Journal of Parallel and Distributed Computing 163 (2022) 147–155

Contents lists available at ScienceDirect

Journal of Parallel and Distributed Computing

www.elsevier.com/locate/jpdc

Pipelined Preconditioned Conjugate Gradient Methods for real and complex linear systems for distributed memory architectures

Manasi Tiwari [*], Sathish Vadhiyar

Department of Computational and Data Sciences, Indian Institute of Science, Bangalore, 560012, India

ARTICLE INFO

Article history:
Received 3 July 2021
Received in revised form 22 December 2021
Accepted 19 January 2022
Available online 29 January 2022

Keywords:
Preconditioned Conjugate Gradient
Pipelining
Overlapping communication and computations
Complex Hermitian positive definite systems
Complex symmetric systems

ABSTRACT

Preconditioned Conjugate Gradient (PCG) is a popular method for solving large and sparse linear systems of equations. The performance of PCG at scale is affected due to the costly global synchronization steps that arise in dot-products on distributed memory systems. Pipelined PCG (PIPECG) removes the costly global synchronization steps from PCG by only executing a single non-blocking allreduce per iteration and overlapping it with independent computations.

In our previous work, we have developed a novel pipelined PCG algorithm called PIPECG-OATI (One Allreduce per Two Iterations) for real linear systems which executes a single non-blocking allreduce per two iterations and provides a large overlap of global communication with independent computations at higher number of cores. Our method achieves this overlap by using iteration combination and by introducing new recurrence and non-recurrence computations. We implement optimizations in the PIPECG-OATI method to use cache memory efficiently.

In this work, we present PIPECG-OATI-c method for linear systems with complex Hermitian positive definite and complex symmetric matrices. We compare our method with various pipelined CG methods on a variety of problems and demonstrate that our method always gives the least run times. We performed experiments with our method using 20M and 30M unknowns on up to 16K cores and obtained up to 2.48X performance improvement over PCG and 2.14X performance improvement over PIPECG methods. We also experimented with up to 1-billion unknowns on 16K cores, the largest problem size explored for the CG problem, to our knowledge, and obtained about 25% improvement over PCG.

1. Introduction

High performance computing applications in Computational Fluid Dynamics, Computational Electromagnetics etc. solve partial differential equations (PDEs) over space and time. These PDEs are discretized using finite volume, finite element or finite difference methods and they result in a linear system of equations $Ax = b$. Generally, the coefficient matrix A obtained from these discretization schemes is large and sparse. Iterative methods are used to solve these linear systems of equations with large and sparse matrices. In iterative methods, we begin with an initial solution x_0 and reduce the error in solution until it reaches the user defined error tolerance. The most widely used iterative methods used for solving these sparse systems are Krylov Subspace methods. The basic idea behind Krylov methods when solving a linear system $Ax = b$ is to build a solution within the Krylov subspace composed of several powers of matrix A multiplied by vector b, that is, $\{b, Ab, A^2b, ..., A^mb\}$.

Conjugate Gradient (CG) [18] [26] method is one of the widely used Krylov Subspace methods and is used to find the solution of linear systems with large and sparse positive definite matrices. In exact arithmetic, CG gives the solution of a system of size N in N steps. A preconditioner can be applied to the system to condition the input system and to improve convergence.

CG method is used to solve linear systems with real coefficient matrices which are symmetric ($A = A^T$) and positive definite. The coefficient matrix A is usually real, but complex matrices do occur, for example, in computational electromagnetics. CG method can also be used to solve linear systems with complex coefficient matrices. Two types of complex linear systems occur, linear systems with complex Hermitian positive definite matrices ($A = \overline{A^T}$) and linear systems with complex symmetric matrices ($A = A^T$). CG method has been successfully used for the solution of linear systems with large and sparse Hermitian positive definite matrices in [27][3]. The method, in its classical form, cannot be used to

* Corresponding author.
E-mail addresses: manasitiwari@iisc.ac.in (M. Tiwari), vss@iisc.ac.in (S. Vadhiyar).

https://doi.org/10.1016/j.jpdc.2022.01.008
0743-7315/© 2022 Elsevier Inc. All rights reserved.

solve linear systems with complex symmetric matrices as shown in [20][15]. However, some linear systems with complex symmetric matrices can be solved using the CG method by changing the dot product formulation for the complex vectors [4–6].

The main computational kernels in Preconditioned Conjugate Gradient (PCG) are Sparse Matrix Vector Product (SPMV), Preconditioner Application (PC), Vector-Multiply-Adds (VMAs) and Dot Products. The dot products use allreduce operations in distributed memory systems. The scalability bottleneck in PCG is the synchronization that happens across all cores due to the allreduce operations in the algorithm. Hence, existing works by Saad [25], Meurant [21], Azevedo et al. [12] and Chronopoulos et al. [8] had worked on reducing the number of allreduces to one per iteration as opposed to the three that exists in the original PCG.

The overlap of allreduce with useful work has been made possible by the advent of non-blocking collectives like MPI_IAllreduce in the MPI-3 standard [1]. Gropp's Asynchronous PCG [17] was the first work to use non-blocking collectives to overlap allreduces with computation. It used the original PCG consisting of three allreduces per iteration and introduced inexpensive Vector-Multiply-Add (VMA) operations to eliminate the dependencies that prevent the overlap. The resulting algorithm is then able to overlap one allreduce with SPMV and the other two with PC. Pipelined PCG (PIPECG) proposed by Ghysels et al. [16], on which this work is based, uses Chronopoulos-Gear PCG [8] which has one allreduce per iteration. By introducing extra VMAs, it overlaps this allreduce with a PC and an SPMV. Pipelined PCG with deeper pipelines (PIPELCG) was introduced in [11][10] in order to overlap more work with allreduce at higher core counts. It starts with Generalized Minimal Residual (GMRES) as the base algorithm. Another variant of CG which has only one allreduce per iteration is the three-term recurrence variant. However, this has been shown to have lower accuracy than the original three two-term recurrence PCG variants. A pipelined version of this variant (PIPECG3) was proposed by Eller et al. [14]. It overlaps the single allreduce with two PCs and two SPMVs.

This work is an extension of our previous work [29] in which we proposed a novel pipelined PCG method for distributed memory systems, called PIPECG-OATI (PIPECG-One Allreduce per Two Iterations) obtained by combining iterations and by introducing new recurrence and non-recurrence computations. In PIPECG-OATI, we reduce the number of allreduces to one per two iterations and overlap this allreduce with two PCs and two SPMVs.

The PIPECG-OATI method presented in our previous work [29] was derived and tested for real linear systems with symmetric positive definite matrices. In this work, we extend the PIPECG-OATI method to support linear systems with complex Hermitian positive definite and complex symmetric matrices and call it the PIPECG-OATI-c ('c' for complex systems) method. For providing support for complex Hermitian positive definite systems, extra dot products are introduced in PIPECG-OATI so that the method is arithmetically equivalent to the PCG and PIPECG methods. For providing support for complex symmetric systems, the formulation of complex dot products is changed.

We compared the performance of the PIPECG-OATI-c method with other PCG variants. We performed experiments with our method using 20M and 30M unknowns on up to 16K cores and obtained up to 2.48X performance improvement over PCG and 2.14X performance improvement over PIPECG methods. We also experimented with up to 1-billion unknowns on 16K cores, the largest problem size explored for the CG problem, to our knowledge, and obtained about 25% improvement over PCG. We also performed weak scaling results with 1250 rows per core on up to 16K cores and our method performed at least 15% better than the other methods at all the core counts. Our method also reaches the accuracy values used in practical problems the fastest when compared to the other methods.

Algorithm 1 Preconditioned Conjugate Gradient (PCG).

```
1:  $r_0 = b - Ax_0$; $u_0 = M^{-1}r_0$;
2:  $\gamma_0 = (u_0, r_0)$; $norm_0 = \sqrt{(u_0, u_0)}$
3:  for i=0,1... do
4:      if $i > 0$ then
5:          $\beta_i = \gamma_i / \gamma_{i-1}$
6:      else
7:          $\beta_i = 0$
8:      end if
9:      $p_i = u_i + \beta_i p_{i-1}$
10:     $s = Ap_i$
11:     $\delta = (s, p_i)$
12:     MPI_Allreduce on $\delta$
13:     $\alpha = \gamma_i / \delta$
14:     $x_{i+1} = x_i + \alpha p_i$
15:     $r_{i+1} = r_i - \alpha s$
16:     $u_{i+1} = M^{-1} r_{i+1}$
17:     $\gamma_{i+1} = (u_{i+1}, r_{i+1})$;
18:     MPI_Allreduce on $\gamma_{i+1}$
19:     $norm_{i+1} = \sqrt{(u_{i+1}, u_{i+1})}$
20:     MPI_Allreduce on $norm_{i+1}$
21: end for
```

The rest of the paper is organized as follows: Section 2 gives background related to PCG and PIPECG, Section 3 describes our algorithms for PIPECG-OATI and PIPECG-OATI-c methods. Section 4 presents the analysis and comparison of our method compared to other works. Section 5 presents the optimizations we have implemented in our method. Section 6 presents experiments, results and discussions for our proposed methods and Section 7 gives the conclusions and future work.

2. Background

2.1. PCG method

The Preconditioned Conjugate Gradient Method (PCG) introduced by Hestenes and Stiefel [18] is given in Algorithm 1. It iteratively solves the preconditioned system $M^{-1}Ax = M^{-1}b$ where both A and M are symmetric and positive definite square matrices of size N × N. As shown in Algorithm 1, the computational kernels in PCG's for loop are Sparse Matrix Vector Product (SPMV) in line 10, Preconditioner Application (PC) in line 16, Vector-Multiply-Adds (VMAs) in lines 9, 14 and 15 and dot products in lines 11, 17 and 19. Note that the third dot product (line 19) is for calculating the residual norm which is used to check for convergence.

2.2. Impact of ordering, partitioning and preconditioning strategies

The performance of the SPMV kernel has been shown to be dependent on ordering of the rows in the matrix [23][30]. The A matrix derived from a PDE can, possibly after a permutation of rows, be mapped to the underlying machine architecture in such a way that the SPMV only requires communication between neighboring nodes, i.e. nodes that are separated by a small number of hops. Such permutation of rows can also significantly reduce the communication volume between these neighboring nodes. In this work, we don't explicitly change the permutation of the rows of the A matrix because we work with well structured matrices which only require communication with the neighboring nodes and also have low communication volumes. Moreover, our work is intended for generic network architectures. Further, our methods can be used together with any elaborate reordering procedures applied on the matrix.

The performance of the SPMV kernel also depends on the partitioning strategies used as shown in [7][24]. In this work, we use the PETSc library [4] to handle all the underlying communication

Algorithm 2 Pipelined Preconditioned Conjugate Gradient (PIPECG).

```
 1: $r_0 = b - Ax_0$; $u_0 = M^{-1}r_0$; $w_0 = Au_0$;
 2: $\gamma_0 = (r_0, u_0)$; $\delta = (w_0, u_0)$; $norm_0 = \sqrt{(u_0, u_0)}$
 3: $m_0 = M^{-1}w_0$; $n_0 = Am_0$
 4: for i=0,1... do
 5:     if $i > 0$ then
 6:         $\beta_i = \gamma_i/\gamma_{i-1}$; $\alpha_i = \gamma_i/(\delta - \beta_i\gamma_i/alpha_{i-1})$;
 7:     else
 8:         $\beta_i = 0$; $\alpha_i = \gamma_i/\delta$
 9:     end if
10:     $z_i = n_i + \beta_i z_{i-1}$
11:     $q_i = m_i + \beta_i q_{i-1}$
12:     $s_i = w_i + \beta_i s_{i-1}$
13:     $p_i = u_i + \beta_i p_{i-1}$
14:     $x_{i+1} = x_i + \alpha_i p_i$
15:     $r_{i+1} = r_i - \alpha_i s_i$
16:     $u_{i+1} = u_i - \alpha_i q_i$
17:     $w_{i+1} = w_i - \alpha_i z_i$
18:     $\gamma_{i+1} = (r_{i+1}, u_{i+1})$
19:     $\delta = (w_{i+1}, u_{i+1})$
20:     $norm_{i+1} = \sqrt{(u_{i+1}, u_{i+1})}$
21:     MPI_Iallreduce on $\gamma_{i+1}, \delta, norm_{i+1}$
22:     $m_{i+1} = M^{-1}w_{i+1}$
23:     $n_{i+1} = Am_{i+1}$
24: end for
```

in the SPMV kernel. PETSc, by default, employs a 1-dimensional block-row distribution of the matrices and 1-dimensional distribution of the vectors as well. There is one vector that needs to be replicated at every process for the SPMV kernel and the communication for that replication is handled by PETSc. There are different matrix partitioning techniques supported in PETSc. In this work, we use the default 1-d distribution of the matrices as our aim is not to reduce communication within the SPMV kernel itself but to overlap the global communication with computations in the iterative solver as a whole. Our methods can be used in conjunction with any partitioning strategies.

The performance of the CG method depends on the preconditioner (PC) used as shown in [28][19][9]. Different PCs would converge in different number of iterations with the CG method. The volume and pattern of communications depend on the type of PC that is employed. We use PETSc's Jacobi PC for all our tests since our objective is to minimize global communication patterns. Our methods can be used together with any preconditioners that can be applied to the CG method.

2.3. Overlapping non-blocking allreduce with computations

VMAs require no communication. Dot products use allreduce which requires all the processors to synchronize and send their local dot products so that the global dot product can be calculated. In the original PCG Algorithm 1, the allreduce used in the dot products cannot be overlapped with any work because the results of the dot products are needed immediately in the next step. So the cores remain idle till the communication for calculating global dot product completes. Also, there are three allreduces per iteration in PCG, so the cores have to incur synchronization and idling cost thrice. As the number of cores increase, the time taken for allreduce increases, thus the cores remain idle for a longer time and this becomes the bottleneck and hinders obtaining good performance for PCG at higher number of cores.

2.4. PIPECG method

The Pipelined Preconditioned Conjugate Gradient Method (PIPECG) was proposed by Ghysels and Vanroose [16] for obtaining performance improvements on distributed memory architectures. As shown in Algorithm 2, PIPECG introduces extra AXPY operations (lines 10, 11, 12, 16, 17) to remove the dependencies between the dot products and PC and SPMV so that PC and SPMV can be computed while the communication for dot products is being completed, thereby increasing the utilization of the cores. The MPI_Iallreduce (line 21) for the dot products (line 18, 19, 20) can be overlapped with the PC and SPMV (line 22, 23).

Algorithm 3 PIPECG Method: two iterations unrolled.

```
 1: $r_0 = b - Ax_0$; $u_0 = M^{-1}r_0$; $w_0 = Au_0$;
 2: $\gamma_0 = (r_0, u_0)$; $\delta_0 = (w_0, u_0)$; $norm_0 = \sqrt{(u_0, u_0)}$
 3: $m_0 = M^{-1}w_0$; $n_0 = Am_0$
 4: for i=0,2,4,... do
 5:     if $i > 0$ then
 6:         $\beta_i = \gamma_i/\gamma_{i-1}$; $\alpha_i = \gamma_i/(\delta_i - \beta_i\gamma_i/\alpha_{i-1})$;
 7:     else
 8:         $\beta_i = 0$; $\alpha_i = \gamma_i/\delta_i$
 9:     end if
10:     $z_i = n_i + \beta_i z_{i-1}$; $q_i = m_i + \beta_i q_{i-1}$; $s_i = w_i + \beta_i s_{i-1}$
11:     $p_i = u_i + \beta_i p_{i-1}$; $x_{i+1} = x_i + \alpha_i p_i$; $r_{i+1} = r_i - \alpha_i s_i$
12:     $u_{i+1} = u_i - \alpha_i q_i$; $w_{i+1} = w_i - \alpha_i z_i$
13:     $\gamma_{i+1} = (r_{i+1}, u_{i+1})$; $\delta_{i+1} = (w_{i+1}, u_{i+1})$; $norm_{i+1} = \sqrt{(u_{i+1}, u_{i+1})}$
14:     MPI_Iallreduce on $\gamma_{i+1}, \delta_{i+1}, norm_{i+1}$
15:     $m_{i+1} = M^{-1}w_{i+1}$; $n_{i+1} = Am_{i+1}$
16:     $\beta_{i+1} = \gamma_{i+1}/\gamma_i$; $\alpha_{i+1} = \gamma_{i+1}/(\delta_{i+1} - \beta_{i+1}\gamma_{i+1}/\alpha_i)$;
17:     $z_{i+1} = n_{i+1} + \beta_{i+1} z_i$; $q_{i+1} = m_{i+1} + \beta_{i+1} q_i$
18:     $s_{i+1} = w_{i+1} + \beta_{i+1} s_i$; $p_{i+1} = u_{i+1} + \beta_{i+1} p_i$
19:     $x_{i+2} = x_{i+1} + \alpha_{i+1} p_{i+1}$; $r_{i+2} = r_{i+1} - \alpha_{i+1} s_{i+1}$
20:     $u_{i+2} = u_{i+1} - \alpha_{i+1} q_{i+1}$; $w_{i+2} = w_{i+1} - \alpha_{i+1} z_{i+1}$
21:     $\gamma_{i+2} = (r_{i+2}, u_{i+2})$; $\delta_{i+2} = (w_{i+2}, u_{i+2})$; $norm_{i+2} = \sqrt{(u_{i+2}, u_{i+2})}$
22:     MPI_Iallreduce on $\gamma_{i+2}, \delta_{i+2}, norm_{i+2}$
23:     $m_{i+2} = M^{-1}w_{i+2}$; $n_{i+2} = Am_{i+2}$
24: end for
```

The PIPECG method overlaps a single MPI_Iallreduce with one PC and one SPMV. While this is a reasonable strategy for lower number of cores, when we run the PIPECG code at higher number of cores, the time taken by the MPI_Iallreduce can not be fully overlapped by the PC and SPMV. In order to hide the latency introduced by MPI_Iallreduce at higher number of cores, we must overlap it with more work.

3. Methodology

We present a brief derivation of PIPECG-OATI method in Section 3.1 and a derivation of PIPECG-OATI-c method in sections 3.2 and 3.3.

3.1. PIPECG-OATI method for real linear systems

We propose a novel algorithm in [29], PIPECG-OATI, which combines two iterations of PIPECG, reduces the number of MPI_Iallreduce to one per two iterations and then overlaps it with two PCs and two SPMVs. This is done at the cost of introducing extra VMA operations. The primary challenge in combining two iterations of PIPECG and pipelining it is that it has dependencies that require an extra PC and an extra SPMV for each combined-iteration. So, a total of three PCs and three SPMVs would be required in a combined-iteration as opposed to two PCs and two SPMVs in two uncombined iterations. Since the PC and SPMV are the most computationally intensive kernels in each iteration, an extra PC and SPMV would degrade the performance of PIPECG-OATI. To deal with this challenge, we introduced new non-recurrence computations in each iteration of PIPECG-OATI which brings down the number of PCs and SPMVs to two per combined-iteration. To find a detailed derivation of the PIPECG-OATI method for real linear systems, please refer to our previous work [29].

In Algorithm 3, we unroll two iterations of PIPECG. Following are steps for deriving PIPECG-OATI from Algorithm 3.

1. Move the PC, m_{i+1} and SPMV, n_{i+1} (line 15) after the PC, m_{i+2} and SPMV, n_{i+2} (line 23) by introducing recurrence relations for them.

Algorithm 4 PIPECG-OATI (PIPECG-One Allreduce per Two Iterations): After Step 4.

```
1: $r_0 = b - Ax_0$; $u_0 = M^{-1}r_0$; $w_0 = Au_0$;
2: $\gamma_0 = \lambda_7 = (r_0, u_0)$; $\delta_0 = \lambda_8 = (w_0, u_0)$; $norm_0 = real(\sqrt{\lambda_9}) = real(\sqrt{(u_0, u_0)})$
3: $m_0 = M^{-1}w_0$; $n_0 = Am_0$; $g_0 = M^{-1}n_0$; $h_0 = Ag_0$; $e_0 = M^{-1}h_0$; $f_0 = Ae_0$
4: $\lambda_1 = (w_0, m_0)$; $\lambda_4 = (n_0, m_0)$
5: for i=0,2,4,... do
6:     if $i > 0$ then
7:         $\beta_i = \gamma_i/\gamma_{i-1}$; $\alpha_i = \gamma_i/(\delta_i - \beta_i\gamma_i/\alpha_{i-1})$;
8:     else
9:         $\beta_i = 0; \alpha_i = \gamma_i/\delta_i$
10:    end if
11:    $\gamma_{i+1} = \lambda_7 - 2\alpha_i(\lambda_8 + \beta_i\lambda_0) + \alpha_i{}^2(\lambda_1 + 2 * \beta_i\lambda_2 + \beta_i{}^2\lambda_3)$
12:    $\delta_{i+1} = \lambda_8 - \alpha_i(\lambda_1 + \beta_i\lambda_2) - \alpha_i(\lambda_1 + \beta_i\lambda_2) + \alpha_i{}^2(\lambda_4 + \beta_i\lambda_5 + \beta_i\lambda_5 + \beta_i{}^2\lambda_6)$
13:    $\beta_{i+1} = \gamma_{i+1}/\gamma_i$; $\alpha_{i+1} = \gamma_{i+1}/(\delta_{i+1} - \beta_{i+1}\gamma_{i+1}/\alpha_i)$;
14:    VecOps
15:    MPI_Iallreduce on $\lambda_0, \lambda_1 ... \lambda_9$
16:    $g_{i+2} = M^{-1}n_{i+2}$; $h_{i+2} = Ag_{i+2}$
17:    $e_{i+2} = M^{-1}h_{i+2}$; $f_{i+2} = Ae_{i+2}$
18:    $a_{i+1} = (g_{i+1} - g_{i+2})/\alpha_{i+1}$
19:    $b_{i+1} = (h_{i+1} - h_{i+2})/\alpha_{i+1}$
20:    $\gamma_{i+2} = \lambda_7$; $\delta_{i+2} = \lambda_8$; $norm_{i+2} = \sqrt{\lambda_9}$
21: end for
```

Algorithm 5 VecOps.

```
1: $z_i = n_i + \beta_i z_{i-1}$; $q_i = m_i + \beta_i q_{i-1}$; $s_i = w_i + \beta_i s_{i-1}$
2: $p_i = u_i + \beta_i p_{i-1}$; $c_i = g_i + \beta_i c_{i-1}$; $d_i = h_i + \beta_i d_{i-1}$;
3: $a_i = e_i + \beta_i a_{i-1}$; $b_i = f_i + \beta_i b_{i-1}$;
4: $x_{i+1} = x_i + \alpha_i p_i$; $r_{i+1} = r_i - \alpha_i s_i$
5: $u_{i+1} = u_i - \alpha_i q_i$; $w_{i+1} = w_i - \alpha_i z_i$
6: $m_{i+1} = m_i - \alpha_i c_i$; $n_{i+1} = n_i - \alpha_i d_i$
7: $g_{i+1} = g_i - \alpha_i a_i$; $h_{i+1} = h_i - \alpha_i b_i$
8: $z_{i+1} = n_{i+1} + \beta_{i+1} z_i$; $q_{i+1} = m_{i+1} + \beta_{i+1} q_i$;
9: $s_{i+1} = w_{i+1} + \beta_{i+1} s_i$ $p_{i+1} = u_{i+1} + \beta_{i+1} p_i$;
10: $c_{i+1} = g_{i+1} + \beta_{i+1} c_i$; $d_{i+1} = h_{i+1} + \beta_{i+1} d_i$;
11: $x_{i+2} = x_{i+1} + \alpha_{i+1} p_{i+1}$; $r_{i+2} = r_{i+1} - \alpha_{i+1} s_{i+1}$
12: $u_{i+2} = u_{i+1} - \alpha_{i+1} q_{i+1}$; $w_{i+2} = w_{i+1} - \alpha_{i+1} z_{i+1}$
13: $m_{i+2} = m_{i+1} - \alpha_{i+1} c_{i+1}$; $n_{i+2} = n_{i+1} - \alpha_{i+1} d_{i+1}$
14: $\lambda_0 = (u_{i+2}, s_{i+1})$; $\lambda_1 = (w_{i+2}, m_{i+2})$;
15: $\lambda_2 = (w_{i+2}, q_{i+1})$; $\lambda_3 = (s_{i+1}, q_{i+1})$;
16: $\lambda_4 = (n_{i+2}, m_{i+2})$; $\lambda_5 = (n_{i+2}, q_{i+1})$;
17: $\lambda_6 = (z_{i+1}, q_{i+1})$; $\lambda_7 = (r_{i+2}, u_{i+2})$;
18: $\lambda_8 = (u_{i+2}, w_{i+2})$; $\lambda_9 = (u_{i+2}, u_{i+2})$;
```

2. Express the dot products γ_{i+1}, δ_{i+1} and $norm_{i+1}$ (line 13) as recurrence relations and move the resulting new dot products after the previous iteration's dot products γ_{i+2}, δ_{i+2} and $norm_{i+2}$ (line 21).
3. As the new dot products will need results of PC, m_{i+2} and SPMV, n_{i+2} beforehand, introduce recurrence relations for these PC and SPMV.
4. To deal with extra PC, a_{i+1} and SPMV, b_{i+1}, introduce new non-recurrence computations.

Algorithm 4 captures all the above mentioned reorganizations, new recurrence relations and new non-recurrence computations. Putting it all together, PIPECG-OATI overlaps one MPI_Iallreduce (line 15) with two PCs and two SPMVs (lines 16 and 17) at the cost of introducing 21 new Vector Operations and 7 new dot products (shown in Algorithm 5) which can be computed with a single allreduce.

3.2. PIPECG-OATI-c method for complex Hermitian systems

The PIPECG-OATI-c method for complex Hermitian positive definite systems can be derived by following the same steps as listed in Section 3.1. However, step 2 cannot be exactly followed for complex Hermitian systems. For PIPECG-OATI-c to be computationally equivalent to PCG and other pipelined CG variants, we have to add extra dot products.

In a complex Hermitian system, the coefficient matrix A, the preconditioner M^{-1} and the vectors have complex entries. For complex vectors a and b, the dot product is defined as:

$$a \cdot b = (a, \overline{b}), \text{ where } \overline{b} \text{ is the complex conjugate of } b.$$

In order to compute γ_{i+1} in Section 3.1 of our previous work [29], we substitute the full equations of r_{i+1} and u_{i+1}:

$$\begin{aligned}\gamma_{i+1} &= r_{i+1} \cdot u_{i+1} = (r_i - \alpha_i s_i) \cdot (u_i - \alpha_i q_i) \\ &= r_i \cdot u_i - \alpha_i(r_i \cdot q_i) - \alpha_i(s_i \cdot u_i) + \alpha_i^2(s_i \cdot q_i).\end{aligned}$$

Further, we used the equality $q = M^{-1}s$ for the following transformation.

$$r_i \cdot q_i = r_i \cdot M^{-1}s_i = M^{-1}r_i \cdot s_i = u_i \cdot s_i = s_i \cdot u_i.$$

Thus we obtained the following equation involving less number of dot products.

$$\gamma_{i+1} = r_i \cdot u_i - 2\alpha_i(s_i \cdot u_i) + \alpha_i{}^2(s_i \cdot q_i)$$

For real systems, $u_i \cdot s_i = s_i \cdot u_i$. But for complex Hermitian systems, this relation does not hold true because complex dot products are non-commutative.

$$u_i \cdot s_i = (u_i, \overline{s_i}).$$

$$s_i \cdot u_i = (s_i, \overline{u_i}).$$

We used the transformation for the real case because it reduced the number of dot products that we had to calculate. However, for complex Hermitian systems, even after applying the transformation we will have to calculate two dot products, $s_i \cdot u_i$ and $u_i \cdot s_i$. For complex Hermitian systems, γ_{i+1} will be

$$\gamma_{i+1} = r_i \cdot u_i - \alpha_i(u_i \cdot s_i) - \alpha_i(s_i \cdot u_i) + \alpha_i^2(s_i \cdot q_i).$$

We further simplify $u_i \cdot s_i$ as

$$u_i \cdot s_i = u_i \cdot (w_i + \beta_i s_{i-1}) = u_i \cdot w_i + \beta_i(u_i \cdot s_{i-1})$$

We further simplify $s_i \cdot u_i$ as

$$s_i \cdot u_i = (w_i + \beta_i s_{i-1}) \cdot u_i = w_i \cdot u_i + \beta_i(s_{i-1} \cdot u_i)$$

We further simplify $s_i \cdot q_i$ as

$$\begin{aligned}s_i \cdot q_i &= (w_i + \beta_i s_{i-1}) \cdot (m_i + \beta_i q_{i-1}) \\ &= w_i \cdot m_i + \beta_i(w_i \cdot q_{i-1}) + \beta_i(s_{i-1} \cdot m_i) + \beta_i^2(s_{i-1} \cdot q_{i-1}).\end{aligned}$$

Substituting values of $u_i \cdot s_i$, $s_i \cdot u_i$ and $s_i \cdot q_i$ in γ_{i+1}, we obtain

$$\begin{aligned}\gamma_{i+1} = r_i \cdot u_i &- \alpha_i(u_i \cdot w_i + \beta_i(u_i \cdot s_{i-1})) \\ &- \alpha_i(w_i \cdot u_i + \beta_i(s_{i-1} \cdot u_i)) + \alpha_i^2(w_i \cdot m_i + \beta_i(w_i \cdot q_{i-1}) \\ &+ \beta_i(s_{i-1} \cdot m_i) + \beta_i^2(s_{i-1} \cdot q_{i-1}))\end{aligned}$$

Now, expressing δ_{i+1} by substituting the full equations of w_{i+1} and u_{i+1},

$$\begin{aligned}\delta_{i+1} &= w_{i+1} \cdot u_{i+1} = (w_i - \alpha_i z_i) \cdot (u_i - \alpha_i q_i) \\ &= w_i \cdot u_i - \alpha_i(w_i \cdot q_i) - \alpha_i(z_i \cdot u_i) + \alpha_i^2(z_i \cdot q_i).\end{aligned}$$

We further simplify $w_i \cdot q_i$ as

$$w_i \cdot q_i = w_i \cdot (m_i + \beta_i q_{i-1}) = w_i \cdot m_i + \beta_i(w_i \cdot q_{i-1})$$

We further simplify $z_i \cdot u_i$ as

$$z_i \cdot u_i = (n_i + \beta_i z_{i-1}) \cdot u_i = n_i \cdot u_i + \beta_i(z_{i-1} \cdot u_i)$$

Algorithm 6 PIPECG-OATI-c Method.

```
1: $r_0 = b - Ax_0;\ u_0 = M^{-1}r_0;\ w_0 = Au_0;$
2: $\gamma_0 = \lambda_{10} = (r_0, \overline{u_0});\ \delta_0 = \lambda_{11} = (w_0, \overline{u_0});\ norm_0 = real(\sqrt{\lambda_{12}}) = real(\sqrt{(u_0, \overline{u_0})})$
3: $m_0 = M^{-1}w_0;\ n_0 = Am_0;\ g_0 = M^{-1}n_0;\ h_0 = Ag_0;\ e_0 = M^{-1}h_0;\ f_0 = Ae_0$
4: $\lambda_1 = (w_0, \overline{m_0}); \lambda_6 = (n_0, \overline{m_0}); \lambda_{13} = (u_0, \overline{w_0})$
5: for i=0,2,4,... do
6:     if $i > 0$ then
7:         $\beta_i = \gamma_i/\gamma_{i-1};\ \alpha_i = \gamma_i/(\delta_i - \beta_i\gamma_i/\alpha_{i-1});$
8:     else
9:         $\beta_i = 0; \alpha_i = \gamma_i/\delta_i$
10:    end if
11:    $\gamma_{i+1} = \lambda_{10} - \alpha_i(\lambda_{13} + \beta_i\lambda_{14}) - \alpha_i(\lambda_{11} + \beta_i\lambda_0) + \alpha_i{}^2(\lambda_1 + \beta_i\lambda_2 + \beta_i\lambda_3 + \beta_i{}^2\lambda_4)$
12:    $\delta_{i+1} = \lambda_{11} - \alpha_i(\lambda_1 + \beta_i\lambda_2) - \alpha_i(\lambda_5 + \beta_i\lambda_3) + \alpha_i{}^2(\lambda_6 + \beta_i\lambda_7 + \beta_i\lambda_8 + \beta_i{}^2\lambda_9)$
13:    $\beta_{i+1} = \gamma_{i+1}/\gamma_i;\ \alpha_{i+1} = \gamma_{i+1}/(\delta_{i+1} - \beta_{i+1}\gamma_{i+1}/\alpha_i);$
14:    VecOps
15:    MPI_Iallreduce on $\lambda_0, \lambda_1 ... \lambda_{14}$
16:    $g_{i+2} = M^{-1}n_{i+2};\ h_{i+2} = Ag_{i+2}$
17:    $e_{i+2} = M^{-1}h_{i+2};\ f_{i+2} = Ae_{i+2}$
18:    $a_{i+1} = (g_{i+1} - g_{i+2})/\alpha_{i+1}$
19:    $b_{i+1} = (h_{i+1} - h_{i+2})/\alpha_{i+1}$
20:    $\gamma_{i+2} = \lambda_{10};\ \delta_{i+2} = \lambda_{11};\ norm_{i+2} = real(\sqrt{\lambda_{12}})$
21: end for
```

Algorithm 7 VecOps.

```
1: $z_i = n_i + \beta_i z_{i-1};\ q_i = m_i + \beta_i q_{i-1};\ s_i = w_i + \beta_i s_{i-1}$
2: $p_i = u_i + \beta_i p_{i-1};\ c_i = g_i + \beta_i c_{i-1};\ d_i = h_i + \beta_i d_{i-1};$
3: $a_i = e_i + \beta_i a_{i-1};\ b_i = f_i + \beta_i b_{i-1};$
4: $x_{i+1} = x_i + \alpha_i p_i;\ r_{i+1} = r_i - \alpha_i s_i$
5: $u_{i+1} = u_i - \alpha_i q_i;\ w_{i+1} = w_i - \alpha_i z_i$
6: $m_{i+1} = m_i - \alpha_i c_i;\ n_{i+1} = n_i - \alpha_i d_i$
7: $g_{i+1} = g_i - \alpha_i a_i;\ h_{i+1} = h_i - \alpha_i b_i$
8: $z_{i+1} = n_{i+1} + \beta_{i+1} z_i;\ q_{i+1} = m_{i+1} + \beta_{i+1} q_i;$
9: $s_{i+1} = w_{i+1} + \beta_{i+1} s_i\ \ p_{i+1} = u_{i+1} + \beta_{i+1} p_i;$
10: $c_{i+1} = g_{i+1} + \beta_{i+1} c_i;\ d_{i+1} = h_{i+1} + \beta_{i+1} d_i;$
11: $x_{i+2} = x_{i+1} + \alpha_{i+1} p_{i+1};\ r_{i+2} = r_{i+1} - \alpha_{i+1} s_{i+1}$
12: $u_{i+2} = u_{i+1} - \alpha_{i+1} q_{i+1};\ w_{i+2} = w_{i+1} - \alpha_{i+1} z_{i+1}$
13: $m_{i+2} = m_{i+1} - \alpha_{i+1} c_{i+1};\ n_{i+2} = n_{i+1} - \alpha_{i+1} d_{i+1}$
14: $\lambda_0 = (s_{i+1}, \overline{u_{i+2}}); \lambda_1 = (w_{i+2}, \overline{m_{i+2}}); \lambda_2 = (w_{i+2}, \overline{q_{i+1}});$
15: $\lambda_3 = (s_{i+1}, \overline{m_{i+2}}); \lambda_4 = (s_{i+1}, \overline{q_{i+1}}); \lambda_5 = (n_{i+2}, \overline{u_{i+2}});$
16: $\lambda_6 = (n_{i+2}, \overline{m_{i+2}}); \lambda_7 = (n_{i+2}, \overline{q_{i+1}}); \lambda_8 = (z_{i+1}, \overline{m_{i+1}});$
17: $\lambda_9 = (z_{i+2}, \overline{q_{i+2}}); \lambda_{10} = (r_{i+2}, \overline{u_{i+2}}); \lambda_{11} = (w_{i+2}, \overline{u_{i+2}});$
18: $\lambda_{12} = (u_{i+2}, \overline{u_{i+2}}); \lambda_{13} = (u_{i+2}, \overline{w_{i+2}}); \lambda_{14} = (u_{i+2}, \overline{s_{i+1}});$
```

We further simplify $z_i \cdot q_i$ as

$$\begin{aligned} z_i \cdot q_i &= (n_i + \beta_i z_{i-1}) \cdot (m_i + \beta_i q_{i-1}) \\ &= n_i \cdot m_i + \beta_i (n_i \cdot q_{i-1}) + \beta_i (z_{i-1} \cdot m_i) + \beta_i^2 (z_{i-1} \cdot q_{i-1}). \end{aligned}$$

Substituting values of $w_i \cdot q_i$, $z_i \cdot u_i$ and $z_i \cdot q_i$ in δ_{i+1}, we obtain

$$\begin{aligned} \delta_{i+1} = {} & w_i \cdot u_i - \alpha_i (w_i \cdot m_i + \beta_i (w_i \cdot q_{i-1})) \\ & - \alpha_i (n_i \cdot u_i + \beta_i (z_{i-1} \cdot u_i)) + \alpha_i^2 (n_i \cdot m_i + \beta_i (n_i \cdot q_{i-1}) \\ & + \beta_i (z_{i-1} \cdot m_i) + \beta_i^2 (z_{i-1} \cdot q_{i-1})) \end{aligned}$$

We can further transform $z_{i-1} \cdot u_i$ into $s_{i-1} \cdot m_i$ by using some equalities

$$\begin{aligned} z_{i-1} \cdot u_i &= Aq_{i-1} \cdot u_i = q_{i-1} \cdot Au_i = q_{i-1} \cdot w_i = M^{-1} s_{i-1} \cdot w_i \\ &= s_{i-1} \cdot M^{-1} w_i = s_{i-1} \cdot m_i. \end{aligned}$$

Because of the above transformation, we don't have to compute $z_{i-1} \cdot u_i$ separately because we are already computing $s_{i-1} \cdot m_i$ for γ_{i+1}. Combined with the dot products of the previous iteration, we need to calculate a total of fifteen dot products stored in an array λ, all of which can be computed using one allreduce. Compared to PIPECG-OATI method where we had to calculate ten dot products in each iteration, we need to calculate five more dot products for PIPECG-OATI-c method because of non-commutative nature of complex dot products.

The rest of the steps in Section 3.1 are followed for deriving PIPECG-OATI-c method. Algorithm 6 and Algorithm 7 show the PIPECG-OATI-c method with fifteen dot products. The changes made to the PIPECG-OATI method to derive PIPECG-OATI-c method are shown in red. PIPECG-OATI-c overlaps one MPI_Iallreduce (line 15) with two PCs and two SPMVs (lines 16 and 17) at the cost of introducing five new dot products to the PIPECG-OATI method.

Table 1
Differences between various PCG Methods for two iterations of execution. L=2 for PIPELCG.

Method	#Allr	Time for G, PC, SPMV	FLOP		Memory
			real/CS	CH	
PCG	6	6G+2PC+2SPMV	24	24	4
PIPECG	2	2(max(G, PC+SPMV))	44	44	9
PIPELCG	2	max(G, 2PC+2SPMV)	52	52	14
PIPECG3	1	max(G, 2PC+2SPMV)	90	-	25
PIPECG-OATI	1	max(G, 2PC+2SPMV)	80	90	19

3.3. PIPECG-OATI-c method for complex symmetric systems

In PETSc [4] library's PCG method implementation, support for complex symmetric matrices is provided by changing the way complex dot product is calculated. Applying the same convention of calculating dot products to our PIPECG-OATI-c method, we find that in fact we obtain the same algorithm as Algorithm 4 for real matrices. We change the way complex dot products are calculated as follows:

$$a \cdot b = (a, b).$$

In this case, the vector b is not conjugated. If we calculate the complex dot products in this way, we don't need to calculate extra dot products because we can use the commutative property of dot products to calculate the dot products once and reuse them again. Algorithm 4 derived for real linear systems also takes advantage of the commutative property of the real dot products and hence, for linear systems with complex symmetric matrices, we can use Algorithm 4 directly.

4. Overview of different CG variants

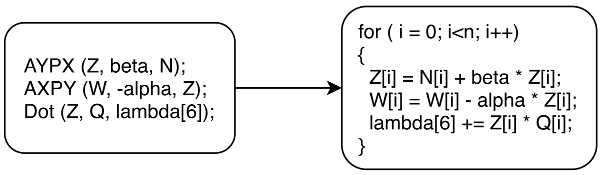
In this section, we provide an overview of state-of-the-art PCG variants as shown in Table 1. The #Allr column shows the number of allreduces in two iterations for every method. The Time column shows the time taken for two iterations for global allreduce (G), Preconditioner (PC) and Sparse Matrix Vector Product (SPMV). The FLOP column lists the number of Floating Point Operations (xN) in VMAs and dot products for two iterations. It is further subdivided into two columns: real/CS (complex symmetric) and CH (complex Hermitian). This is done because the number of dot products can be different for real, complex Hermitian and complex symmetric linear systems. The Memory column counts the number of vectors that need to be kept in the memory (excluding x and b).

The PIPECG-OATI (PIPECG-OATI-c for CH, CS cases) method has one allreduce per two iterations and overlaps it with two PCs and two SPMVs. It will give performance improvements over other methods when the time taken for global allreduce (G) is completely overlapped by the two PCs and SPMVs which will happen at higher number of cores. The PIPECG-OATI method also has different number of FLOPs for real, complex Hermitian and complex symmetric linear systems, as we have seen in section 3.

We also note that the FLOPs in PIPECG-OATI are significantly more than that in PCG, PIPECG and PIPELCG methods. We introduce optimizations in the next section to deal with this overhead.

5. Optimization

In this section, we discuss implementation optimizations that help in getting significant performance improvement by making efficient memory accesses.



```
AYPX (Z, beta, N);
AXPY (W, -alpha, Z);
Dot (Z, Q, lambda[6]);
```

```
for ( i = 0; i<n; i++)
{
  Z[i] = N[i] + beta * Z[i];
  W[i] = W[i] - alpha * Z[i];
  lambda[6] += Z[i] * Q[i];
}
```

Fig. 1. Merging VecOps.

As shown in Fig. 1 left, in the naïve implementation of PIPECG-OATI-c, we call individual AYPX, AXPY and DOT functions to implement the VecOps. However if we merge the VecOps as shown in Fig. 1 right, we can reuse the vectors already loaded in the cache and reduce the number of vector reads and writes to the main memory. Since the main memory accesses are expensive as compared to cache accesses, merging VecOps eliminates the need for accessing main memory repeatedly. As PIPECG-OATI-c introduces 21 extra VMAs and 12 new Dot Products as shown in Algorithm 5, using this technique helps in efficient memory accesses and hence reduces the overhead introduced by these extra operations.

In addition to this, we used the *pragma CRI ivdep* directive before the for loop of the merged VecOps. This helps the compiler to take advantage of the vector architecture of the underlying CPUs.

6. Experiments and results

6.1. Experiment setup

We ran tests on our Institute's supercomputer cluster called SahasraT, a Cray-XC40 machine which has 1376 compute nodes. Each node has two CPU sockets with 12 cores each, 128GB RAM and connected using Cray Aries interconnect. We have implemented our methods in the PETSc library [5]. We use cray-mpich version 7.7.2. For the non-blocking collective MPI_Iallreduce to make progress, it is necessary to configure our customized PETSc code with –LIBS=-ldmapps for dynamic linking to the DMAPP library and set MPICH_NEMESIS_ASYNC_PROGRESS to 1 in the job script. For the complex test cases to run correctly, it is necessary to configure the PETSc code with the option –with-scalar-type=complex. Our implementations for PIPECG-OATI and PIPECG-OATI-c are available in the open-source PETSc repository.[1]

Complex Symmetric Test Case: We use ex11 provided by PETSc in its tutorials folder. It solves the complex Helmholtz equation:

$$-\Delta u - \sigma_1 \times u + i \times \sigma_2 \times u = f \tag{1}$$

where Δ is the Laplace operator. Dirichlet boundary conditions are applied on all sides and the 2-D 5-point finite difference stencil is used. After the stencil is used to create the matrix A, the equation $Ax = b$ is solved. The solution vector x is initialized to a vector of zeroes. The complex Helmholtz equation has many applications in various fields of physics, such as optics, acoustics, electrostatics and quantum mechanics. We test with 30 million and 1 billion grid points for complex symmetric cases.

Complex Hermitian Test Case: We use ex45 provided by PETSc in its tutorials folder. It solves the Poisson equation:

$$-\Delta u = 1, \qquad 0 < x, y, z < 1 \tag{2}$$

with boundary conditions

$$u = 1, \quad for \quad x = 0, x = 1, y = 0, y = 1, z = 0, z = 1. \tag{3}$$

[1] Available as KSPPIPECG2. URL: https://www.mcs.anl.gov/petsc/petsc-current/docs/manualpages/KSP/KSPPIPECG2.html#KSPPIPECG2.

We modify the example to use 3-D 27-point finite difference stencil which is used in many benchmarks such as HPCG [13]. As we wanted complex Hermitian systems, we modified the code to initialize the same Poisson matrices with complex entries such that the resulting matrix is Hermitian positive definite. The equation $Ax = b$ is solved in the tests. The solution vector x_0 is initialized to a vector of zeroes. The Poisson equation is used to model electrostatics, steady-state diffusion, and other physical processes. We test with 20 million grid points and 1250 rows per core for complex Hermitian cases.

Our method PIPECG-OATI-c (unoptimized and optimized versions) are compared with the complex versions of other PCG variants i.e. PCG-c, PIPECG-c and PIPELCG-c (with L=2). The unoptimized version of PIPECG-OATI-c corresponds to the implementation of PIPECG-OATI-c which does not have merged VecOps as described in section 5. PIPECG3 does not support complex numbers, therefore, we haven't provided comparisons with PIPECG3 method here. We use Jacobi Preconditioner in all tests. Each test was run five times and the average speedup gained is presented. We present results for performance modeling, strong scaling, weak scaling and accuracy of our method compared with state-of-the-art methods.

6.2. Performance modeling results

We use a simple performance model to understand the expected performance of the PIPECG-OATI-c method at scale. The main computational kernels in the PIPECG-OATI-c method are SPMV, PC, VMAs and dot products. The dot product kernel is further divided into local dot product and the global allreduce. The VMAs and local dot product together form the VecOps. As given in Section 4, the VecOps are counted as floating point operations (FLOP) and from the same section, we formulate the expected time taken by the PIPECG-OATI-c method as:

$$t_{expected} = (max(G, 2PC + 2SPMV) + t_{flop}) \times no\_of\_iterations/2 \tag{4}$$

where t_{flop} is the time taken by the FLOPs. The number of iterations is halved since one iteration of PIPECG-OATI-c actually performs two iterations of the PCG method.

We model the performance of the PIPECG-OATI-c method for a complex symmetric system where the dimensions of the matrix A are 1 billion $\times$ 1 billion. We measure the times G, PC, SPMV and t_{flop} independently on various problem sizes and number of cores. We then substitute these component times in Equation (4) and obtain the $t_{expected}$ for PIPECG-OATI-c method.

We present the comparison between the expected and the measured times of PIPECG-OATI-c method in Fig. 2. Speedups for estimated and measured times for 100 iterations of PIPECG-OATI-c are presented with respect to measured time for 200 iterations of PCG. From Fig. 2, we observe that the PIPECG-OATI-c method is expected to perform better than PCG method as the number of cores increase and the work per cores decreases. The expected speedups for PIPECG-OATI-c are always more than the measured speedups. We believe that the reason for this is that our simple performance model ignores penalties introduced due to OS jitter, core speed variability or load imbalance that start to play an important role for larger systems. Nevertheless, our performance model adequately captures the trend of the actual speedups obtained with increasing number of cores.

6.3. Strong scaling results

Fig. 3 shows the strong scaling of different methods on a 5-pt 2D complex symmetric problem with 30 million (30M) unknowns

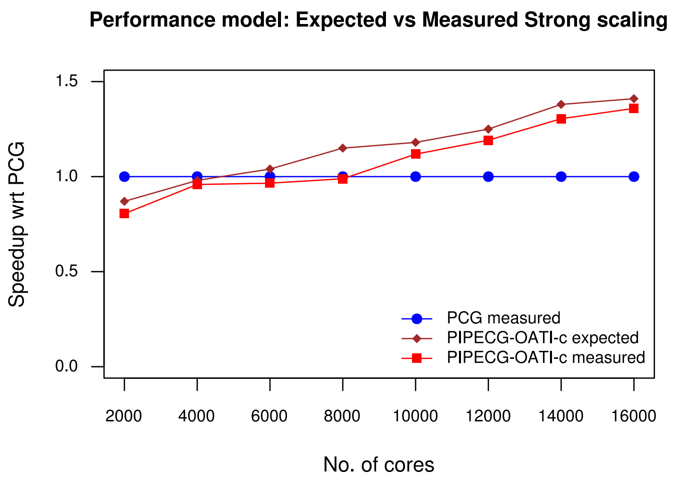


Fig. 2. Expected vs. Measured strong scaling of PIPECG-OATI-c on a 5-pt complex symmetric problem with 1B unknowns.

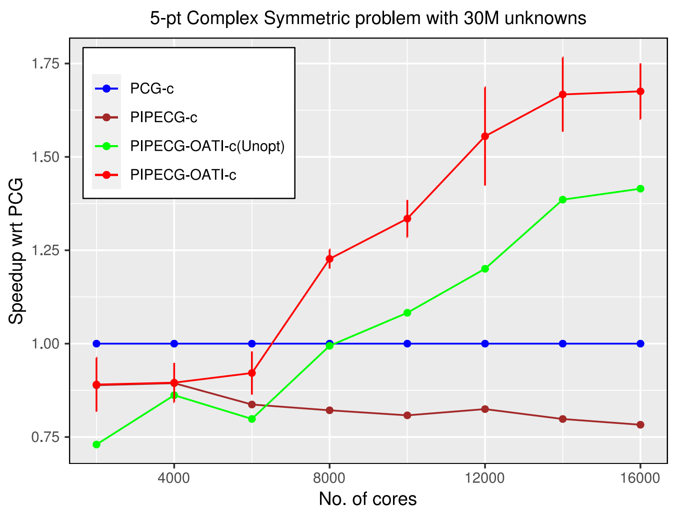


Fig. 3. Strong scaling of different methods on a 5-pt complex symmetric problem with 30M unknowns.

on up to 16000 cores. All the methods run to convergence for a relative tolerance of 10^{-5} in 8145 iterations. We plot the speedup obtained by each method with respect to PCG. The bars on the PIPECG-OATI-c method indicate the standard deviation in speedups for five runs and the point plotted is the average speedup.

We observe from Fig. 3 that the performance of PIPECG-c is always lower than PCG-c despite PIPECG-c overlapping one allreduce with computations. As 5-pt 2D problem provides less number of non zeroes in the coefficient matrix, one PC and one SPMV are not very expensive. Thus, allreduce is not efficiently overlapped with the computations. Also, the cost of unoptimized VecOps in PIPECG-c overshadows the allreduce it overlaps with the computations. So, PIPECG-c does not provide much overlap and in addition introduces unoptimized VecOps. This leads to low performance of PIPECG-c than the other methods.

Fig. 3 also shows that PIPECG-OATI-c unoptimized version performs better than PCG-c and PIPECG-c. This happens as it overlaps two PCs and two SPMVs with one allreduce. So, even though a single PC and SPMV in the 5-pt problem are not very expensive, two PCs and two SPMVs provide considerable work to be overlapped with the allreduce. Also, it starts performing better than PCG-c from 10000 cores because from 10000 cores on-wards, the allreduce cost becomes large enough to be overlapped by two PCs and

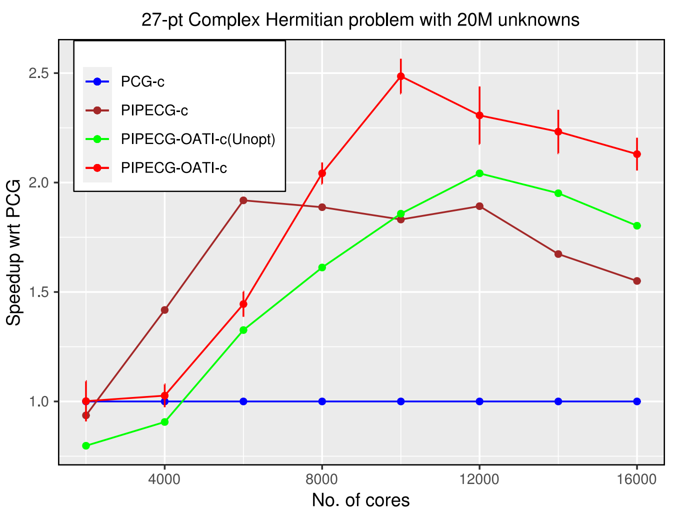


Fig. 4. Strong scaling of different methods on a 27-pt complex Hermitian problem with 20M unknowns.

two SPMVs. However, it still performs worse than PIPECG-OATI-c because the unoptimized VecOps provide additional overhead.

Finally, we observe that PIPECG-OATI-c optimized version performs the best when compared to all the other methods because it overlaps larger work with allreduce and uses cache memory effectively for the VecOps. So, for the 5-pt complex symmetric problem with 30M unknowns, PIPECG-OATI-c provides up to 1.67x speedup wrt PCG-c (at 16k cores), up to 2.14x speedup wrt PIPECG-c (at 16k cores) and up to 1.18x speedup wrt PIPECG-OATI-c unoptimized version (at 16k cores).

PIPELCG-c (L=2) method cannot compute the preconditioned norm of the residual whereas all the other methods determine the convergence based on the preconditioned residual norm. As PIPELCG-c determines convergence based on only natural norm of the residual, we ran the method to convergence for natural norm and observed that it gives different (more) number of iterations than the other methods. Due to this reason, we ran PIPECG-OATI-c to convergence for the natural norm of the residual so that it can be compared with PIPELCG-c method. We find that PIPECG-OATI-c gives up to 1.38x speedup over PIPELCG-c for this problem (at 16k cores).

Fig. 4 shows the strong scaling of different methods on a 27-pt 3D complex Hermitian problem with 20M unknowns on up to 16000 cores. All the methods ran to convergence for a relative tolerance of 10^{-5} in 5203 iterations.

We observe from Fig. 4 that PIPECG-c performs better than PCG-c because the PC and SPMV in the 27-pt 3D problem are expensive enough to completely overlap the cost of a single allreduce. The unoptimized VecOps do produce some overhead but it is overshadowed by the complete overlapping of the allreduce with PC and SPMV. PIPECG-OATI-c unoptimized performs better than PCG-c but worse than PIPECG-c till 8k cores. This happens because the allreduce cost is completely overlapped by one PC and one SPMV till 8k cores in PIPECG-c. The larger overlap provided by PIPECG-OATI-c unoptimized is not useful at lower number of cores and the additional overhead provided by PIPECG-OATI-c unoptimized due to extra VecOps becomes dominant. However, we observe that after 8k cores, it starts performing better than PIPECG-c method because the allreduce cost becomes adequate to be overlapped with two PCs and two SPMVs. PIPECG-OATI-c optimized performs better than all the methods. The curves start dropping from 10k cores on-wards because the work per core decreases and the allreduce cost becomes more than the work provided by the methods.

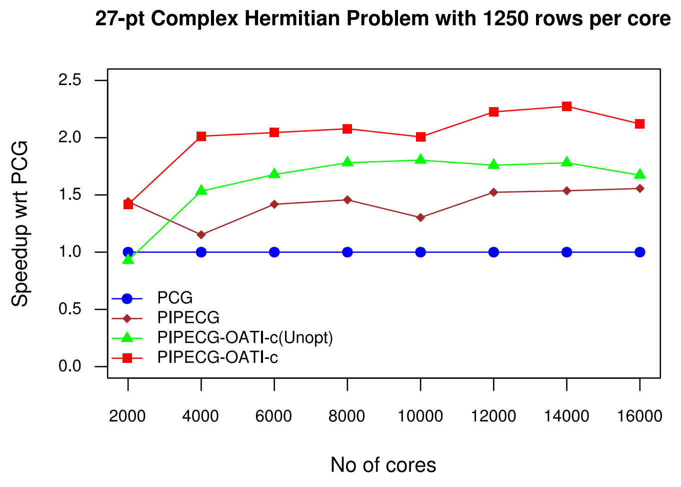


Fig. 5. Weak Scaling results for 27-pt Complex Hermitian problem with 1250 rows per core.

So, for the 27-pt complex Hermitian problem with 20M unknowns, PIPECG-OATI-c optimized provides up to 2.48x speedup wrt PCG-c (at 10k cores), up to 1.35x speedup wrt PIPECG-c (at 10k cores) and up to 1.34x speedup wrt PIPECG-OATI-c unoptimized version (at 10k cores). We compared the time for natural norm convergence for PIPELCG-c and PIPECG-OATI-c and found that PIPECG-OATI-c gives up to 1.5x speedup over PIPELCG-c for this problem (at 10k cores).

We also compared the performance of our method with the other methods for a 5-pt complex symmetric problem with 1 billion unknowns on 16k cores. To our knowledge, this is the largest problem size experimented for the CG method. We found that at 16k cores for this test case, PCG-c method takes 8.2 minutes, PIPECG-c method takes 9.5 minutes, PIPECG-OATI-c unoptimized version takes 10.4 minutes and PIPECG-OATI-c optimized version takes 6 minutes. Thus, we conclude that PIPECG-OATI-c method performs better than the other CG methods even for very large test cases at higher number of cores.

6.4. Weak scaling results

Fig. 5 shows the weak scaling of different methods on a 27-pt 3D complex Hermitian problem with 1250 rows per core on up to 16000 cores. All the methods are run for 200 iterations so that they overcome initialization costs and display cache locality benefits.

We observe from the figure that PIPECG-OATI-c optimized version performs better than the other methods at all the core counts. The overlap provided by PIPECG-OATI-c method along with the merged VecOps makes it the best performing method for this problem followed by PIPECG-OATI-c unoptimized version, which still performs better than PIPECG despite having overhead of extra VecOps. PIPECG performs better than PCG due to its overlap of the allreduce with one PC and SPMV.

Thus, we conclude that as problem size increases with number of cores, PIPECG-OATI-c provides better performance than all the other methods.

6.5. Accuracy results

In the PCG method, the convergence is checked as:

$$\|u_i\| < max(rtol * \|b\|, atol)$$

where u_i is preconditioned residual, $rtol$ is relative tolerance and $atol$ is absolute tolerance. In PETSc based applications, $rtol$ is set to 10^{-5} by default. In the OpenFOAM [2][22] based applications which solve pressure Poisson equations, we see that default value for $rtol$ is set to 10^{-2}.

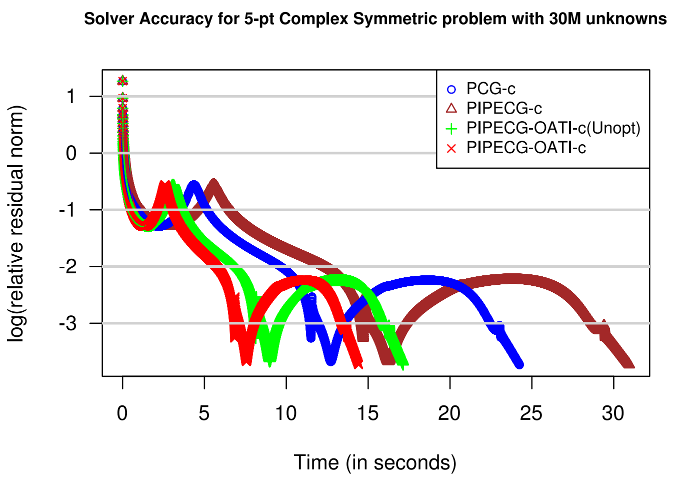


Fig. 6. Solver Accuracy/Performance Experiment for 5-pt complex symmetric problem with 30M unknowns. Relative residual values are plotted as a function of time at 16k cores.

Fig. 6 shows the relative residual values attained by each method as a function of time for the 5-pt complex symmetric problem with 30M unknowns on 16k cores. Here, we see that PIPECG-OATI-c reaches the threshold of $rtol * \|b\|$ (where $rtol$ is set to 10^{-5}) the fastest as compared to all the other methods with PIPECG-c being the slowest. These results align with the strong scaling results for the same problem shown in Fig. 3. Thus, we conclude that for widely used values of $rtol$, our method can be used to solve the linear system of equations obtained from real world applications faster as compared to other methods.

7. Conclusions and future work

In this work, we extended the PIPECG-OATI method from our previous work [29] to support complex Hermitian positive definite and complex symmetric systems and we obtained the PIPECG-OATI-c method. PIPECG-OATI-c reduces the number of allreduces to one per two iterations and overlaps it with two PCs and two SPMVs using MPI_Iallreduce at the cost of introducing five extra dot products to PIPECG-OATI. We also provide support for complex symmetric systems by changing the way complex dot products are calculated. We provide an optimized implementation of PIPECG-OATI-c which helps in efficient memory accesses and hence gives performance improvements. Our optimized version of PIPECG-OATI-c gives a speedup of up to 1.34x over the unoptimized version of PIPECG-OATI-c.

We performed experiments with 20M and 30M unknowns on up to 16K cores and obtained up to 2.48X performance improvement over PCG and 2.14X performance improvement over PIPECG methods. We also experimented with up to 1-billion unknowns on 16K cores, the largest problem size explored for the CG problem, to our knowledge, and obtained about 25% improvement over PCG. We conclude that PIPECG-OATI-c gives performance benefits over PCG-c and PIPECG-c at high number of cores when the allreduce cost becomes more and can be completely overlapped by the two PCs and two SPMVs provided by the complex linear system. Our method also reaches the accuracies used in practical problems the fastest when compared to the other methods.

In future, we plan to reduce the number of allreduces to one per s number of iterations and overlap it with useful work with support for unpreconditioned, preconditioned and natural norms.

We plan to test and analyze the behavior of our method PIPECG-OATI-c on multi-node multi-GPU systems.

CRediT authorship contribution statement

Manasi Tiwari: Conceptualization, Methodology, Implementation, Experimentation, Data Collection, Visualization, Investigation, Writing.

Sathish Vadhiyar: Conceptualization, Experiment Design, Writing, Reviewing, Editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

[1] MPICH 3.3.3, https://www.mpich.org/, 2019.
[2] OpenFOAM, https://www.openfoam.org/, 2019.
[3] S.F. Ashby, T.A. Manteuffel, P.E. Saylor, A taxonomy for conjugate gradient methods, SIAM J. Numer. Anal. 27 (6) (1990) 1542–1568, https://doi.org/10.1137/0727091.
[4] S. Balay, W.D. Gropp, L.C. McInnes, B.F. Smith, Efficient management of parallelism in object oriented numerical software libraries, in: E. Arge, A.M. Bruaset, H.P. Langtangen (Eds.), Modern Software Tools in Scientific Computing, Birkhäuser Press, 1997, pp. 163–202.
[5] S. Balay, S. Abhyankar, M.F. Adams, J. Brown, P. Brune, K. Buschelman, L. Dalcin, A. Dener, V. Eijkhout, W.D. Gropp, D. Karpeyev, D. Kaushik, M.G. Knepley, D.A. May, L.C. McInnes, R.T. Mills, T. Munson, K. Rupp, P. Sanan, B.F. Smith, S. Zampini, H. Zhang, H. Zhang, PETSc Web page, https://www.mcs.anl.gov/petsc, 2019.
[6] S. Balay, S. Abhyankar, M.F. Adams, J. Brown, P. Brune, K. Buschelman, L. Dalcin, A. Dener, V. Eijkhout, W.D. Gropp, D. Karpeyev, D. Kaushik, M.G. Knepley, D.A. May, L.C. McInnes, R.T. Mills, T. Munson, K. Rupp, P. Sanan, B.F. Smith, S. Zampini, H. Zhang, H. Zhang, PETSc users manual, Tech. Rep. ANL-95/11 - Revision 3.13, Argonne National Laboratory, 2020, https://www.mcs.anl.gov/petsc.
[7] U. Catalyurek, C. Aykanat, Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication, IEEE Trans. Parallel Distrib. Syst. 10 (7) (1999) 673–693, https://doi.org/10.1109/71.780863.
[8] A.T. Chronopoulos, C.W. Gear, S-step iterative methods for symmetric linear systems, J. Comput. Appl. Math. 25 (2) (1989) 153–168, https://doi.org/10.1016/0377-0427(89)90045-9.
[9] P. Concus, G.H. Golub, G. Meurant, Block preconditioning for the conjugate gradient method, SIAM J. Sci. Stat. Comput. 6 (1) (1985) 220–252, https://doi.org/10.1137/0906018.
[10] S. Cools, J. Cornelis, W. Vanroose, Numerically stable recurrence relations for the communication hiding pipelined conjugate gradient method, arXiv:1902.03100, 2019.
[11] J. Cornelis, S. Cools, W. Vanroose, The communication-hiding conjugate gradient method with deep pipelines, arXiv:1801.04728, 2018.
[12] E. D'Azevedo, V. Eijkhout, C. Romine, Lapack working note 56: Reducing communication costs in the conjugate gradient algorithm on distributed memory multiprocessors, Tech. rep., USA, 1993.
[13] J. Dongarra, M. Heroux, P. Luszczek, HPCG Benchmark: a new metric for ranking high performance computing systems $_*$, 2015.
[14] P.R. Eller, W. Gropp, Scalable non-blocking preconditioned conjugate gradient methods, in: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '16, IEEE Press, 2016.
[15] R.W. Freund, Conjugate gradient-type methods for linear systems with complex symmetric coefficient matrices, SIAM J. Sci. Stat. Comput. 13 (1) (1992) 425–448, https://doi.org/10.1137/0913023.
[16] P. Ghysels, W. Vanroose, Hiding global synchronization latency in the preconditioned conjugate gradient algorithm, Parallel Comput. 40 (7) (2014) 224–238, https://doi.org/10.1016/j.parco.2013.06.001.
[17] W. Gropp, Update on Libraries for Blue Waters, Tech. rep., http://jointlab.ncsa.illinois.edu/events/workshop3/pdf/presentations/Gropp-Update-on-Libraries.pdf.
[18] M.R. Hestenes, E. Stiefel, Methods of conjugate gradients for solving linear systems, J. Res. Natl. Bur. Stand. 49 (1952) 409–436.
[19] O.G. Johnson, C.A. Micchelli, G. Paul, Polynomial preconditioners for conjugate gradient calculations, SIAM J. Numer. Anal. 20 (2) (1983) 362–376, https://doi.org/10.1137/0720025.
[20] P. Joly, G. Meurant, Complex conjugate gradient methods, Numer. Algorithms 4 (1993) 379–406, https://doi.org/10.1007/BF02145754.
[21] G. Meurant, Multitasking the conjugate gradient method on the cray x-mp/48, Parallel Comput. 5 (3) (1987) 267–280, https://doi.org/10.1016/0167-8191(87)90037-8.
[22] OpenFOAM Poisson solver example, https://cfd.direct/openfoam/user-guide/v6-fvsolution/, 2020.
[23] A. Pinar, M. Heath, Improving performance of sparse matrix-vector multiplication, in: SC '99: Proceedings of the 1999 ACM/IEEE Conference on Supercomputing, 1999, p. 30.
[24] L. Romero, E. Zapata, Data distributions for sparse matrix vector multiplication, Parallel Comput. 21 (4) (1995) 583–605, https://doi.org/10.1016/0167-8191(94)00087-Q.
[25] Y. Saad, Practical use of some Krylov subspace methods for solving indefinite and nonsymmetric linear systems, SIAM J. Sci. Stat. Comput. 5 (1) (1984) 203–228, https://doi.org/10.1137/0905015.
[26] Y. Saad, Iterative Methods for Sparse Linear Systems, 2nd edition, Society for Industrial and Applied Mathematics, USA, 2003.
[27] S. Serra-Capizzano, Conditioning and solution of Hermitian (block) Toeplitz systems by means of preconditioned conjugate gradient methods, Proc. SPIE Int. Soc. Opt. Eng. 2563 (1995) 326–337, https://doi.org/10.1117/12.211409.
[28] O. Tatebe, Y. Oyanagi, Efficient implementation of the multigrid preconditioned conjugate gradient method on distributed memory machines, in: Supercomputing '94:Proceedings of the 1994 ACM/IEEE Conference on Supercomputing, 1994, pp. 194–203.
[29] M. Tiwari, S. Vadhiyar, Pipelined preconditioned conjugate gradient methods for distributed memory systems, in: 27th IEEE International Conference on High Performance Computing, Data, and Analytics, HiPC 2020, Pune, India, December 16–19, 2020, IEEE, 2020, pp. 151–160.
[30] J. White, P. Sadayappan, On improving the performance of sparse matrix-vector multiplication, in: Proceedings Fourth International Conference on High-Performance Computing, 1997, pp. 66–71.

Manasi Tiwari received her B.Tech degree from Institute of Engineering and Technology, India in 2017 and is now pursuing her Phd at Department of Computational and Data Sciences, Indian Institute of Science, Bengaluru, India. Her research interests include parallel linear solvers, distributed computing and heterogeneous computing specially involving GPU accelerators.

Sathish Vadhiyar is a Professor in Supercomputer Education and Research Centre, Indian Institute of Science. He obtained his B.E. degree from the Department of Computer Science and Engineering at Thiagarajar College of Engineering, India in 1997 and received his Master's degree in Computer Science at Clemson University, USA in 1999. He graduated with a Ph.D from the Computer Science Department at University of Tennessee, USA in 2003. His research areas include building application frameworks involving runtime frameworks for irregular applications including graph applications, hybrid CPU-GPU execution strategies, and programming models for accelerator-based systems, acceleration of climate and weather applications on different kinds of parallel systems, middleware for production supercomputer systems and fault tolerance for large-scale systems.