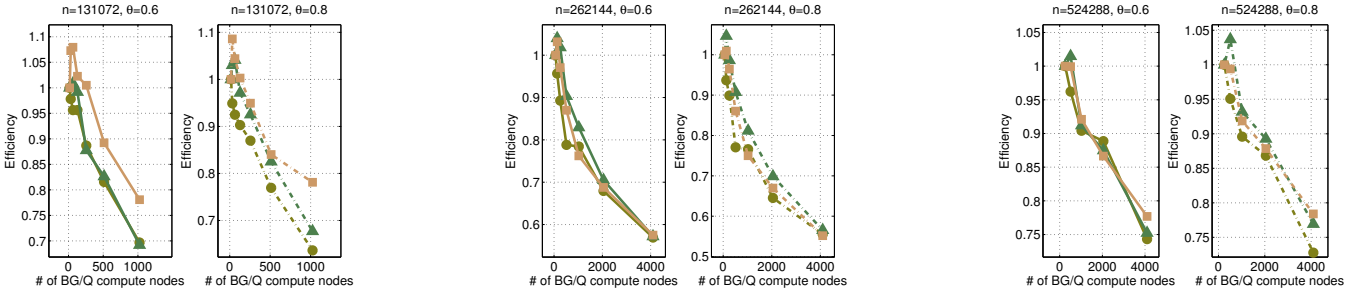


Fig. 9: Efficiency of the MM product. • : $p = 20$, ▲ : $p = 40$, ■ : $p = 80$. Solid lines: $\theta = 0.6$. Dashed lines: $\theta = 0.8$.

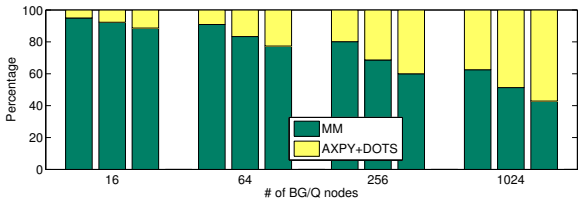


Fig. 10: Performance profile of PP-BCG in terms of compute primitives for $n = 131072$, $\theta = 0.6$. For each ensemble of compute nodes, the leftmost, middle and rightmost bars correspond to $p = 20, 40$ and 80, respectively.

processes and varying δ, θ and p. The left subfigure shows the solution times of ScaLAPACK and PP-BCG as the number of right-hand sides varies. While the direct solver is practically oblivious to the number of rhs solved, PP-BCG is affected by this parameter in a linear manner. Thus, depending on the condition number of A, there is a value of δ beyond which is more efficient to use the direct solver instead of PP-BCG, as can be also seen in the right subfigure of Fig. 12 where

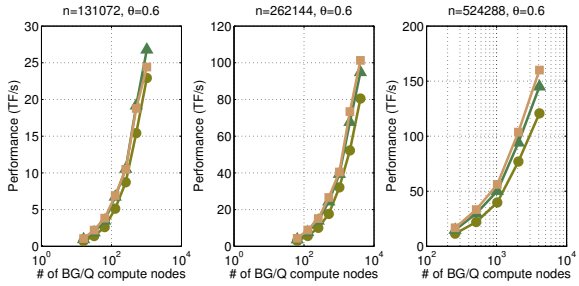


Fig. 11: Performance (TF/s). • : $p = 20$, ▲ : $p = 40$, ■ : $p = 80$.

TABLE IV: Avg. # iterations/p rhs for PP-BCG & speedup over BCG for symmetrically perturbed model covariance matrices.

	n = 32768			**n = 65536**			**n = 131072**		
batch size (p)	20	40	80	20	40	80	20	40	80
PP-BCG iterations									
$\theta = 0.6$	58	44	36	68	54	42	82	78	70
$\theta = 0.8$	76	59	48	104	78	58	130	117	102
PP-BCG speedup over BCG									
$\theta = 0.6$	1.75	2.00	2.04	1.95	1.89	1.98	1.80	1.74	1.83
$\theta = 0.8$	2.21	2.34	2.41	2.15	2.25	2.11	2.27	2.04	1.92

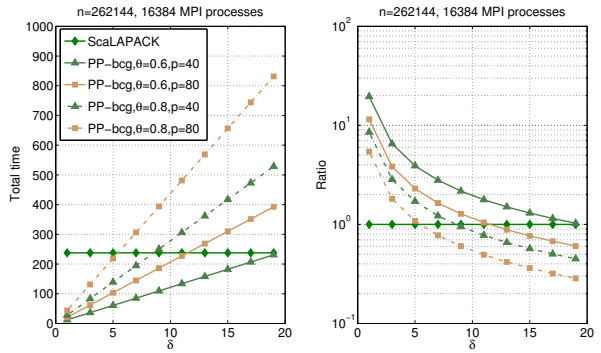


Fig. 12: Comparison of ScaLAPACK and PP-BCG as δ, θ and p vary. (Left): runtimes; (Right) speedup over ScaLAPACK.

we plot the speedup ratio of PP-BCG over ScaLAPACK as the number of total batches to be solved varies.

V. Conclusions - Future directions

We analyzed the numerical aspects of PP-BCG and described its parallel implementation and a performance model for distributed memory 2-D processor grids. Experiments on a BG/Q MPP illustrate the superiority of the method compared to well known approaches when solving large linear systems with model covariance matrices and mrhs from a UQ application. The code is available from the authors upon request. We are currently pursuing a study of the techniques that we described on co-processor systems in combination with mixed precision and iterative refinement. We are also considering the extension of these techniques for applications in other areas and the study of the implications of special matrix structure such as sparsity on the overall performance.

Acknowledgments

We thank Jie Chen, George Karypis, Giorgos Kollias, Eugenia Kontopoulou and Andreas Stathopoulos for their input.

References

[1] E. Anderson et al., *LAPACK Users' Guide*, SIAM, 3d ed., 1999.
[2] C.M. 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.
[3] E. Bavier, M. Hoemmen, S. Rajamanickam, and H. Thornquist, *Amesos2 and Belos - Direct and iterative solvers for large sparse linear systems.*, Scientific Programming, (2012).
[4] C. Bekas, A. Curioni, and I. Fedulova, *Low-cost data uncertainty quantification*, Concur. Comput.: Pract. Exper., 24 (2012), pp. 908–920.
[5] C. Bekas, E. Kokiopoulou, and Y. Saad, *An estimator for the diagonal of a matrix*, Appl. Numer. Math., 57 (2007), pp. 1214 – 1229.
[6] L.S. Blackford et al., *ScaLAPACK Users' Guide*, SIAM, 1997.
[7] H. Calandra et al., *Flexible variants of block restarted GMRES methods with application to geophysics*, SIAM J. Sci. Comput., 34 (2012), pp. A714–A736.
[8] ———, *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. Comp., 35 (2013), pp. S345–S367.
[9] 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.
[10] T.F. Chan and W.L. Wan, *Analysis of projection methods for solving linear systems with multiple right-hand sides*, SIAM J. Sci. Comput., 18 (1997), pp. 1698–1721.
[11] J. Chen, *A deflated version of the block conjugate gradient algorithm with an application to Gaussian process maximum likelihood estimation*, Tech. Report ANL/MCS-P1927-0811, Argonne Nat'l. Lab., 2011.
[12] ———, *How accurately should I solve linear systems when applying the Hutchinson trace estimator?*, SIAM J. Sci. Comput., (to appear).
[13] J. Chen, T.L.H. 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.
[14] A.T. Chronopoulos and C.W. Gear, *s-step iterative methods for symmetric linear systems*, JCAM, 25 (1989), pp. 153 – 168.
[15] M. Clemens, M. Helias, T. Steinmetz, and G. Wimmer, *Multiple right-hand side techniques for the numerical simulation of quasistatic electric and magnetic fields*, JCAM, 215 (2008), pp. 328 – 338.
[16] P. Drineas, M. Magdon-Ismail, M.W. Mahoney, and D.P. Woodruff, *Fast approximation of matrix coherence and statistical leverage*, JMLR, (2012), pp. 3475–3506.
[17] C. Farhat, L. Crivelli, and F.-X. Roux, *Extending substructure based iterative solvers to multiple load and repeated analyses*, Comput. Methods Appl. Mech. Engrg., 117 (1994), pp. 195–209.
[18] E. Gallopoulos, B. Philippe, and A.H. Sameh, *Parallelism in Matrix Computations*, Springer, 2015.
[19] 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.H.V. Topping and M. Papadrakakis, eds., Civil-Comp Press, Edinburgh, 1994, pp. 47–51.
[20] A. Gaul, *Recycling Krylov subspace methods for sequences of linear systems: Analysis and applications*, PhD thesis, TU Berlin, 2014.
[21] P. Ghysels and W. Vanroose, *Hiding global synchronization latency in the preconditioned conjugate gradient algorithm*, Parallel Comput., 40 (2014), pp. 224 – 238.
[22] M. Gilge, *IBM System Blue Gene Solution: Blue Gene/Q Application Development*, IBM Int'l. Tech. Supp. Org., 2nd ed., June 2013.
[23] M.R. Hestenes and E. Stiefel, *Methods of conjugate gradients for solving linear systems*, J. Res. NBS, 49 (1952), pp. 409–436.
[24] M. Hoemmen, *Communication-Avoiding Krylov Subspace Methods*, PhD thesis, University of California, Berkeley, 2010.
[25] V. Kalantzis, C. Bekas, A. Curioni, and E. Gallopoulos, *Accelerating data uncertainty quantification by solving linear systems with multiple right-hand sides*, Numer.Alg., 62 (2013), pp. 637–653.
[26] 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.
[27] D.P. O'Leary, *The block conjugate gradient algorithm and related methods*, Lin. Alg. Appl., 29 (1980), pp. 293 – 322.
[28] ———, *Parallel implementation of the block conjugate gradient algorithm*, Parallel Comput., 5 (1987), pp. 127 – 139.
[29] M.L. Parks et al., *Recycling Krylov subspaces for sequences of linear systems*, SIAM J. Sci. Comput., 28 (2006), pp. 1651–1674.
[30] M.L. Parks, K.M. Soodhalter, and D.B Szyld, *A block recycled GMRES method with investigations into aspects of solver performance*, Tech. Report. 16-04-04, Temple U., April 2016.
[31] Y. Saad, *Iterative Methods for Sparse Linear Systems*, SIAM, 2003.
[32] Y. Saad, *On the Lanczos method for solving symmetric systems with several right hand sides*, Math. Comp., 48 (Apr. 1987), pp. 651–662.
[33] 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.
[34] C.F. Smith, A.F. 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.
[35] 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.
[36] M.L. Stein, J. Chen, and M. Anitescu, *Stochastic approximation of score functions for Gaussian processes*, Ann. Appl. Stat., 7 (2013), pp. 1162–1191.
[37] 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.
[38] W. Gropp, et al., *MPI - The Complete Reference, The MPI Extensions*, vol. 2, MIT Press, 1998.
[39] J. Xia, Y. Xi, S. Cauley, and V. Balakrishnan, *Fast sparse selected inversion*, SIMAX, 36 (2015), pp. 1283–1314.