Pipelined Preconditioned s-step Conjugate Gradient Methods for Distributed Memory Systems

Manasi Tiwari
Department of Computational and Data Sciences
Indian Institute of Science
Bengaluru, India
manasitiwari@iisc.ac.in

Sathish Vadhiyar
Department of Computational and Data Sciences
Indian Institute of Science
Bengaluru, India
vss@iisc.ac.in

***Abstract*—Preconditioned Conjugate Gradient (PCG) method is a widely used iterative method for solving large linear systems of equations. Pipelined variants of PCG present independent computations in the PCG method and overlap these computations with non-blocking allreduces. We have developed a novel pipelined PCG algorithm called PIPE-sCG (Pipelined s-step Conjugate Gradient) that provides a large overlap of global communication and computations at higher number of cores in distributed memory CPU systems. Our method achieves this overlap by introducing new recurrence computations. We have also developed a preconditioned version of PIPE-sCG. The advantages of our methods are that they do not introduce any extra preconditioner or sparse matrix vector product kernels in order to provide the overlap and can work with preconditioned, unpreconditioned and natural norms of the residual, as opposed to the state-of-the-art methods. We compare our method with other pipelined CG methods for Poisson problems and demonstrate that our method gives the least runtimes. Our method gives up to 2.9x speedup over PCG method, 2.15x speedup over PIPECG method and 1.2x speedup over PIPECG-OATI method at large number of cores.**

***Index Terms*—Preconditioned Conjugate Gradient, Distributed Memory Systems, Pipelining, Overlapping communication and computations, s-step Methods**

I. Introduction

Solving partial differential equations (PDEs) over space and time is a key component of many high performance computing applications in computational fluid dynamics, thermal simulations and so on. The PDEs obtained from such applications are discretized using finite volume, finite element or finite difference methods. These discretization methods result in a linear system of equations $Ax = b$. Generally, the A matrix obtained by using these schemes is large and sparse and such systems can be solved using iterative methods. A popular class of iterative methods used for solving these systems are Krylov Subspace methods. The basic idea behind Krylov Subspace 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) [1] [2] method is a widely used Krylov Subspace method which is instrumental in finding the solution of linear systems with symmetric sparse positive definite matrices. A preconditioner is often applied to the system to condition the input system and to improve convergence.

The key computational kernels in Preconditioned Conjugate Gradient (PCG) method 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 bottleneck in PCG for distributed memory systems is the synchronization that happens across all cores due to the blocking allreduce operations in the algorithm.

The existing research by Saad [3], Meurant [4], Azevedo et al. [5] and Chronopoulos et al. [6] has worked on reducing the number of allreduces to one per iteration as opposed to the three that exists in the original PCG. Additionally in [6], Chronopoulos et al. propose the s-step CG (sCG) method in which the number of allreduces are reduced to one per s iterations at the expense of introducing one extra SPMV kernel. The preconditioned version of the s-step CG method (PsCG) is presented in [7]. The non-blocking collectives like MPI_IAllreduce were introduced in the MPI-3 standard [8] and the overlapping of allreduce with useful work was made possible. Pipelined PCG (PIPECG) proposed by Ghysels et al [9] uses a CG variant which has one allreduce per iteration. By introducing extra VMAs, they overlap this non-blocking allreduce with an SPMV and a PC.

In this work, we propose a novel pipelined PCG method for distributed memory systems, called PIPE-sCG (Pipelined s-step Conjugate Gradient) which has one allreduce per s iterations and overlaps it with s SPMVs and s can be defined at run time. We start with sCG [6] which has s+1 SPMVs per iteration and reduce the number of SPMVs to s per iteration. Then, we eliminate the dependencies between the dot products and s SPMVs by using new recurrence computations and thereafter overlap the non-blocking allreduce for dot products with s SPMVs. We also propose a preconditioned version of PIPE-sCG called PIPE-PsCG. For this, we introduce preconditioning to the PIPE-sCG method. As opposed to the preconditioned sCG(i.e. PsCG) method which has s+1 PCs and s+1 SPMVs in each iteration, the PIPE-PsCG method has s PCs and s SPMVs in each iteration. Then, the non-blocking allreduce for dot products can be overlapped with these s PCs and s SPMVs. Furthermore, PIPE-PsCG method can compare the user defined tolerance with any of preconditioned, unpreconditioned or natural residual norm without introducing any extra PC or SPMV kernels as opposed to state-of-the-art methods. We perform a detailed cost analysis of the PIPE-PsCG method and compare it with other PCG variants. We test the performance of PIPE-PsCG method on various problems and present results. PIPE-PsCG gives up to 2.9x speedup over PCG method, 2.15x speedup over PIPECG method and 1.2x speedup over PIPECG-OATI method at large number of cores.

The rest of the paper is organized as follows: Section II gives the related work, Section III gives background related to PCG, s-step CG and preconditioned s-step CG, Section IV describes our algorithm for PIPE-sCG and preconditioned PIPE-sCG method using recurrence computations, Section V presents the computational cost analysis and comparison of our method to other related works. Section VI presents experiments, results and discussions for our proposed method and Section VII gives the conclusion and future work.

II. Related Work

Pipelined variants of PCG attempt to eliminate the dependencies in the computations of the PCG algorithm and overlap the resultant non-dependent computations using non-blocking allreduces. Apart from the PIPECG method [9] mentioned in section I, there have been other efforts for overlapping communication and computations.

PIPECG3, a pipelined version of the PCG method with three term recurrence relations was proposed by Eller et al [10]. It launches a single allreduce in every two iterations and overlaps the single allreduce with two SPMVs and two PCs. However, this has been shown to have low accuracy than the original three two-term recurrence PCG variants. In our previous work [11], we had proposed PIPECG-OATI (One Allreduce per Two Iterations) in which we used iteration combination and non-recurrence computations to launch one non-blocking allreduce in two iterations and overlap it with two PCs and two SPMVs. We had shown that PIPECG-OATI gives up to 3x performance improvement over all the other state-of-art pipelined methods. While this previous work uses one allreduce per two iterations, our current work aims to provide a flexible framework to use one allreduce every s iterations and overlap it with s PCs and s SPMVs.

Pipelined PCG with deep pipelines (PIPELCG) was introduced in [12] [13] in order to overlap more work with allreduce at higher core counts. PIPELCG starts with Generalised Minimal Residual (GMRES) as the base algorithm and overlaps each allreduce with s PCs and s SPMVs. However, it launches a non-blocking allreduce in each iteration and can only work with natural norm of the residual. To accommodate unpreconditioned and preconditioned norms, PIPELCG would require an extra PC and SPMV in each iteration. Our method PIPE-sCG starts with sCG (s-step Conjugate Gradient) as the base algorithm. It launches one non-blocking allreduce in s iterations and overlaps it with s PCs and s SPMVs. PIPE-sCG can work with all kinds of norms without introducing any extra PC and SPMV.

Communication avoiding s-step CG method is proposed by Hoemmen in [14]. It used the Matrix Powers kernel which reduces the communication due to SPMVs. However, the use of matrix powers kernel with s-step CG makes it difficult to use certain preconditioners. Hence, they propose a Communication avoiding CG (CA-CG) method which is based on the three term recurrence variant of CG method. CA-CG uses matrix powers kernel and can be used with any preconditioner. PIPE-PsCG doesn't use matrix powers kernel as the aim of our work is to overlap the global communication induced by allreduce, and not reduce the communication induced by SPMV itself. PIPE-PsCG can work with any preconditioner. The matrix powers kernel can be used with PIPE-PsCG to reduce the SPMV communication but it can prevent from using certain preconditioners. Therefore, we don't use matrix powers kernel in our work.

III. Background

PCG: The Preconditioned Conjugate Gradient Method (PCG) introduced by Hestenes and Stiefel [1] is given in Algorithm 1. It iteratively solves $M^{-1}Ax = M^{-1}b$ where both M and A are symmetric and positive definite square matrices of size NxN. The method starts with an initial

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:    $\alpha = \gamma_i/\delta$
13:    $x_{i+1} = x_i + \alpha p_i$
14:    $r_{i+1} = r_i - \alpha s$
15:    $u_{i+1} = M^{-1}r_{i+1}$
16:    $\gamma_{i+1} = (u_{i+1}, r_{i+1})$;
17:    $norm_{i+1} = \sqrt{(u_{i+1}, u_{i+1})}$
18: end for
```

solution x_0. The residual vector of the original system is $r_i = b - Ax_i$, the residual of the preconditioned system is $u_i = M^{-1}r_i$ and p_i is the direction vector. We keep updating x_0 with the new direction vector in each iteration till the residual norm reaches a user defined tolerance. If the exact solution is $x^* = A^{-1}b$, then the error at step i is defined as $e_i = x^* - x_i$. All subsequent approximations x_i lie in a Krylov subspace $K_i(M^{-1}A, u_0)$, which is defined as:
$K_i(M^{-1}A, u_0) = \text{span}\{u_0, M^{-1}Au_0, .., (M^{-1}A)^{i-1}u_0\}$

The CG iteration generates a sequence of iterates $x_i \in x_0 + K_i(M^{-1}A, u_0)$, with the property that at step i, the error functional $\|e_i\|_A = \sqrt{e_i{}^T A e_i}$ is minimized.

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 15, Vector-Multiply-Adds (VMAs) in lines 9, 13 and 14 and dot products in lines 11, 16 and 17. The SPMV often only requires communication with the neighbouring nodes which has been implemented efficiently in state-of-the-art libraries. Depending on the type of PC we are using, there might be no communication at all or communication with neighbouring nodes. Communication efficient PCs already exist in state-of-the-art libraries. VMAs require no communication.

Dot products use allreduce which requires all the cores 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, 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.

s-step CG (sCG): The s-step Conjugate Gradient Method (sCG) proposed by Chronopoulos et al. [6] is given in Algorithm 2. It iteratively solves $Ax = b$. The basic idea behind the sCG method is to perform s consecutive iterations of the original CG method in one iteration. While each iteration of the PCG method computes a direction vector and minimizes the error functional for that direction vector, each iteration of the sCG method forms direction spaces instead of single direction vectors (as in CG), and minimizes the error functional over each space.

In algorithm 2, we first select an initial solution x_0. Then, we use the s linearly independent directions $\{r_i, .., A^{s-1}r_i\}$ to lift the iteration s dimensions out of the ith step Krylov subspace $\{r_0, .., A^{is}r_0\}$. These linearly independent directions are stored in Q. Then, we calculate 2s vector moments or dot products as we will need them for calculating $\alpha's$ and $\beta's$. The dot products that we need to calculate are:
$vm = \{(r_0, r_0), (r_0, Ar_0), (r_0, A^2r_0), .., (r_0, A^sr_0),$
$(Ar_0, A^sr_0), .., (A^{s-1}r_0, A^sr_0)\}$.

For even iterates, the linearly independent directions must be made A-conjugate to the preceding s directions, which are $\{{p_{i-1}}^1, .., {p_{i-1}}^s\}$ stored in P. For this we need the $\beta's$. After the linearly independent directions are made A-conjugate to the previous directions, they are stored in Q and their entries are called $\{{p_i}^1, .., {p_i}^s\}$ Finally, the error functional must be minimized simultaneously in all s new directions to obtain the new solution vector x_{i+1} and residual r_{i+1}. For this we need the $\alpha's$. The vector of $\alpha's$ and the matrix of $\beta's$ are calculated in Scalar Work. For the odd iterates, exact same steps are followed but the $\{{p_i}^1, .., {p_i}^s\}$ are stored in P and the $\{{p_{i-1}}^1, .., {p_{i-1}}^s\}$ stored in Q. For an elaborate description of the sCG method, see [6].

As shown in Algorithm 2, the computational kernels in sCG's for loop are s+1 SPMVs in lines 11, 12 (for even iterates), lines 17, 18 (for odd iterates); Linear Combinations (LCs) in lines 9, 10(even iterates), 15, 16(odd iterates); and 2s dot products in line 13 (even iterates) and 19 (odd iterates). The scalar work on line 7 involves solving two s x s linear systems using any direct solver. We use LU factorization for solving these systems and obtain $\alpha's$ and $\beta's$. Since, in an iteration the method will execute either the if or the $else$ clause, we observe that in each iteration, there will be s+1 SPMVs and 2s dot products. However, the allreduce for these 2s dot products can be combined into a single allreduce. Thus the s-step CG method provides one allreduce in each iteration where each iteration does the work of s consecutive steps of the PCG method. Effectively, s-step CG method provides one allreduce per s iterations of the PCG method.

Algorithm 2 s-step Conjugate Gradient Method (sCG)

```
1: Select $x_0$
2: Set $P = 0$
3: Compute $Q = \{r_0 = b - Ax_0, Ar_0, A^2r_0, .., A^{s-1}r_0\}$
4: Compute $A^sr_0$
5: $vm = \{(r_0, r_0), (r_0, Ar_0), ...(A^{s-1}r_0, A^sr_0)\}$
6: for i=0,1,2.. do
7:    Scalar Work
8:    if i even then
9:       $Q = Q + P[\beta^1, \beta^2, ...., \beta^s]$
10:      $x_{i+1} = x_i + Q\alpha$
11:      $P = \{r_{i+1} = b - Ax_{i+1}, Ar_{i+1}, .., A^{s-1}r_{i+1}\}$
12:      Compute $A^sr_{i+1}$
13:      $vm = \{(r_{i+1}, r_{i+1}), .., (A^{s-1}r_{i+1}, A^sr_{i+1})\}$
14:   else
15:      $P = P + Q[\beta^1, \beta^2, ...., \beta^s]$
16:      $x_{i+1} = x_i + P\alpha$
17:      $Q = \{r_{i+1} = b - Ax_{i+1}, Ar_{i+1}, .., A^{s-1}r_{i+1}\}$
18:      Compute $A^sr_{i+1}$
19:      $vm = \{(r_{i+1}, r_{i+1}), .., (A^{s-1}r_{i+1}, A^sr_{i+1})\}$
20:   end if
21: end for
```

Preconditioned s-step CG (PsCG): The Preconditioned s-step Conjugate Gradient Method (PsCG) introduced in [7] is given in Algorithm 3. It iteratively solves $M^{-1}Ax = M^{-1}b$. Here, $\{u_i = M^{-1}r_i, .., (M^{-1}A)^{s-1}u_i\}$ are the linearly independent direction vectors used to lift the iteration s dimensions out of the ith Krylov Subspace. The sCG method generates a sequence of iterates $x_i \in x_0 + K_i(A, r_0)$ whereas the PsCG method generates a sequence of iterates $x_i \in x_0 + K_i(M^{-1}A, u_0)$. As shown in Algorithm 3, the computational kernels in PsCG for loop are s+1 PCs and s+1 SPMVs in lines 12, 13, 14 (even iterates) , 19, 20 and 21(odd iterates); LCs in lines 10, 11 (even iterates), 17, 18(odd iterates); and 2s dot products in line 15(even iterates) and 22(odd iterates).

IV. Methodology

As shown in Algorithm 2, one iteration of sCG actually performs the work of s iterations of the PCG method. However, s iterations of the PCG method have s SPMVs whereas one

Algorithm 3 Preconditioned s-step Conjugate Gradient Method (PsCG)

```
1: Select $x_0$
2: Set $P = 0$
3: Compute $r_0 = b - Ax_0$, $u_0 = M^{-1}r_0$
4: $Q = \{u_0, M^{-1}Au_0, .., (M^{-1}A)^{s-1}u_0\}$
5: Compute $(M^{-1}A)^s u_0$
6: $vm = \{(r_0, u_0), (r_0, M^{-1}Au_0), (r_0, (M^{-1}A)^2 u_0), ..,$
       $(A(M^{-1}A)^{s-1}u_0, (M^{-1}A)^s u_0)\}$
7: for i=0,1,2.. do
8:     Scalar Work
9:     if i even then
10:        $Q = Q + P[\beta^1, \beta^2, ...., \beta^s]$
11:        $x_{i+1} = x_i + Q\alpha$
12:        Compute $r_{i+1} = b - Ax_{i+1}$, $u_{i+1} = M^{-1}r_{i+1}$
13:        $P = \{u_{i+1}, M^{-1}Au_{i+1}, ..(M^{-1}A)^{s-1}u_{i+1}\}$
14:        Compute $(M^{-1}A)^s u_{i+1}$
15:        $vm = \{(r_{i+1}, u_{i+1}), (r_{i+1}, M^{-1}Au_{i+1}), ..,$
               $(A(M^{-1}A)^{s-1}u_{i+1}, (M^{-1}A)^s u_{i+1})\}$
16:    else
17:        $P = P + Q[\beta^1, \beta^2, ...., \beta^s]$
18:        $x_{i+1} = x_i + P\alpha$
19:        Compute $r_{i+1} = b - Ax_{i+1}$, $u_{i+1} = M^{-1}r_{i+1}$
20:        $Q = \{u_{i+1}, M^{-1}Au_{i+1}, ..(M^{-1}A)^{s-1}u_{i+1}\}$
21:        Compute $(M^{-1}A)^s u_{i+1}$
22:        $vm = \{(r_{i+1}, u_{i+1}), (r_{i+1}, M^{-1}Au_{i+1}), ..,$
               $(A(M^{-1}A)^{s-1}u_{i+1}, (M^{-1}A)^s u_{i+1})\}$
23:    end if
24: end for
```

iteration of sCG has s+1 SPMVs. So, there is an extra SPMV in the sCG iteration. SPMV is the most computationally expensive kernel in the PCG iteration and even one extra SPMV provides a large overhead. Furthermore, as shown in Algorithm 2, the dot products need the results of the SPMVs that happen immediately before them. Thus, it is imperative that the SPMVs are executed before dot products. Similarly, the computations that follow the dot products need the results of the dot products. Hence, while the allreduce for the dot product progresses, the cores have to be idle as there is no independent computation that can be overlapped with that communication. Similarly, for PsCG, one iteration has s+1 PCs as compared to s PCs in s iterations of the PCG method. PC is also a computationally expensive kernel which provides overhead. Additionally, as shown in algorithm 3, the dot products depend on the result of the PCs and SPMVs before them. There is a dependency between the PCs, SPMVs and dot products and there are no independent computations that can be overlapped with the allreduce.

In the upcoming era of exascale supercomputers, when we run the sCG method at higher number of cores, the allreduce cost will become the most dominant term in the total execution time. It is necessary that we overlap this allreduce cost with useful computations so that the cores don't remain idle.

In order to build communication overlapping variant of the sCG method, we have to eliminate the dependencies between the dot products and the SPMVs so that the SPMVs can be executed by the cores while the communication for non-blocking allreduce of the dot products takes place. Hence, the cores wouldn't have to be idle. Still, the sCG method has one drawback which would overshadow the benefits of overlapping the allreduce communication with SPMVs. That drawback is the extra SPMV in each iteration. If we expect to see any kind of performance benefits from overlapping the communication and computations in sCG when compared to the original PCG and other pipelined variants, the amount of computations in all methods should be same. Therefore, the primary challenge here is to remove the extra SPMV from the sCG method. Once we have the sCG method with s SPMVs, we can proceed to eliminate the dependencies between the dot products and SPMVs.

This section is organized as follows. We first derive the sCG method with s SPMVs in IV-A. We then develop the pipelined sCG method in IV-B. Finally, we derive the preconditioned version of the pipelined sCG method in IV-C.

A. *sCG method with s SPMVs*

In algorithm 2, as we examine the s+1 SPMVs that are calculated, we observe that r_{i+1} is calculated as $r_{i+1} = b - Ax_{i+1}$. We know that $x_{i+1} = x_i + Q\alpha$ for even iterates and $x_{i+1} = x_i + P\alpha$ for odd iterates. The entries of P or Q are $\{{p_i}^1, {p_i}^2, .., {p_i}^s\}$.
Therefore, the expansion of x_{i+1} is:

$x_{i+1} = x_i + {\alpha_i}^1 {p_i}^1 + {\alpha_i}^2 {p_i}^2 + .. + {\alpha_i}^s {p_i}^s$

Substituting x_{i+1} in $r_{i+1} = b - Ax_{i+1}$, we get:

$r_{i+1} = b - A(x_i + {\alpha_i}^1 {p_i}^1 + {\alpha_i}^2 {p_i}^2 + .. + {\alpha_i}^s {p_i}^s)$

$\implies r_{i+1} = b - Ax_i - {\alpha_i}^1 A{p_i}^1 - {\alpha_i}^2 A{p_i}^2 - .. - {\alpha_i}^s A{p_i}^s$

Substituting $r_i = b - Ax_i$:

$r_{i+1} = r_i - {\alpha_i}^1 A{p_i}^1 - {\alpha_i}^2 A{p_i}^2 - .. - {\alpha_i}^s A{p_i}^s$

$\implies r_{i+1} = r_i - \{A{p_i}^1, A{p_i}^2, ..., A{p_i}^s\}\alpha$

We introduce two matrices, AQ and AP. Their initial entries are $\{Ar_i, A^2 r_i.., A^s r_i\}$. For even iterates, AQ has to be made A-conjugate to AP which has $\{A{p_{i-1}}^1, A{p_{i-1}}^2, .., A{p_{i-1}}^s\}$. So, we introduce a recurrence relation:

$AQ = AQ + AP[\beta^1, \beta^2, ..., \beta^s]$

After this, the contents of AQ are:

$AQ = \{A{p_i}^1, A{p_i}^2, .., A{p_i}^s\}$

For even iterates, AP has to be made A-conjugate to AQ which has $\{A{p_{i-1}}^1, A{p_{i-1}}^2, .., A{p_{i-1}}^s\}$. So, we introduce a linear combination which is a recurrence relation:

$AP = AP + AQ[\beta^1, \beta^2, ..., \beta^s]$

After this, the contents of AP are:

$AP = \{A{p_i}^1, A{p_i}^2, .., A{p_i}^s\}$

Thus r_{i+1} becomes:

$r_{i+1} = r_i - AQ\alpha$ for even iterates.

$r_{i+1} = r_i - AP\alpha$ for odd iterates.

All the new matrices, their initial entries and the recurrence LCs to populate them are shown in algorithm 4 in red. Putting it all together, we have obtained sCG method with s SPMVs in each iteration, present on line 14 and 15 (even iterates) and on line 23 and 24 (odd iterates).

Algorithm 4 sCG method with s SPMVs

```
1: Select $x_0$
2: Set $P = 0$, $AP = 0$
3: Compute $Q = \{r_0 = b - Ax_0, Ar_0, A^2r_0, .., A^{s-1}r_0\}$
4: Compute $A^sr_0$
5: $vm = \{(r_0, r_0), (r_0, Ar_0), ...(A^{s-1}r_0, A^sr_0)\}$
6: for i=0,1,2.. do
7:     Scalar Work
8:     if i even then
9:         $AQ = \{Ar_i, A^2r_i, .., A^sr_i\}$
10:        $Q = Q + P[\beta^1, \beta^2, ...., \beta^s]$
11:        $AQ = AQ + AP[\beta^1, \beta^2, ...., \beta^s]$
12:        $x_{i+1} = x_i + Q\alpha$
13:        $r_{i+1} = r_i - AQ\alpha$
14:        $P = \{r_{i+1}, Ar_{i+1}, .., A^{s-1}r_{i+1}\}$
15:        Compute $A^sr_{i+1}$
16:        $vm = \{(r_{i+1}, r_{i+1}), .., (A^{s-1}r_{i+1}, A^sr_{i+1})\}$
17:    else
18:        $AP = \{Ar_i, A^2r_i, .., A^sr_i\}$
19:        $P = P + Q[\beta^1, \beta^2, ...., \beta^s]$
20:        $AP = AP + AQ[\beta^1, \beta^2, ...., \beta^s]$
21:        $x_{i+1} = x_i + P\alpha$
22:        $r_{i+1} = r_i - AP\alpha$
23:        $Q = \{r_{i+1}, Ar_{i+1}, .., A^{s-1}r_{i+1}\}$
24:        Compute $A^sr_{i+1}$
25:        $vm = \{(r_{i+1}, r_{i+1}), .., (A^{s-1}r_{i+1}, A^sr_{i+1})\}$
26:    end if
27: end for
```

B. PIPE-sCG method

In order to overlap the non-blocking allreduce of the dot products with SPMVs, we have to eliminate the dependencies between them. As we can observe from algorithm 4, the dot products use the result of the s SPMVs i.e. $\{Ar_{i+1}, A^2r_{i+1}, .., A^sr_{i+1}\}$.
For even iterates, we have:
$r_{i+1} = r_i - AQ\alpha$
Then we can obtain $\{Ar_{i+1}, A^2r_{i+1}, .., A^sr_{i+1}\}$ using the following recurrence LC (linear combination):

$$Ar_{i+1} = Ar_i - A(AQ)\alpha$$
$$A^2r_{i+1} = A^2r_i - A^2(AQ)\alpha$$
$$\vdots$$
$$A^sr_{i+1} = A^sr_i - A^s(AQ)\alpha$$

We introduce a matrix of matrices AQm whose entries will be $\{AQ, A(AQ), .., A^s(AQ)\}$. The initial entries of AQm are:

$$AQm[0] = AQ = \{Ar_i, A^2r_i, .., A^sr_i\}$$
$$AQm[1] = A(AQ) = \{A^2r_i, A^3r_i, .., A^{s+1}r_i\}$$
$$AQm[2] = A^2(AQ) = \{A^3r_i, A^4r_i, .., A^{s+2}r_i\}$$
$$\vdots$$

Algorithm 5 Pipelined sCG method with s SPMVs (PIPE-sCG)

```
1: Select $x_0$
2: Set $P = 0$
3: for i=0,1,..,s do
4:     $APm[i] = 0$
5: end for
6: Compute $Q = \{r_0 = b - Ax_0, Ar_0, A^2r_0, .., A^{s-1}r_0\}$
7: Compute $A^sr_0$
8: $vm = \{(r_0, r_0), (r_0, Ar_0), ...(A^{s-1}r_0, A^sr_0)\}$
9: MPI_Iallreduce on vm
10: Compute $A^{s+1}r_0, ..., A^{2s}r_0$
11: for i=0,1,2.. do
12:     Scalar Work
13:     if i even then
14:         for i=0,1,..,s do
15:             $AQm[i] = A^{i+1}Q$
16:         end for
17:         $Q = Q + P[\beta^1, \beta^2, ...., \beta^s]$
18:         for j=0,1,..,s do
19:             $AQm[j] = AQm[j] + APm[j][\beta^1, \beta^2, ...., \beta^s]$
20:         end for
21:         $x_{i+1} = x_i + Q\alpha$
22:         for j=0,1,..,s-1 do
23:             $P[j] = Q[j] - AQm[j]\alpha$
24:         end for
25:         $A^sr_{i+1} = A^sr_i - AQm[s]\alpha$
26:         $vm = \{(r_{i+1}, r_{i+1}), .., (A^{s-1}r_{i+1}, A^sr_{i+1})\}$
27:         MPI_Iallreduce on vm
28:         Compute $A^{s+1}r_{i+1}, ..., A^{2s}r_{i+1}$
29:     else
30:         for i=0,1,..,s do
31:             $APm[i] = A^{i+1}P$
32:         end for
33:         $P = P + Q[\beta^1, \beta^2, ...., \beta^s]$
34:         for j=0,1,..,s do
35:             $APm[j] = APm[j] + AQm[j][\beta^1, \beta^2, ...., \beta^s]$
36:         end for
37:         $x_{i+1} = x_i + P\alpha$
38:         for j=0,1,..,s-1 do
39:             $Q[j] = P[j] - APm[j]\alpha$
40:         end for
41:         $A^sr_{i+1} = A^sr_i - APm[s]\alpha$
42:         $vm = \{(r_{i+1}, r_{i+1}), .., (A^{s-1}r_{i+1}, A^sr_{i+1})\}$
43:         MPI_Iallreduce on vm
44:         Compute $A^{s+1}r_{i+1}, ..., A^{2s}r_{i+1}$
45:     end if
46: end for
```

$AQm[s] = A^s(AQ) = \{A^{s+1}r_i, A^{s+2}r_i, .., A^{2s}r_i\}$

We observe that entries of AQm require the s SPMVs $A^{s+1}r_i, A^{s+2}r_i, .., A^{2s}r_i$. But we had not calculated these SPMVs earlier. So, we have to compute these s SPMVs. However, we notice that the result of the new SPMVs is not needed by the dot products. So the new s SPMVs can be computed while the allreduce of the dot products progresses. Also, the entry $AQm[i]$ has to be made A-conjugate to entry $APm[i]$. So, we introduce the recurrence LCs:

$AQm[0] = AQm[0] + APm[0][\beta^1, \beta^2, ..., \beta^s]$

$AQm[1] = AQm[1] + APm[1][\beta^1, \beta^2, ..., \beta^s]$

$\vdots$

$AQm[s] = AQm[s] + APm[s][\beta^1, \beta^2, ..., \beta^s]$

For odd iterates, the same procedure as above is followed, except that we define a matrix of matrices called APm whose initial entries are defined in the same way as even iterates. Then these entries are made A-conjugate to AQm. Finally the s SPMVs $\{Ar_{i+1}, A^2r_{i+1}, .., A^sr_{i+1}\}$ are calculated as:

$Ar_{i+1} = Ar_i - A(AP)\alpha$

$A^2r_{i+1} = A^2r_i - A^2(AP)\alpha$

$\vdots$

$A^sr_{i+1} = A^sr_i - A^s(AP)\alpha$

All the new matrices, their initial entries, the recurrence LCs to update them and the new SPMVs are shown in algorithm 5 in red. Putting it all together, we have obtained a pipelined sCG (PIPE-sCG) method with s SPMVs in each iteration, present on line 28 (even iterates) and on line 44 (odd iterates). These s SPMVs can be overlapped with a non-blocking allreduce present on line 27(even iterates) and line 43(odd iterates).

C. *PIPE-PsCG method*

We apply preconditioning to PIPE-sCG method in algorithm 5. For even iterates, when we apply preconditioning, the dot products to be calculated (as shown in algorithm 3) are: $vm = \{(r_{i+1}, u_{i+1}), (r_{i+1}, M^{-1}Au_{i+1}), .., (A(M^{-1}A)^{s-1}u_{i+1}, (M^{-1}A)^su_{i+1})\}$.
Calculation of vm requires r_{i+1}. In preconditioned method, the entries of P are $\{u_{i+1}, M^{-1}Au_{i+1}, ..(M^{-1}A)^{s-1}u_{i+1}\}$. It stores the preconditioned residual u_{i+1} and not r_{i+1}. So, we introduce a matrix $P2$ which stores $\{r_{i+1}, AM^{-1}r_{i+1}, ..(AM^{-1})^{s-1}r_{i+1}\}$. The rest of the entries are useful in intermediate computations. Since, we cannot calculate the entries of $P2$ directly using PC and SPMV kernels, we introduce the following recurrence LC for calculating it: $P2[j] = Q2[j] - AQ2m[j]\alpha$.
$Q2$ has the same entries as $P2$ but from odd iterates. $Q2$ is made A-conjugate to previous $P2$ using the recurrence LC: $Q2 = Q2 + P2[\beta^1, \beta^2, ..., \beta^s]$.
We introduce a new matrix of matrices $AQ2m$, which has the initial entries $\{(AM^{-1})Q2, (AM^{-1})^2Q2), .., (AM^{-1})^{s+1}Q2)\}$.
Then $AQ2m[j]$ is made A-conjugate to $AP2m[j]$ using the recurrence LCs:

Algorithm 6 Pipelined Preconditioned sCG method(PIPE-PsCG)

```
1: Select $x_0$
2: Set $P = 0$, $P2 = 0$
3: for i=0,1,..,s do
4:     $APm[i] = 0$
5:     $AP2m[i] = 0$
6: end for
7: Compute $r_0 = b - Ax_0$, $u_0 = M^{-1}r_0$
8: Compute $AM^{-1}r_0, M^{-1}AM^{-1}r_0, ..., (M^{-1}A)^sM^{-1}r_0$
9: $Q = \{u_0, M^{-1}Au_0, .., (M^{-1}A)^{s-1}u_0\}$
10: $Q2 = \{r_0, AM^{-1}r_0, .., (AM^{-1})^{s-1}r_0\}$
11: $vm = \{(r_0, u_0), (r_0, M^{-1}Au_0), (r_0, (M^{-1}A)^2u_0), ..,$
        $(A(M^{-1}A)^{s-1}u_0, (M^{-1}A)^su_0)\}$
12: MPI_Iallreduce on vm
13: Compute $A(M^{-1}A)^sM^{-1}r_0, (M^{-1}A)^{s+1}M^{-1}r_0, ...,$
        $(M^{-1}A)^{2s}M^{-1}r_0$
14: for i=0,1,2.. do
15:     Scalar Work
16:     if i even then
17:         for i=0,1,..,s do
18:             $AQm[i] = (M^{-1}A)^{i+1}Q$
19:             $AQ2m[i] = (AM^{-1})^{i+1}Q2$
20:         end for
21:         $Q = Q + P[\beta^1, \beta^2, ...., \beta^s]$
22:         $Q2 = Q2 + P2[\beta^1, \beta^2, ...., \beta^s]$
23:         for j=0,1,..,s do
24:             $AQm[j] = AQm[j] + APm[j][\beta^1, \beta^2, ...., \beta^s]$
25:             $AQ2m[j] = AQ2m[j] + AP2m[j][\beta^1, \beta^2, ...., \beta^s]$
26:         end for
27:         $x_{i+1} = x_i + Q\alpha$
28:         for j=0,1,..,s-1 do
29:             $P[j] = Q[j] - AQm[j]\alpha$
30:             $P2[j] = Q2[j] - AQ2m[j]\alpha$
31:         end for
32:         $(AM^{-1})^sr_{i+1} = (AM^{-1})^sr_i - AQm[s]\alpha$
33:         $(M^{-1}A)^su_{i+1} = (M^{-1}A)^su_i - AQ2m[s]\alpha$
34:         $vm = \{(r_{i+1}, u_{i+1}), (r_{i+1}, M^{-1}Au_{i+1}), ..,$
                $(A(M^{-1}A)^{s-1}u_{i+1}, (M^{-1}A)^su_{i+1})\}$
35:         MPI_Iallreduce on vm
36:         Compute $A(M^{-1}A)^sM^{-1}r_{i+1}, (M^{-1}A)^{s+1}M^{-1}r_{i+1},$
                $.., (M^{-1}A)^{2s}M^{-1}r_{i+1}$
37:     else
38:         Call Odd Iterates
39:     end if
40: end for
```

Algorithm 7 Odd Iterates

```
1: for i=0,1,..,s do
2:     $APm[i] = (M^{-1}A)^{i+1}P$
3:     $AP2m[i] = (AM^{-1})^{i+1}P2$
4: end for
5: $P = P + Q[\beta^1, \beta^2, ...., \beta^s]$
6: $P2 = P2 + Q2[\beta^1, \beta^2, ...., \beta^s]$
7: for j=0,1,..,s do
8:     $APm[j] = APm[j] + AQm[j][\beta^1, \beta^2, ...., \beta^s]$
9:     $AP2m[j] = AP2m[j] + AQ2m[j][\beta^1, \beta^2, ...., \beta^s]$
10: end for
11: $x_{i+1} = x_i + P\alpha$
12: for j=0,1,..,s-1 do
13:     $Q[j] = P[j] - APm[j]\alpha$
14:     $Q2[j] = P2[j] - AP2m[j]\alpha$
15: end for
16: $(AM^{-1})^s r_{i+1} = (AM^{-1})^s r_i - AQm[s]\alpha$
17: $(M^{-1}A)^s u_{i+1} = (M^{-1}A)^s u_i - AQ2m[s]\alpha$
18: $vm = \{(r_{i+1}, u_{i+1}), (r_{i+1}, M^{-1}Au_{i+1}), ..,$
        $(A(M^{-1}A)^{s-1}u_{i+1}, (M^{-1}A)^s u_{i+1})\}$
19: MPI_Iallreduce on vm
20: Compute $A(M^{-1}A)^s M^{-1} r_{i+1}, (M^{-1}A)^{s+1} M^{-1} r_{i+1},$
        $.., (M^{-1}A)^{2s} M^{-1} r_{i+1}$
```

$$AQ2m[j] = AQ2m[j] + AP2m[j][\beta^1, \beta^2, ..., \beta^s]$$

We introduce similar recurrence LCs for odd iterates. All the new matrices, their initial entries, the recurrence LCs to update them and the new PCs and SPMVs are shown in algorithm 6 and 7 in red. Putting it all together, we have obtained a pipelined PsCG (PIPE-PsCG) method with s PCs and s SPMVs in each iteration, present on line 36 (even iterates) of algorithm 6 and on line 20 (odd iterates) of algorithm 7. These s PCs and s SPMVs can be overlapped with a non-blocking allreduce present on line 35(even iterates) and line 19(odd iterates).

V. Computational cost analysis and comparison with different methods

In this section, we analyse and compare PIPE-PsCG with state-of-the-art variants of PCG in Table I. The #Allr column shows the number of allreduces per s iterations for every method. The Time column shows the time taken per s iterations for global allreduce (G), Preconditioner (PC) and Sparse Matrix Vector Product (SPMV). The FLOPS column lists the number of Floating Point Operations (xN) in VMAs and dot products for s iterations. For PsCG and PIPE-PsCG methods, the recurrence LCs can be seen as a series of VMAs and we can calculate the number of flops in these VMAs. The Memory column counts the number of vectors that need to be kept in the memory (excluding x and b).

The PCG method [1] has 3s allreduces per s iterations. It uses blocking allreduces and provides no overlap with useful work. Therefore, the times for the 3s allreduces and s PCs and s SPMVs add up. The PIPECG method [9] has s allreduces per s iterations and overlaps one allreduce with one PC and one SPMV.

Method	**#Allr**	**Time for allreduce (G), Preconditioner (PC) and SPMV operations**	**FLOPS**	**Memory**
PCG	3s	s(3G+PC+SPMV)	12s	4
PIPECG	s	s(max(G, PC+SPMV))	22s	9
PIPELCG	s	max(G,s(PC+SPMV)	6s^2 + $14s$	14
PIPECG3	$\lceil s/2 \rceil$	$\lceil s/2 \rceil$(max(G,2(PC+SPMV)))	90*$\lceil s/2 \rceil$	25
PIPECG-OATI	$\lceil s/2 \rceil$	$\lceil s/2 \rceil$(max(G,2(PC+SPMV)))	80*$\lceil s/2 \rceil$	19
PsCG	1	G+(s+1)(PC+SPMV)	2s^2 + $4s + 2$	2s+2
PIPE-PsCG	1	max(G,s(PC+SPMV)	4s^3 + $12s^2$ + $2s + 5$	4s^2 + $12s$ + 5

TABLE I
Differences between various PCG Methods for s iterations of execution.

The PIPELCG method [12] has s allreduces per s iterations. Due to the inherent nature of the PIPELCG algorithm, it requires an extra PC and SPMV in each iteration to compute preconditioned and unpreconditioned residual norms. The entries in table I for PIPELCG are for natural residual norm. PIPE-PsCG can compute the natural, preconditioned and unpreconditioned residual norms without introducing any extra PCs or SPMVs.

The PIPECG3 method [10] and PIPECG-OATI method [11] execute two iterations of PCG in each of their own iteration. So, s iterations of PCG will mean $\lceil s/2 \rceil$ iterations of PIPECG3 and PIPECG-OATI. In each of their iterations, they overlap the allreduce with 2 PCs and 2 SPMVs. The PsCG method [7] has one allreduce per s iterations and has s+1 PCs and s+1 SPMVs in each iteration. But it uses blocking allreduce and cannot overlap that allreduce with useful work.

PIPE-PsCG has one allreduce per s iterations and uses non-blocking allreduce which helps in overlapping s PCs and s SPMVs with the allreduce. Therefore, the time taken per s iterations is the time taken for allreduce or the time for s PCs and s SPMVs, whichever is larger. The PIPE-PsCG method will give performance improvements over other methods when the time taken for global allreduce (G) is completely overlapped by the s PCs and s SPMVs in the problem. Since G increases as number of cores increase, we expect performance improvements at higher number of cores. We also note that the number of FLOPS in PIPE-PsCG are significantly more than the other methods. We see results related to this in the s sensitivity analysis in Section VI. We also see that the memory required by the PIPE-PsCG method is more than other pipelined variants but we have not seen this being a problem for any of the problems that we have worked on as large systems often have abundant main memory in their nodes. The use of multiple nodes to reduce the computational cost means a large amount of aggregated memory from multiple nodes and thus, we don't run out of memory.

PIPECG, PIPELCG, PIPECG3 and PsCG have been shown to stagnate at higher values of relative residuals than the PCG method. In all the pipelined variants, including PIPE-PsCG, rounding errors are introduced in the residuals due to the introduction of recurrence relations.

VI. Experiments and Results

A. *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 PIPE-PsCG method in the PETSc library [15]. We use cray-mpich version 7.7.2. For the non-blocking collective MPI_Iallreduce to make progress, it is necessary to configure our customised PETSc code with –LIBS=-ldmapps for dynamic linking to the DMAPP library and set MPICH_NEMESIS_ASYNC_PROGRESS to 1 in the job script.

We show the performance of our methods by solving the Poisson differential equation on a regular 3D grid discretized with a 125-point stencil. We also show the performance on matrices from the SuiteSparse Matrix Collection [16] obtained from real world application. The equation $Ax = b$ is solved in the tests. The RHS vector b is initialized to Ax^* where x^* is assigned to a vector of ones. The solution vector x_0 is initialized to a vector of zeroes.

Our methods, PIPE-sCG and PIPE-PsCG, are compared with PCG, PIPECG, PIPECG3, PIPECG-OATI and PsCG (Preconditioned s-step CG) methods available in PETSc. We use Jacobi Preconditioner in all preconditioned variants unless stated otherwise. We present results for strong scaling, sensitivity to s, experiments with different preconditioners and accuracy experiments.

B. *Strong Scaling Experiments and Results*

Figure 1 shows the strong scaling of different methods on a 125pt Poisson problem with 1 million (1M) unknowns on up to 120 nodes (2880 cores). All the methods are run to convergence for a relative tolerance of 10^{-5}. We plot the speedup obtained by each method with respect to PCG on one node (24 cores). Here, s=3 for PsCG, PIPE-sCG and PIPE-PsCG.

We observe from Figure 1 that PCG reaches 11.3x speedup at 40 nodes and then the speedup degrades as the number of nodes increase further because the allreduce time increases as the number of nodes increase and PCG does not overlap it with any computation. For PIPECG, we observe that 14.79x speedup is gained at 40 nodes and then the speedup starts degrading because after 40 nodes, the allreduce times become larger than the time taken for one PC and one SPMV. We see that PIPECG3 and PIPECG-OATI both perform better than PCG and PIPECG reaching 17.77x and 19.76x speedup respectively because they overlap two PCs and two SPMVs with the allreduce and provide optimized VMA implementations. However, their speedups also start degrading from 60 nodes on-wards because after 60 nodes, the allreduce times

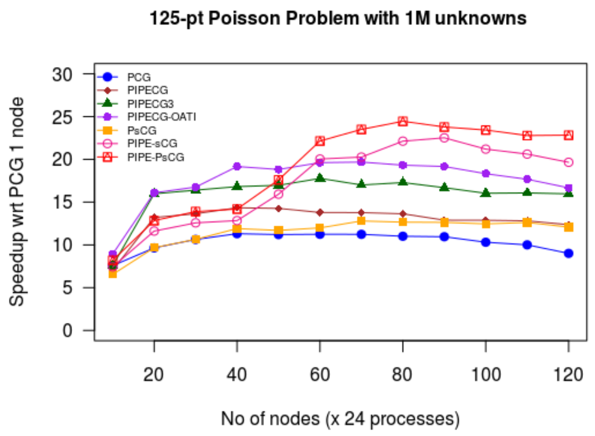

Fig. 1. Strong scaling of different methods on a 125-pt Poisson problem with 1M unknowns on up to 120 nodes.

become larger than the time taken for two PCs and two SPMVs. We see that PsCG performs better than PCG reaching 12.79x speedup because it reduces number of allreduces to one per three iterations. But it always performs worse than all the other methods because it has an extra PC and SPMV in each iteration. We observe that PIPE-PsCG starts performing better than PIPECG from 50 nodes on-wards and better than PIPECG3 and PIPECG-OATI from 60 nodes on-wards because it overlaps more computations with the increased allreduce costs. We observe that PIPE-sCG performs worse than PIPE-PsCG as it takes more iterations to converge due to lack of preconditioning.

In conclusion, for the 125-pt problem with 1M unknowns, PIPE-PsCG provides up to 2.18x speedup wrt PCG, up to 1.84x speedup wrt PIPECG, up to 1.41x speedup wrt PIPECCG3, up to 1.26x speedup wrt PIPECG-OATI and 2x speedup wrt to PsCG. All these speedups are obtained at 80 nodes.

Figure 2 shows the strong scaling of different methods on the ecology2 matrix from the SuiteSparse Matrix Collection [16] on up to 120 nodes. As shown in Table II, it has 1M unknowns. All the methods are run to convergence for a relative tolerance of 10^{-2}. Here, s=3 for PsCG, PIPE-sCG and PIPE-PsCG. For the ecology2 matrix, PIPE-PsCG provides up to 2.9x speedup wrt PCG, up to 2.15x speedup wrt PIPECG, up to 1.4x speedup wrt PIPECG3, up to 1.2x speedup wrt PIPECG-OATI and 2.43x speedup wrt to PsCG. All these speedups are obtained at 120 nodes.

The 2x speedup of our PIPE-PsCG over PsCG for both problems shows that while it is important to reduce the number of allreduces as in PsCG, true performance benefits can be obtained for large number of cores only by reducing the number of SPMVs per iteration and by efficiently overlapping the allreduces with useful computations.

For the ecology2 matrix, the relative tolerance was set to

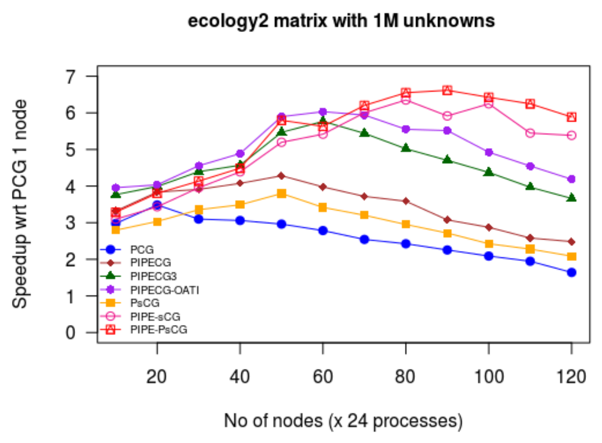

Fig. 2. Strong scaling of different methods on a ecology2 matrix with 1M unknowns on up to 120 nodes.

10^{-2} because the methods PsCG, PIPE-sCG and PIPE-PsCG do not converge for a relative tolerance of 10^{-5}. As discussed in Section V, the residual norms in all these methods stagnate at a higher value. In order to achieve the same lower values of residual norms as PCG method, we combine the PIPE-PsCG method with the method from our previous work, the PIPECG-OATI method. It works as follows: until the residual stagnation starts, PIPE-PsCG method is used to advance the solution of the linear system. As soon as residual stagnation begins, we extract the solution x^* calculated by PIPE-PsCG method and provide it as initial solution to the PIPECG-OATI method. Then, the PIPECG-OATI method advances the solution until the relative tolerance of 10^{-5} is reached. We call this method as Hybrid-pipelined method as it is a hybrid between PIPE-PsCG and PIPECG-OATI methods.

Matrix	**N**	**nnz**	**PCG**	**PIPECG**	**PIPECG-OATI**	**Hybrid-pipelined**
ecology2	999999	4995991	1.52	2.302	3.87	**3.96**
thermal2	1228045	8580313	2.15	3.04	3.52	**4.16**
Serena	1391349	64131971	2.23	4.47	7.15	**8.28**

TABLE II
COMPARISON OF CG METHODS FOR MATRICES FROM SUITESPARSE MATRIX COLLECTION ON 120 NODES. SPEEDUPS ARE SHOWN WRT PCG ON ONE NODE.

The performance of the Hybrid-pipelined method is shown on matrices from the SuiteSparse Matrix collection in Table II. All the methods are run to convergence for relative tolerance of 10^{-5} on 120 nodes. Speedups are shown with respect to PCG on one node. We observe that the most speedup (shown in bold) is provided by Hybrid-pipelined method for different matrices. We observe that the Hybrid-pipelined method provides increased benefits over PIPECG-OATI method when the number of non-zeroes (nnz) are higher (as in case of Serena) because the matrices with more nnz's provide more computations to be overlapped with the allreduce at higher number of cores.

C. *Sensitivity to s*

Figure 3 shows the performance of PIPE-PsCG method for different s values on a 125pt 3D Poisson problem with 1M unknowns on up to 140 nodes (3360 cores). All the methods run to convergence for a relative tolerance of 10^{-5}. We compare the results for three values of s. At s=3, PIPE-PsCG has one allreduce per 3 iterations, at s=4 it has one allreduce per four iterations and at s=5 it has one allreduce per five iterations. We observe that PIPE-PsCG at s=3 performs better

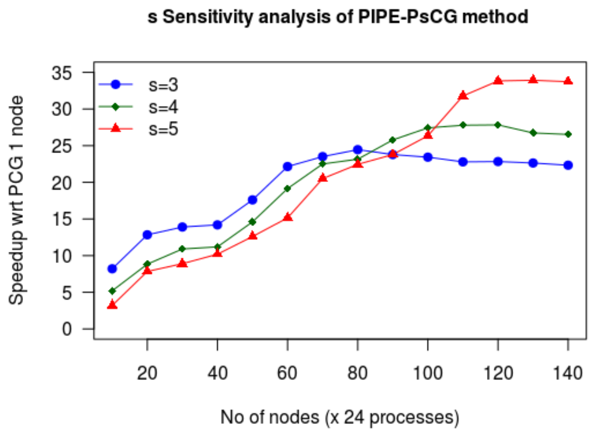

Fig. 3. s Sensitivity analysis of the PIPE-PsCG method for 125pt 1M Poisson problem up to 140 nodes

than s=4 and s=5 till 70 nodes. Similarly, PIPE-PsCG at s=4 performs better than s=5 till 100 nodes. This can be explained as follows. At lower number of cores, the cost of one allreduce is small. Thus, having fewer allreduces in the entire execution and overlapping it with large amount of computations doesn't give significant performance benefits. Additionally, for higher values of s, FLOPS become a significant overhead as shown in Section V. Therefore, the net result is that we end up losing performance. At higher number of cores, the cost of allreduce becomes significant and thus, having fewer allreduces in the entire execution and overlapping it with large amount of computation gives significant performance benefits which is enough to compensate for the overhead introduced by FLOPS. We conclude that PIPE-PsCG with high value of s starts giving performance benefits at higher number of cores.

D. *Preconditioners*

In this section, we present the results of using different preconditioners with the various PCG variants. We use three preconditioners available in PETSc library, SOR (Successive Over Relaxation), MG (Multigrid) and GAMG (Geometric-Algebraic Multigrid). These preconditioners reduce the number of iterations needed to reach convergence. **Figure 4** shows the performance of various PCG variants for these preconditioners on the 125pt 1M Poisson problem at 120 nodes. All the methods run to convergence for a relative tolerance of 10^{-5}. We plot the speedup obtained by each method with respect to PCG on one node. Here, s=3 for PsCG and PIPE-PsCG.

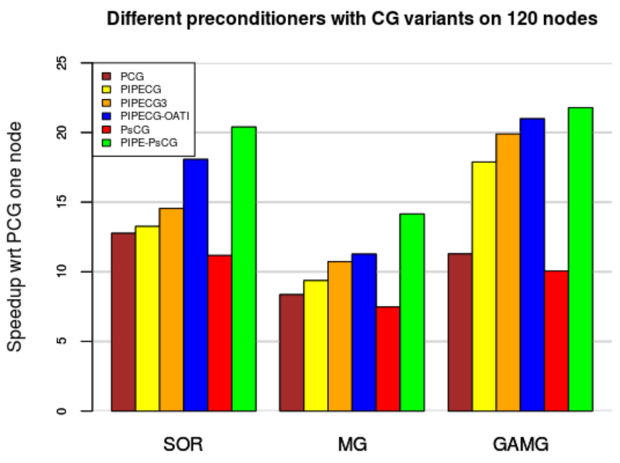

Fig. 4. Different preconditioners on 125pt 1M Poisson problem at 120 nodes

We observe that PIPE-PsCG gives the largest speedup for all preconditioners. PsCG performs worse than even PCG. Despite having fewer allreduces, it has an extra PC and SPMV in every iteration. If we use a simple preconditioner like Jacobi preconditioner, then PsCG is able to overcome the overhead of extra PC and SPMV due to the fewer allreduces and perform better than PCG as shown in figures 1 and 2. However, for advanced preconditioners like SOR, MG, and GAMG, the cost of PC is higher and an extra PC and SPMV provide much overhead and PsCG performs worse than PCG.

We observe that the speedup of PIPE-PsCG with respect to PIPECG-OATI varies for different preconditioners. This depends on the computational intensity of the preconditioner. The SOR and MG are not as computationally intensive as GAMG. Therefore, PIPE-PsCG can provide better overlap of three PCs and three SPMVs with the allreduce. GAMG is computationally expensive and PIPECG-OATI provides the overlap of two PCs and two SPMVS. Compared to this, PIPE-PsCG provides a little additional overlap with the third PC and third SPMV. Hence, the performance improvement is low.

We conclude that we can use different preconditioners with PIPE-PsCG method and get varying performance benefits.

E. Accuracy Experiments and Results

In the CG 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 [17] [18] based applications which solve pressure Poisson equations, we see that default value for $rtol$ is set to 10^{-2}.

Figure 5 shows the relative residual values attained by PCG, PIPECG, PIPECG3, PIPECG-OATI, PsCG and PIPE-PsCG methods as a function of time for the 125-pt 1M unknowns. Here, we see that PIPE-PsCG reaches the threshold of $rtol * \|b\|$ (where $rtol$ is set to 10^{-5}) fastest as compared to all the other methods with PCG being the slowest. 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 with performance benefits.

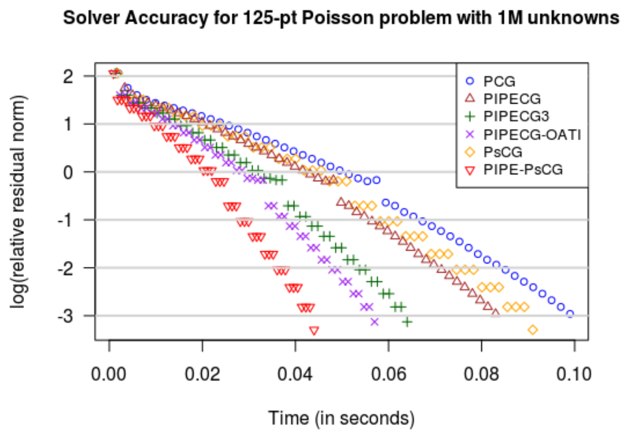

Fig. 5. Solver Accuracy/Performance Experiment for 125-pt 1M Poisson problem. Relative Residual Values as a function of time at 80 nodes.

VII. Conclusion and Future Work

In this work, we developed a novel PIPE-sCG method for distributed memory systems which has one allreduce per s iterations and overlaps it with s SPMVs using MPI_Iallreduce and s can be defined at run time. In this process, we also developed a version of sCG which has s SPMVs as opposed to the s+1 SPMVs in sCG. We also developed a preconditioned version of PIPE-sCG called PIPE-PsCG in which the non-blocking allreduce for dot products is overlapped with s PCs and s SPMVs. An advantage of PIPE-PsCG method is that it can compare the user defined tolerance with any of preconditioned, unpreconditioned or natural norm of the residual without introducing any extra PC or SPMV kernels as opposed to state-of-the-art methods. We have shown that PIPE-PsCG gives performance benefits over other PCG variants at high number of cores when the allreduce cost becomes large and can be completely overlapped by the s PCs and s SPMVs. Our method PIPEC-PsCG gives up to 2.9x speedup wrt PCG, 2.15x speedup wrt PIPECG, 1.4x speedup wrt to PIPECG3, 1.2x speedup wrt PIPECG-OATI and 2.43x speedup wrt PsCG at higher number of cores.

In the future, we plan to automate the process of choosing the s parameter for the PIPE-PsCG method. We plan to devise a model which would give the optimum s value when the linear system dimensions, the number of cores on which we want to solve the linear system and the desired accuracy are given to it as input. In this way, the user doesn't have to worry about choosing the optimum s.

REFERENCES

[1] M. R. Hestenes and E. Stiefel, "Methods of conjugate gradients for solving linear systems," *Journal of research of the National Bureau of Standards*, vol. 49, pp. 409–436, 1952.

[2] Y. Saad, *Iterative Methods for Sparse Linear Systems*, 2nd ed. USA: Society for Industrial and Applied Mathematics, 2003.

[3] ——, "Practical use of some krylov subspace methods for solving indefinite and nonsymmetric linear systems," *SIAM Journal on Scientific and Statistical Computing*, vol. 5, no. 1, pp. 203–228, 1984. [Online]. Available: https://doi.org/10.1137/0905015

[4] G. Meurant, "Multitasking the conjugate gradient method on the cray x-mp/48," *Parallel Computing*, vol. 5, no. 3, pp. 267–280, 1987. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0167819187900378

[5] E. D"Azevedo, V. Eijkhout, and C. Romine, "Lapack working note 56: Reducing communication costs in the conjugate gradient algorithm on distributed memory multiprocessors," University of Tennessee, USA, Tech. Rep., 1993.

[6] A. T. Chronopoulos and C. W. Gear, "S-step iterative methods for symmetric linear systems," *J. Comput. Appl. Math.*, vol. 25, no. 2, p. 153–168, Feb. 1989. [Online]. Available: https://doi.org/10.1016/0377-0427(89)90045-9

[7] A. Chronopoulos and C. Gear, "On the efficient implementation of preconditioned s-step conjugate gradient methods on multiprocessors with memory hierarchy," *Parallel Computing*, vol. 11, no. 1, pp. 37–53, 1989. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0167819189900628

[8] "Mpich 3.3.3," 2019. [Online]. Available: https://www.mpich.org/

[9] P. Ghysels and W. Vanroose, "Hiding global synchronization latency in the preconditioned conjugate gradient algorithm," *Parallel Comput.*, vol. 40, no. 7, p. 224–238, Jul. 2014. [Online]. Available: https://doi.org/10.1016/j.parco.2013.06.001

[10] P. R. Eller and W. Gropp, "Scalable non-blocking preconditioned conjugate gradient methods," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '16. IEEE Press, 2016, pp. 204–215.

[11] M. Tiwari and 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. [Online]. Available: https://doi.org/10.1109/HiPC50609.2020.00029

[12] J. Cornelis, S. Cools, and W. Vanroose, "The communication-hiding conjugate gradient method with deep pipelines," *arXiv.org*, 2018.

[13] S. Cools, J. Cornelis, and W. Vanroose, "Numerically stable recurrence relations for the communication hiding pipelined conjugate gradient method," *arXiv.org*, 2019.

[14] M. Hoemmen, "Communication-avoiding krylov subspace methods," Ph.D. dissertation, University of California at Berkeley, USA, 2010.

[15] 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, and H. Zhang, "PETSc Web page," https://www.mcs.anl.gov/petsc, 2019. [Online]. Available: https://www.mcs.anl.gov/petsc

[16] "Suitesparse matrix collection." [Online]. Available: https://sparse.tamu.edu/

[17] "Openfoam," 2019. [Online]. Available: https://www.openfoam.org/

[18] "Openfoam," 2020. [Online]. Available: https://cfd.direct/openfoam/user-guide/v6-fvsolution/