2020 IEEE 27th International Conference on High Performance Computing, Data, and Analytics (HiPC) | 978-1-6654-2292-5/20/$31.00 ©2020 IEEE | DOI: 10.1109/HIPC50609.2020.00029

Pipelined Preconditioned 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 has been one of the widely used methods for solving linear systems of equations for sparse problems. Pipelined PCG (PIPECG) attempts to eliminate the dependencies in the computations in the PCG algorithm and overlap non-dependent computations by reorganizing the traditional PCG code and using non-blocking allreduces. We have developed a novel pipelined PCG algorithm called PIPECG-OATI (One Allreduce per Two Iterations) that provides large overlap of global communication and computations at higher number of cores in distributed memory CPU systems. Our method achieves this overlapping by using iteration combination and by introducing new non-recurrence computations. We compare our method with other pipelined CG methods on a variety of problems and demonstrate that our method always gives the least runtimes. Our method gives up to 3x speedup over PCG method and 1.73x speedup over PIPECG method at large number of cores.**

***Index Terms*—Preconditioned Conjugate Gradient, distributed memory systems, pipelining, overlapping communication and computations**

I. Introduction

Many High Performance Computing applications in Computational Fluid Dynamics, Electromagnetics, Finance etc. need to solve Partial Differential Equations over space and time. These partial differential equations are discretized using finite volume, finite element or finite difference methods and they result in a linear system of equations $Ax = b$. The A matrix obtained by using these discretization schemes is generally sparse. Iterative methods are used to solve these linear system of equations with sparse matrices. In iterative methods, we begin with an initial guess x_0 and iterate till we get the correct solution. 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) [1] [2] method is one of the widely used Krylov Subspace methods and is used to find the solution of linear systems with symmetric sparse positive definite matrices. In exact arithmetic, it 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.

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 need allreduce operations in distributed memory systems.

For distributed memory systems, the bottleneck in PCG is the synchronization that happens across all cores due to the allreduce operations in the algorithm. Hence, existing research has worked on reducing the number of allreduces to one per iteration as opposed to the three that exists in the naïve algorithm [2] [3] [4].

With the advent of non-blocking collectives like MPI_IAllreduce in the MPI-3 standard [5], the overlapping of allreduce with useful work has been made possible. Gropp's Asynchronous PCG [6] was the first work to use non blocking collectives to overlap allreduces with computation. They used the naïve 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 [7], on which this work is based, uses Chronopoulos-Gear PCG [4] which has one allreduce per iteration. By introducing extra VMAs, they overlap this allreduce with an SPMV and a PC. Pipelined PCG with deeper pipelines (PIPELCG) was introduced in [8] [9] in order to overlap more work with allreduce at higher core counts. They start with Generalised Minimal Residual (GMRES) as the base algorithm. Another variant of CG which has only one allreduce per iteration has three-term recurrence relations. However, this has been shown to have low accuracy than the naïve three two-term recurrence PCG variants. A pipelined version of this variant (PIPECG3) was proposed by Eller et al [10]. It overlaps the single allreduce with two SPMVs and two PCs.

In this work, we propose a novel pipelined PCG method for distributed memory systems, called PIPECG-OATI (PIPECG-One Allreduce per Two Iterations), using iteration combination and new non-recurrence computations. We start with PIPECG and reduce the number of allreduces to one per two iterations and overlap this allreduce with two PCs and SPMVs. Compared to PCG which has three allreduces per iteration and

978-1-6654-2292-5/20/$31.00 ©2020 IEEE
DOI 10.1109/HiPC50609.2020.00029

Authorized licensed use limited to: J.R.D. Tata Memorial Library Indian Institute of Science Bengaluru. Downloaded on December 23,2022 at 14:29:55 UTC from IEEE Xplore. Restrictions apply.

uses blocking allreduces, PIPECG-OATI has one allreduce per two iterations and uses non blocking allreduce. PIPECG-OATI provides upto 3x speedup over PCG. Compared to PIPECG which overlaps one allreduce with one SPMV and one PC, PIPECG-OATI overlaps the allreduce with two SPMVs and two PCs so that as number of cores increase, the increasing latency of allreduce can be hidden by more overlapped work. PIPECG-OATI provides 1.73x speedup over PIPECG.

The rest of the paper is organized as follows: Section II gives background related to PCG and PIPECG, Section III describes our algorithm for PIPECG-OATI method using iteration combination. Section IV presents the analysis and comparison of our method compared to other works. Section V presents the optimizations we have implemented in our method. Section VI presents experiments, results and discussions for our proposed method and Section VII gives the conclusions and future work.

II. Background

PCG: The Preconditioned Conjugate Gradient Method (PCG) introduced by Hestenes and Stiefel [1] is given in Algorithm 1. As shown in Algorithm 1, the computational

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**

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. Note that the third Dot Product (line 17) is for calculating the residual norm which is used to check for convergence. 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 processors to synchronize and send their local dot products so that the global Dot Product can be calculated. In the naïve 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.

PIPECG: The Pipelined Peconditioned Conjugate Gradient Method (PIPECG) was proposed by Ghysels and Vanroose [7] for obtaining performance improvements on distributed memory architectures. As shown in Algorithm 2, PIPECG

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**

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 and the cores don't have to be left idle. The MPI_Iallreduce (line 21) for the Dot Products (line 18, 19, 20) can be overlapped with the PC and SPMV (line 22, 23).

The PIPECG method overlaps a single MPI_Iallreduce with one SPMV and one PC. 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 SPMV and PC. In order

Authorized licensed use limited to: J.R.D. Tata Memorial Library Indian Institute of Science Bengaluru. Downloaded on December 23,2022 at 14:29:55 UTC from IEEE Xplore. Restrictions apply.

to hide the latency introduced by MPI_Iallreduce at higher number of cores, we must overlap it with more work.

III. Methodology

We propose a novel algorithm, 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 SPMVs and two PCs. 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.

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**

In Algorithm 3, we unroll two iterations of PIPECG. For achieving PIPECG-OATI from Algorithm 3, we follow the below steps:

1) Move the PC and SPMV (line 15) after the PC and SPMV of the previous iteration (line 23) by introducing recurrence relations for them.
2) Express the dot products (line 13) as recurrence relations and move the resulting new dot products after the previous iteration's dot products (line 21).
3) As the new dot products will need results of PC and SPMV beforehand, introduce recurrence relations for the PC and SPMV.
4) To deal with extra PC and SPMV, introduce new non-recurrence computations.

Step 1: In order to move the first PC and SPMV (line 15) immediately after the previous iteration's PC and SPMV (line 23), we substitute $w_{i+1} = w_i - \alpha_i z_i$ into $m_{i+1} = M^{-1}w_{i+1}$. So we get:

$m_{i+1} = m_i - \alpha_i c_i$ where $c_i = M^{-1}z_i$.

Substituting m_{i+1} into $n_{i+1} = Am_{i+1}$ we get,

$n_{i+1} = n_i - \alpha_i d_i$ where $d_i = Ac_i$.

Now, introducing recurrence relations for c_i and d_i in a similar way-

$c_i = g_i + \beta_i c_{i-1}$ where $g_i = M^{-1}n_i$.

$d_i = h_i + \beta_i d_{i-1}$ where $h_i = Ag_i$.

Thus, g_i and h_i can be moved to the previous iteration on line 25 and we get Algorithm 4. The recurrence relations added in this step are shown in red.

Algorithm 4 PIPECG-OATI: After Step 1

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: $m_0 = M^{-1}w_0$; $n_0 = Am_0$; $g_0 = M^{-1}n_0$; $h_0 = Ag_0$
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: $p_i = u_i + \beta_i p_{i-1}$; $c_i = g_i + \beta_i c_{i-1}$; $d_i = h_i + \beta_i d_{i-1}$;
12: $x_{i+1} = x_i + \alpha_i p_i$; $r_{i+1} = r_i - \alpha_i s_i$
13: $u_{i+1} = u_i - \alpha_i q_i$; $w_{i+1} = w_i - \alpha_i z_i$
14: $\gamma_{i+1} = (r_{i+1}, u_{i+1})$; $\delta_{i+1} = (w_{i+1}, u_{i+1})$; $norm_{i+1} = \sqrt{(u_{i+1}, u_{i+1})}$
15: MPI_Iallreduce on $\gamma_{i+1}, \delta_{i+1}, norm_{i+1}$
16: $m_{i+1} = m_i - \alpha_i c_i$; $n_{i+1} = n_i - \alpha_i d_i$
17: $\beta_{i+1} = \gamma_{i+1}/\gamma_i$; $\alpha_{i+1} = \gamma_{i+1}/(\delta_{i+1} - \beta_{i+1}\gamma_{i+1}/\alpha_i)$;
18: $z_{i+1} = n_{i+1} + \beta_{i+1} z_i$; $q_{i+1} = m_{i+1} + \beta_{i+1} q_i$
19: $s_{i+1} = w_{i+1} + \beta_{i+1} s_i$; $p_{i+1} = u_{i+1} + \beta_{i+1} p_i$
20: $x_{i+2} = x_{i+1} + \alpha_{i+1} p_{i+1}$; $r_{i+2} = r_{i+1} - \alpha_{i+1} s_{i+1}$
21: $u_{i+2} = u_{i+1} - \alpha_{i+1} q_{i+1}$; $w_{i+2} = w_{i+1} - \alpha_{i+1} z_{i+1}$
22: $\gamma_{i+2} = (r_{i+2}, u_{i+2})$; $\delta_{i+2} = (w_{i+2}, u_{i+2})$; $norm_{i+2} = \sqrt{(u_{i+2}, u_{i+2})}$
23: MPI_Iallreduce on $\gamma_{i+2}, \delta_{i+2}, norm_{i+2}$
24: $m_{i+2} = M^{-1}w_{i+2}$; $n_{i+2} = Am_{i+2}$
25: $g_{i+2} = M^{-1}n_{i+2}$; $h_{i+2} = g_{i+2}$
26: **end for**

Step 2: We need to move the dot products (line 14) to the previous iteration in a way such that we obtain one allreduce per two iterations. In order to do this, we substitute the full

Authorized licensed use limited to: J.R.D. Tata Memorial Library Indian Institute of Science Bengaluru. Downloaded on December 23,2022 at 14:29:55 UTC from IEEE Xplore. Restrictions apply.

equations for r_{i+1}, u_{i+1} and w_{i+1} into the dot products like the following-

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

Further, we can use the transformation $q = M^{-1}s$ to obtain-

$$(r_i, q_i) = (r_i, M^{-1}s_i) = (M^{-1}r_i, s_i) = (u_i, s_i) = (s_i, u_i)$$

This reduces the number of dot products to be calculated as we obtain-

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

Expressing δ_{i+1}, (s_i, u_i) and (s_i, q_i) in similar ways and using transformations to reduce the number of dot products, we obtain the new dot products that we need to calculate. Combined with the dot products of the previous iteration, we need to calculate a total of ten dot products stored in an array λ, all of which can be computed using one allreduce. We express γ (line 11) and δ (line 12) as a function of the elements of array λ and we get Algorithm 5. Here we observe that some elements of λ require m_{i+2} and n_{i+2} which have been moved to line 23. Now the MPI_IAllreduce on line 29 only overlaps one PC and one SPMV on line 30.

Step 3: In order to get values of m_{i+2} and n_{i+2} before the λ are calculated, we use these recurrence relations-

$$c_{i+1} = g_{i+1} + \beta_{i+1}c_i$$
$$d_{i+1} = h_{i+1} + \beta_{i+1}d_i$$
$$m_{i+2} = m_{i+1} - \alpha_{i+1}c_{i+1}$$
$$n_{i+2} = n_{i+1} - \alpha_{i+1}d_{i+1}$$

For calculating g_{i+1} and h_{i+1}, we introduce the following recurrence relations-

$$a_i = e_i + \beta_i a_{i-1}$$
$$b_i = f_i + \beta_i b_{i-1}$$
$$g_{i+1} = g_i - \alpha_i a_i$$
$$h_{i+1} = h_i - \alpha_i b_i$$

where $a_i = M^{-1}d_i$, $b_i = Aa_i$, $e_i = M^{-1}h_i$ and $f_i = Ae_i$. After Step 3, we obtain Algorithm 6. Here, we can see that each iteration then requires three PCs and SPMVs. $a_{i+1} = M^{-1}d_{i+1}$ is the extra PC and $b_{i+1} = Aa_{i+1}$ is the extra SPMV introduced by iteration combination.

Step 4: In order to calculate $a_{i+1} = M^{-1}d_{i+1}$ (line 35, Algorithm 6) and $b_{i+1} = Aa_{i+1}$ (line 36, Algorithm 6), we introduce the following non-recurrence computations:

$$a_{i+1} = (g_{i+1} - g_{i+2})/\alpha_{i+1}$$
$$b_{i+1} = (h_{i+1} - h_{i+2})/\alpha_{i+1}$$

These computations remove the need for an extra PC and SPMV by storing two extra vectors g_{i+1} and h_{i+1}.

Algorithm 7 captures all the described reorganizations, 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 8) which can be computed with a single allreduce.

Comparing our PIPECG-OATI algorithm of Algorithm 7 with the PIPECG algorithm of Algorithm 2, we see that we are able to reduce the number of allreduces to one per two iterations and overlap this allreduce with two PCs and SPMVs without introducing any more PC and SPMV, because of the new non-recurrence computations we introduced.

Algorithm 5 PIPECG-OATI: After Step 2

1: $r_0 = b - Ax_0$; $u_0 = M^{-1}r_0$; $w_0 = Au_0$;
2: $\gamma_0 = \lambda_7 = (r_0, u_0)$; $\delta_0 = \lambda_8 = (w_0, u_0)$; $norm_0 = \lambda_9 = \sqrt{(u_0, u_0)}$
3: $m_0 = M^{-1}w_0$; $n_0 = Am_0$; $g_0 = M^{-1}n_0$; $h_0 = Ag_0$
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: $z_i = n_i + \beta_i z_{i-1}$; $q_i = m_i + \beta_i q_{i-1}$; $s_i = w_i + \beta_i s_{i-1}$
15: $p_i = u_i + \beta_i p_{i-1}$; $c_i = g_i + \beta_i c_{i-1}$; $d_i = h_i + \beta_i d_{i-1}$;
16: $x_{i+1} = x_i + \alpha_i p_i$; $r_{i+1} = r_i - \alpha_i s_i$
17: $u_{i+1} = u_i - \alpha_i q_i$; $w_{i+1} = w_i - \alpha_i z_i$
18: $m_{i+1} = m_i - \alpha_i c_i$; $n_{i+1} = n_i - \alpha_i d_i$
19: $z_{i+1} = n_{i+1} + \beta_{i+1} z_i$; $q_{i+1} = m_{i+1} + \beta_{i+1} q_i$;
20: $s_{i+1} = w_{i+1} + \beta_{i+1} s_i$ $p_{i+1} = u_{i+1} + \beta_{i+1} p_i$;
21: $x_{i+2} = x_{i+1} + \alpha_{i+1} p_{i+1}$; $r_{i+2} = r_{i+1} - \alpha_{i+1} s_{i+1}$
22: $u_{i+2} = u_{i+1} - \alpha_{i+1} q_{i+1}$; $w_{i+2} = w_{i+1} - \alpha_{i+1} z_{i+1}$
23: $m_{i+2} = M^{-1}w_{i+2}$; $n_{i+2} = Am_{i+2}$
24: $\lambda_0 = (u_{i+2}, s_{i+1})$; $\lambda_1 = (w_{i+2}, m_{i+2})$;
25: $\lambda_2 = (w_{i+2}, q_{i+1})$; $\lambda_3 = (s_{i+1}, q_{i+1})$;
26: $\lambda_4 = (n_{i+2}, m_{i+2})$; $\lambda_5 = (n_{i+2}, q_{i+1})$;
27: $\lambda_6 = (z_{i+1}, q_{i+1})$; $\lambda_7 = (r_{i+2}, u_{i+2})$;
28: $\lambda_8 = (u_{i+2}, w_{i+2})$; $\lambda_9 = (u_{i+2}, u_{i+2})$;
29: MPI_Iallreduce on λ_0 to λ_9
30: $g_{i+2} = M^{-1}n_{i+2}$; $h_{i+2} = g_{i+2}$
31: $\gamma_{i+2} = \lambda_7; \delta_{i+2} = \lambda_8; norm_{i+2} = \sqrt{\lambda_9}$
32: **end for**

IV. Analysis and Comparison with Different Methods

In this section, we analyse and compare state-of-the-art variants of PCG as shown in Table 1. The # Allr column shows the number of allreduces per two iterations for every method. The Time column shows the time taken per two 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 two iterations. 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 six allreduces per two iterations. It uses blocking allreduces and provides no overlap with useful work. Therefore, the times for the allreduce and PC and SPMV add up. PIPECG-OATI has one allreduce per two iterations

Authorized licensed use limited to: J.R.D. Tata Memorial Library Indian Institute of Science Bengaluru. Downloaded on December 23,2022 at 14:29:55 UTC from IEEE Xplore. Restrictions apply.

Algorithm 6 PIPECG-OATI: After Step 3

$r_0 = b - Ax_0$; $u_0 = M^{-1}r_0$; $w_0 = Au_0$;
$\gamma_0 = \lambda_7 = (r_0, u_0)$; $\delta_0 = \lambda_8 = (w_0, u_0)$; $norm_0 = \lambda_9 = \sqrt{(u_0, u_0)}$
$m_0 = M^{-1}w_0$; $n_0 = Am_0$; $g_0 = M^{-1}n_0$; $h_0 = Ag_0$; $e_0 = M^{-1}h_0$; $f_0 = Ae_0$
$\lambda_1 = (w_0, m_0)$; $\lambda_4 = (n_0, m_0)$
for i=0,2,4,... **do**
  if $i > 0$ **then**
    $\beta_i = \gamma_i/\gamma_{i-1}$; $\alpha_i = \gamma_i/(\delta_i - \beta_i\gamma_i/\alpha_{i-1})$;
  else
    $\beta_i = 0; \alpha_i = \gamma_i/\delta_i$
  end if
  $\gamma_{i+1} = \lambda_7 - 2\alpha_i(\lambda_8 + \beta_i\lambda_0) + \alpha_i{}^2(\lambda_1 + 2*\beta_i\lambda_2 + \beta_i{}^2\lambda_3)$
  $\delta_{i+1} = \lambda_8 - \alpha_i(\lambda_1 + \beta_i\lambda_2) - \alpha_i(\lambda_1 + \beta_i\lambda_2) + \alpha_i{}^2(\lambda_4 + \beta_i\lambda_5 + \beta_i\lambda_5) + \beta_i{}^2\lambda_6$
  $\beta_{i+1} = \gamma_{i+1}/\gamma_i$; $\alpha_{i+1} = \gamma_{i+1}/(\delta_{i+1} - \beta_{i+1}\gamma_{i+1}/\alpha_i)$;
  $z_i = n_i + \beta_i z_{i-1}$; $q_i = m_i + \beta_i q_{i-1}$; $s_i = w_i + \beta_i s_{i-1}$
  $p_i = u_i + \beta_i p_{i-1}$; $c_i = g_i + \beta_i c_{i-1}$; $d_i = h_i + \beta_i d_{i-1}$;
  $a_i = e_i + \beta_i a_{i-1}$; $b_i = f_i + \beta_i b_{i-1}$;
  $x_{i+1} = x_i + \alpha_i p_i$; $r_{i+1} = r_i - \alpha_i s_i$
  $u_{i+1} = u_i - \alpha_i q_i$; $w_{i+1} = w_i - \alpha_i z_i$
  $m_{i+1} = m_i - \alpha_i c_i$; $n_{i+1} = n_i - \alpha_i d_i$
  $g_{i+1} = g_i - \alpha_i a_i$; $h_{i+1} = h_i - \alpha_i b_i$
  $z_{i+1} = n_{i+1} + \beta_{i+1} z_i$; $q_{i+1} = m_{i+1} + \beta_{i+1} q_i$;
  $s_{i+1} = w_{i+1} + \beta_{i+1} s_i$ $p_{i+1} = u_{i+1} + \beta_{i+1} p_i$;
  $c_{i+1} = g_{i+1} + \beta_{i+1} c_i$; $d_{i+1} = h_{i+1} + \beta_{i+1} d_i$;
  $x_{i+2} = x_{i+1} + \alpha_{i+1} p_{i+1}$; $r_{i+2} = r_{i+1} - \alpha_{i+1} s_{i+1}$
  $u_{i+2} = u_{i+1} - \alpha_{i+1} q_{i+1}$; $w_{i+2} = w_{i+1} - \alpha_{i+1} z_{i+1}$
  $m_{i+2} = m_{i+1} - \alpha_{i+1} c_{i+1}$; $n_{i+2} = n_{i+1} - \alpha_{i+1} d_{i+1}$
  $\lambda_0 = (u_{i+2}, s_{i+1})$; $\lambda_1 = (w_{i+2}, m_{i+2})$;
  $\lambda_2 = (w_{i+2}, q_{i+1})$; $\lambda_3 = (s_{i+1}, q_{i+1})$;
  $\lambda_4 = (n_{i+2}, m_{i+2})$; $\lambda_5 = (n_{i+2}, q_{i+1})$;
  $\lambda_6 = (z_{i+1}, q_{i+1})$; $\lambda_7 = (r_{i+2}, u_{i+2})$;
  $\lambda_8 = (u_{i+2}, w_{i+2})$; $\lambda_9 = (u_{i+2}, u_{i+2})$;
  MPI_Iallreduce on λ_0 to λ_9
  $g_{i+2} = M^{-1}n_{i+2}$; $h_{i+2} = g_{i+2}$
  $e_{i+2} = M^{-1}h_{i+2}$; $f_{i+2} = Ae_{i+2}$
  $a_{i+1} = M^{-1}d_{i+1}$
  $b_{i+1} = Aa_{i+1}$
  $\gamma_{i+2} = \lambda_7; \delta_{i+2} = \lambda_8; norm_{i+2} = \sqrt{\lambda_9}$
end for

Method	# Allr	Time for allreduce (G), Preconditioner (PC) and SPMV operations	FLOPS	Memory
PCG	6	6G+2PC+2SPMV	24	4
PIPECG	2	max(2G, 2PC+2SPMV)	44	9
PIPELCG	2	max(G,2PC+2SPMV)	52	14
PIPECG3	1	max(G, 2PC+2SPMV)	90	25
PIPECG-OATI	1	max(G, 2PC+2SPMV)	80	19

TABLE I
DIFFERENCES BETWEEN VARIOUS PCG METHODS FOR TWO ITERATIONS OF EXECUTION. L=2 FOR PIPELCG.

Algorithm 7 PIPECG-OATI (PIPECG-One Allreduce per Two Iterations): After Step 4

$r_0 = b - Ax_0$; $u_0 = M^{-1}r_0$; $w_0 = Au_0$;
$\gamma_0 = \lambda_7 = (r_0, u_0)$; $\delta_0 = \lambda_8 = (w_0, u_0)$; $norm_0 = \lambda_9 = \sqrt{(u_0, u_0)}$
$m_0 = M^{-1}w_0$; $n_0 = Am_0$; $g_0 = M^{-1}n_0$; $h_0 = Ag_0$; $e_0 = M^{-1}h_0$; $f_0 = Ae_0$
$\lambda_1 = (w_0, m_0)$; $\lambda_4 = (n_0, m_0)$
for i=0,2,4,... **do**
  if $i > 0$ **then**
    $\beta_i = \gamma_i/\gamma_{i-1}$; $\alpha_i = \gamma_i/(\delta_i - \beta_i\gamma_i/\alpha_{i-1})$;
  else
    $\beta_i = 0; \alpha_i = \gamma_i/\delta$
  end if
  $\gamma_{i+1} = \lambda_7 - 2\alpha_i(\lambda_8 + \beta_i\lambda_0) + \alpha_i{}^2(\lambda_1 + 2*\beta_i\lambda_2 + \beta_i{}^2\lambda_3)$
  $\delta_{i+1} = \lambda_8 - \alpha_i(\lambda_1 + \beta_i\lambda_2) - \alpha_i(\lambda_1 + \beta_i\lambda_2) + \alpha_i{}^2(\lambda_4 + \beta_i\lambda_5 + \beta_i\lambda_5) + \beta_i{}^2\lambda_6$
  $\beta_{i+1} = \gamma_{i+1}/\gamma_i$; $\alpha_{i+1} = \gamma_{i+1}/(\delta_{i+1} - \beta_{i+1}\gamma_{i+1}/\alpha_i)$;
  VecOps
  MPI_Iallreduce on $\lambda_0, \lambda_1 ... \lambda_9$
  $g_{i+2} = M^{-1}n_{i+2}$; $h_{i+2} = Ag_{i+2}$
  $e_{i+2} = M^{-1}h_{i+2}$; $f_{i+2} = Ae_{i+2}$
  $a_{i+1} = (g_{i+1} - g_{i+2})/\alpha_{i+1}$
  $b_{i+1} = (h_{i+1} - h_{i+2})/\alpha_{i+1}$
  $\gamma_{i+2} = \lambda_7$; $\delta_{i+2} = \lambda_8$; $norm_{i+2} = \sqrt{\lambda_9}$
end for

Algorithm 8 VecOps

$z_i = n_i + \beta_i z_{i-1}$; $q_i = m_i + \beta_i q_{i-1}$; $s_i = w_i + \beta_i s_{i-1}$
$p_i = u_i + \beta_i p_{i-1}$; $c_i = g_i + \beta_i c_{i-1}$; $d_i = h_i + \beta_i d_{i-1}$;
$a_i = e_i + \beta_i a_{i-1}$; $b_i = f_i + \beta_i b_{i-1}$;
$x_{i+1} = x_i + \alpha_i p_i$; $r_{i+1} = r_i - \alpha_i s_i$
$u_{i+1} = u_i - \alpha_i q_i$; $w_{i+1} = w_i - \alpha_i z_i$
$m_{i+1} = m_i - \alpha_i c_i$; $n_{i+1} = n_i - \alpha_i d_i$
$g_{i+1} = g_i - \alpha_i a_i$; $h_{i+1} = h_i - \alpha_i b_i$
$z_{i+1} = n_{i+1} + \beta_{i+1} z_i$; $q_{i+1} = m_{i+1} + \beta_{i+1} q_i$;
$s_{i+1} = w_{i+1} + \beta_{i+1} s_i$ $p_{i+1} = u_{i+1} + \beta_{i+1} p_i$;
$c_{i+1} = g_{i+1} + \beta_{i+1} c_i$; $d_{i+1} = h_{i+1} + \beta_{i+1} d_i$;
$x_{i+2} = x_{i+1} + \alpha_{i+1} p_{i+1}$; $r_{i+2} = r_{i+1} - \alpha_{i+1} s_{i+1}$
$u_{i+2} = u_{i+1} - \alpha_{i+1} q_{i+1}$; $w_{i+2} = w_{i+1} - \alpha_{i+1} z_{i+1}$
$m_{i+2} = m_{i+1} - \alpha_{i+1} c_{i+1}$; $n_{i+2} = n_{i+1} - \alpha_{i+1} d_{i+1}$
$\lambda_0 = (u_{i+2}, s_{i+1})$; $\lambda_1 = (w_{i+2}, m_{i+2})$;
$\lambda_2 = (w_{i+2}, q_{i+1})$; $\lambda_3 = (s_{i+1}, q_{i+1})$;
$\lambda_4 = (n_{i+2}, m_{i+2})$; $\lambda_5 = (n_{i+2}, q_{i+1})$;
$\lambda_6 = (z_{i+1}, q_{i+1})$; $\lambda_7 = (r_{i+2}, u_{i+2})$;
$\lambda_8 = (u_{i+2}, w_{i+2})$; $\lambda_9 = (u_{i+2}, u_{i+2})$;

Authorized licensed use limited to: J.R.D. Tata Memorial Library Indian Institute of Science Bengaluru. Downloaded on December 23,2022 at 14:29:55 UTC from IEEE Xplore. Restrictions apply.

and uses non-blocking allreduce which helps in overlapping PC and SPMV with allreduce. Therefore, the time taken per two iterations is the time taken for allreduce or the time for PC and SPMV, whichever is larger.

The PIPECG method [7] has two allreduces per two iterations and overlaps one allreduce with one SPMV and one PC. PIPECG-OATI overlaps one allreduce with two PCs and two SPMVs so that as number of cores increase, the increasing times of allreduce can be overlapped with more work.

The PIPELCG method [8] has two allreduces per two iterations, while PIPECG-OATI has one allreduce per two iterations. PIPELCG uses GMRES as the base algorithm and hence is prone to restarts whereas PIPECG-OATI starts with PIPECG as the base algorithm and has no restarts. Due to the inherent nature of the PIPELCG algorithm, it cannot compute preconditioned and unpreconditioned residual norms as it requires extra PC and SPMV to compute these norms. PIPECG-OATI allows us to compute preconditioned and unpreconditioned residual norms without the extra PC and SPMV, because of the non-recurrence computations we introduced.

When compared to PIPECG3 method [10], PIPECG-OATI has lower number of FLOPS and lower memory requirements resulting in performance improvements. Also, PIPECG3 uses the three-term recurrence variant of PCG as the base algorithm and hence can provide lower accuracy of solution in certain cases whereas PIPECG-OATI uses PIPECG as base algorithm which provides similar accuracy in all cases.

The PIPECG-OATI method will give performance improvements over other methods when the time taken for global allreduce (G) is completely overlapped by the two PCs and SPMVs in the problem. Since G increases as number of cores increase and the times of PC and SPMV depend on the number of non zeroes in the matrix of the problem, we predict that we gain performance improvements at higher number of cores for heavily computationally intensive problems. Thus, we conclude that PIPECG-OATI has various advantages over state-of-the-art PCG variants. We also note that the number of floating point operations 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.

V. Optimization

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

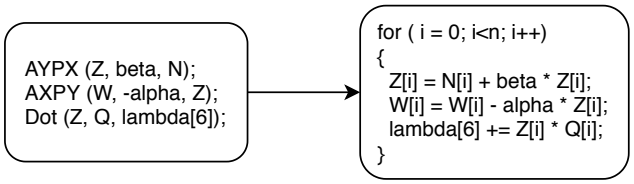

Fig. 1. Merging VecOps

As shown in Figure 1 left, in the naïve implementation of PIPECG-OATI, we call individual AYPX, AXPY and DOT functions to implement the VecOps. However if we merge the VecOps as shown in Figure 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 introduces 21 extra VMAs and 7 new Dot Products as shown in Algorithm 8, 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.

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 PIPECG-OATI method in the PETSc library [11]–[13]. 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. Our implementation for PIPECG-OATI is available in the open-source PETSc repository [1].

We ran tests with 2D 9-point, 3D 27-point and 3D 125-point Poisson Problems with varying number of unknowns.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 method, PIPECG-OATI, is compared with PCG, PIPECG, PIPELCG (with L=2) and PIPECG3 methods available in PETSc. We use Jacobi Preconditioner in all tests. Each test was run four times and the average speedup gained is presented. We present results for strong scaling, accuracy and detailed timing breakdown of our method compared with state-of-the-art methods. We also discuss the performance improvements gained due to our optimizations.

B. Strong Scaling Experiments and Results

Figure 2 shows the strong scaling of different methods on a 9pt 2D Poisson problem with 1 million (1M) unknowns on up to 70 nodes (1680 cores). All the methods run to convergence for an absolute tolerance of 10^{-4} in 1120 iterations. We plot the speedup obtained by each method with respect to PCG on one node (24 cores).

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

Authorized licensed use limited to: J.R.D. Tata Memorial Library Indian Institute of Science Bengaluru. Downloaded on December 23,2022 at 14:29:55 UTC from IEEE Xplore. Restrictions apply.

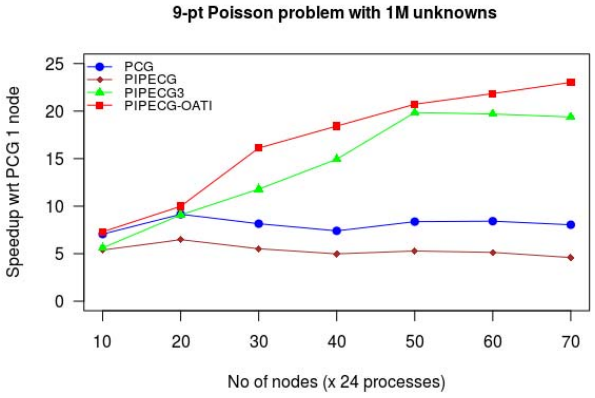

Fig. 2. Strong scaling of different methods on a 9-pt Poisson problem with 1M unknowns.

We observe from Figure 2 that PCG reaches 9x speedup at 20 nodes and then the speedup degrades as the number of nodes increase further. This happens 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 5x speedup is gained but it is always less than PCG despite PIPECG overlapping one allreduce with computations. This happens because the cost of unoptimized VecOps in PIPECG overshadows the allreduce it overlaps with computation. As 9-pt stencil Poisson problem provides less number of non zeroes, hence one PC and one SPMV are not very expensive and not much of the allreduce is overlapped. So, PIPECG does not provide much overlap and in addition introduces unoptimized VecOps. This leads to low performance of PIPECG than the other methods. We see that PIPECG3 and PIPECG-OATI both perform better than PCG and PIPECG reaching 19x and 24x speedup respectively. This happens as they overlap two PCs and two SPMVs with the allreduce. So, even though a single PC and SPMV in 9-pt problem are not very expensive, two PCs and two SPMVs provide considerable work to be overlapped with the allreduce. Furthermore, we see that the PIPECG-OATI provides better speedup than PIPECG3. This is because the number of FLOPS per iteration in PIPECG-OATI are lower than that in PIPECG3 (as discussed in Section IV). Also, the optimized implementation of the VecOps in PIPECG-OATI help in achieving better memory accesses and hence better performance.

So, for the 9-pt problem with 1M unknowns, PIPECG-OATI provides up to 3x speedup wrt PCG (at 70 nodes), up to 6x speedup wrt PIPECG (at 70 nodes) and up to 1.36x speedup wrt PIPECG3 (at 30 nodes).

As discussed in Section IV, PIPELCG method cannot compute the preconditioned norm of the residual whereas all the other methods determine the convergence based on the preconditoned residual norm. As PIPELCG 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 cannot compare the total time of convergence for PIPECG-OATI with PIPELCG. Since the norm type changes only the number of iterations taken by a method to reach to convergence and has no effect on what happens inside the iteration itself, we compare the time taken per two iterations in PIPELCG with the time taken per combined-iteration in PIPECG-OATI and find that PIPECG-OATI gives up to 1.89x speedup over PIPELCG for this problem (at 70 nodes).

Figure 3 shows the strong scaling of different methods on a 9pt 2D Poisson problem with 2M unknowns on up to 110 nodes (2640 processes). All the methods run to convergence for an absolute tolerance of 10^{-3} in 1390 iterations.

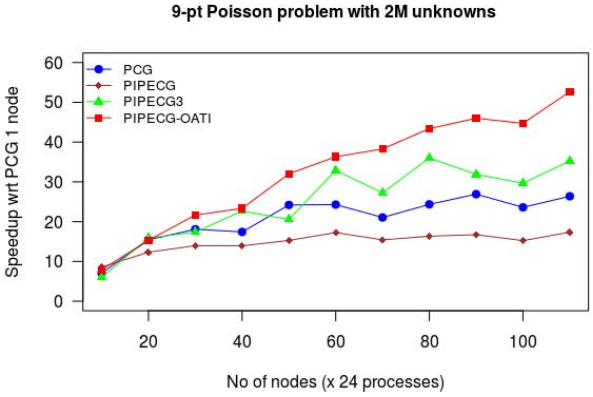

Fig. 3. Strong scaling of different methods on a 9-pt Poisson problem with 2M unknowns.

The trends in the Figure 3 are the same as that for Figure 2 and can be explained by the same reasons. It must be noted that in 2M case, PIPECG-OATI starts showing performance improvement over PCG at 30 nodes as opposed to 20 nodes in the 1M case (Figure 2). This happens because two PCs and two SPMVs for the smaller 1M problem are enough to completely overlap the cost of allreduce at 20 nodes. When we have the larger 2M problem, the two PCs and SPMVs start completely overlapping the cost of allreduce from 30 nodes on wards. We also note that the speedup provided by PIPECG-OATI degrades as we go from 90 to 100 nodes but then increases again for 110 nodes. On finer analysis of multiple runs of the experiment, we found that the allreduce takes a larger time at 100 nodes, even more than 110 nodes. This can also be seen in the performance given by the other methods. Every method's performance degrades at 100 nodes but they perform better at 110 nodes. We understand this is a system specific issue.

For the 9-pt problem with 2M unknowns, PIPECG-OATI provides up to 2x speedup wrt PCG (at 110 nodes), up to 3.05x speedup wrt PIPECG (at 110 nodes) and up to 1.54x speedup wrt PIPECG3 (at 100 nodes). We compared the time per two iterations in PIPELCG with the time per combined-iteration

Authorized licensed use limited to: J.R.D. Tata Memorial Library Indian Institute of Science Bengaluru. Downloaded on December 23,2022 at 14:29:55 UTC from IEEE Xplore. Restrictions apply.

in PIPECG-OATI and found that PIPECG-OATI gives up to 5x speedup over PIPELCG for this problem (at 110 nodes).

Figure 4 shows the strong scaling of different methods on a 27-pt 3D Poisson problem with 2M unknowns on up to 110 nodes (2640 processes). All the methods run to convergence for an absolute tolerance of 10^{-6} in 158 iterations.

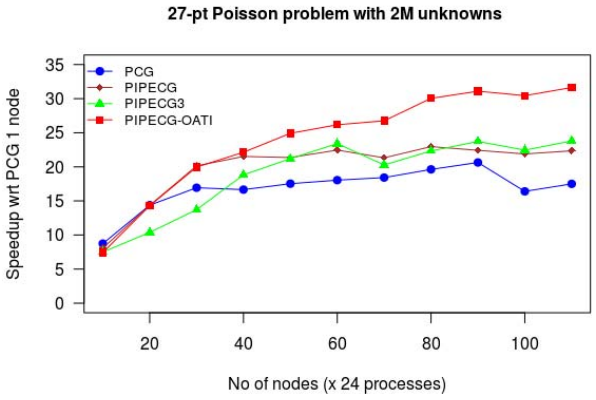

Fig. 4. Strong scaling of different methods on a 27-pt Poisson problem with 2M unknowns.

We observe from Figure 4 that PCG reaches 20x speedup at 90 nodes and then the speedup degrades as the number of nodes increase further. For PIPECG, we observe that 22x speedup is gained. Here PIPECG performs better than PCG because the PC and SPMV in the 27-pt 3D problem are expensive enough to completely overlap the cost of the allreduce. The unoptimized VMAs do produce some overhead but it is overshadowed by the complete overlapping of allreduce with PC and SPMV. We see that PIPECG3 gives almost the same performance as PIPECG and reaches 23x speedup. PIPECG-OATI performs better than PCG and PIPECG reaching 31x speedup. It must be noted here that PIPECG-OATI starts performing better than PIPECG at 50 nodes because the overlap provided by PIPECG-OATI becomes more than the overlap provided by PIPECG.

So, for the 27-pt problem with 2M unknowns, PIPECG-OATI provides up to 1.82x speedup wrt PCG (at 110 nodes), up to 1.4x speedup wrt PIPECG (at 110 nodes) and up to 1.3x speedup wrt PIPECG3 (at 90 nodes). We compared the time per two iterations in PIPELCG with the time per combined-iteration in PIPECG-OATI and found that PIPECG-OATI gives up to 2.34 speedup over PIPELCG for this problem (at 110 nodes).

From this section we conclude that PIPECG-OATI gives varying speedups over all the other methods for different problems. This is because it overlaps two PCs and two SPMVs with a single allreduce and has optimized implementation of VecOps. We also observed that PIPECG-OATI starts performing better than PCG when the cost of allreduce is large enough to be completely overlapped by two PC and two SPMV times which depends on the number of non zeroes in the matrix of the problem. This cost of allreduce increases as the number of cores increase, so we can say that PIPECG-OATI method gives performance improvements over PCG at higher number of cores. We also observed that PIPECG-OATI starts performing better than PIPECG when the overlap provided by PIPECG-OATI is greater than that provided by PIPECG. Since the overlap provided by PIPECG-OATI will become greater only when the allreduce cost increases which happens at higher number of cores, so we conclude that PIPECG-OATI performs better than PIPECG at higher number of cores. So, in this section we verify experimentally what we found theoretically in Section IV.

C. Accuracy Experiments and Results

In the OpenFOAM [14] [15] based Computational Fluid Dynamics applications which solve Pressure Poisson Equations, we see that default values for absolute tolerance are set to 10^{-6} for 3D problems and 10^{-3} for 2D problems. Our method reaches both of these values faster than the other methods.

The PIPECG method [7], PIPELCG method [8] and PIPECG3 method [10] have been shown to stagnate at higher values of absolute residuals than the PCG method. The PIPECG, PIPELCG, PIPECG3 and PIPECG-OATI methods are equivalent to the PCG method in exact arithmetic but in floating point arithmetic, rounding errors are introduced due to the introduction of recurrence relations and hence they stagnate at higher values of residuals.

In this section, we plot the values of absolute residuals reached by different methods against the time taken by them to reach these values at 110 nodes. This tells us that which method would reach a particular residual threshold fastest.

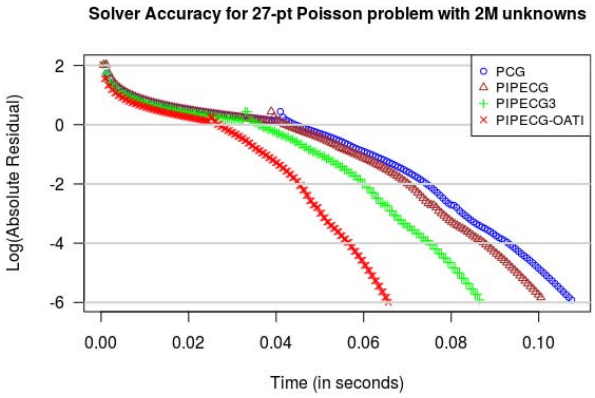

Fig. 5. Solver Accuracy/Performance Experiment for 27-pt Poisson problem. Absolute Residual Values as a function of time at 110 nodes.

Figure 5 shows the absolute residual values attained by each method as a function of time for the 27-pt 3D 2M unknowns problem on 110 nodes (2640 cores). Here, we see that PIPECG-OATI reaches the threshold of 10^{-6} fastest as compared to PCG, PIPECG and PIPECG3 with PCG being the slowest. This can be verified from Figure 4 as PCG

Authorized licensed use limited to: J.R.D. Tata Memorial Library Indian Institute of Science Bengaluru. Downloaded on December 23,2022 at 14:29:55 UTC from IEEE Xplore. Restrictions apply.

provides the least speedup and PIPECG-OATI provides the most speedup at 110 nodes.

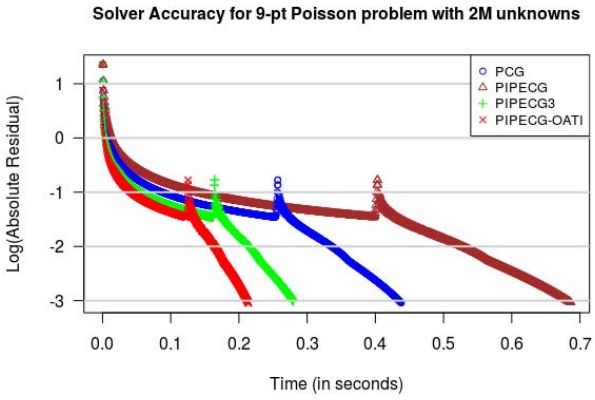

Fig. 6. Solver Accuracy/Performance Experiment for 9-pt Poisson problem. Absolute Residual Values as a function of time at 110 nodes.

Figure 6 shows the absolute residual values attained by each method as a function of time for the 9-pt 2D 2M unknowns problem on 110 nodes (2640 cores). Here we see that PIPECG-OATI reaches the threshold of 10^{-3} fastest as compared to PCG, PIPECG and PIPECG3 with PIPECG being the slowest.

Thus we conclude that for widely used values of absolute tolerance, our method can be used to solve the linear system of equations obtained from CFD applications with high performance.

D. Detailed Timing Breakdowns

In this section, we measure the times taken by the individual kernels in the PCG variants and see how they vary as the number of nodes increase. We present the detailed timing breakdown for 9-pt and 27-pt Poisson Problems with 2M unknowns. We note here that we split the Dot Product operation into two parts- one is the computation of the partial dot product and the other is the allreduce to compute the global dot product. We include the computation part in the VecOps kernel because in our optimized implementation (section V), we merge the vector operations for computing partial dot products with the vector operations for computing VMAs. We also note here that the allreduce time that we show in the figures correspond to the non-overlapped part of the allreduce kernel.

Figure 7 shows the detailed timing breakdown for PCG, PIPECG and PIPECG-OATI as number of nodes increase for the 9-pt 2D Poisson Problem with 2M unknowns. For all methods, we see that the time taken for PC and SPMV decreases as the number of nodes increase as more computing power becomes available for these compute-intensive kernels. For PCG, the time taken for VecOps remains almost constant as the number of nodes increase. The time taken for allreduce increases as the number of nodes increase. Also, the PCG has dependencies such that no part of the allreduce time

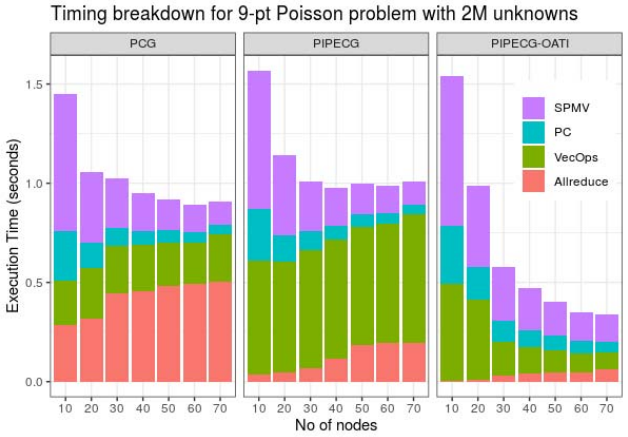

Fig. 7. Detailed timing breakdowns for PCG, PIPECG and PIPECG-OATI as number of nodes increase for 9-pt Poisson Problem with 2M unknowns

can be overlapped with useful computations. For PIPECG, we see that the VecOps take the most time in the total execution time because of unoptimized implementation of VecOps. PIPECG has independent computations which can be overlapped with a non-blocking allreduce and thus is able to overlap the allreduce. But here, as the number of nodes increase, the computations become smaller and time for allreduce becomes bigger, which results in decreasing overlap of the allreduce. For PIPECG-OATI, we see that the optimized implementation of VecOps helps to significantly decrease the time taken by VecOps. Also, the time taken for VecOps decreases as the number of nodes increase. PIPECG-OATI has more independent computations which can be overlapped with the nonblocking allreduce. This results in PIPECG-OATI is able to overlap allreduce with more work than PIPECG and hence significantly lesser times for the non-overlapped part of the allreduce in PIPECG-OATI, as shown in the results.

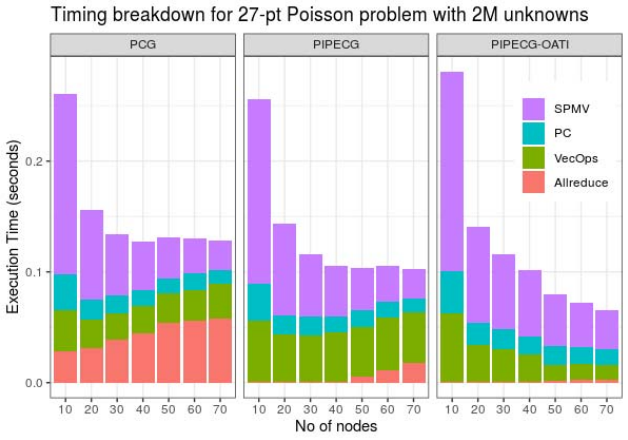

Fig. 8. Detailed timing breakdowns for PCG, PIPECG and PIPECG-OATI as number of nodes increase for 27-pt Poisson Problem with 2M unknowns

Figure 8 shows the detailed timing breakdown for PCG,

Authorized licensed use limited to: J.R.D. Tata Memorial Library Indian Institute of Science Bengaluru. Downloaded on December 23,2022 at 14:29:55 UTC from IEEE Xplore. Restrictions apply.

PIPECG and PIPECG-OATI as number of nodes increase for 27-pt 3D Poisson Problem with 2M unknowns. For PIPECG, we see that the VecOps are not the dominating factor in the total execution time because the PC and SPMV for 3D problems are compute intensive and they become the dominating factor. Here also, PIPECG-OATI is more effective in overlapping work at higher number of nodes than PIPECG. PIPECG-OATI starts performing better than PIPECG at 50 nodes because it can overlap larger part of allreduce as it has more work and it also has optimized VecOps. For this problem, the allreduce is completely overlapped in the PIPECG-OATI method and the communication time is almost non-existent resulting in linear scalability.

From Figures 7 and 8, we observe that PIPECG-OATI performs better than PCG and PIPECG when the time taken by allreduce increases and can be completely overlapped by the two PCs and two SPMVs provided by the problem. It also provides optimized VecOps which reduce the significant overhead introduced by the extra recurrence relations of the algorithm.

E. Performance Improvements due to Optimizations

We compared the performance of PIPECG-OATI with unoptimized VecOps to PIPECG-OATI with optimized VecOps. We found that the performance improvement is 64% for 9-pt, 55% for 27-pt and 30% for 125-pt problems. This shows us that if we didn't optimize the VecOps in PIPECG-OATI, we will not get any improvement over PIPECG because though the overlap may be more, the overhead introduced by 21 extra VMAs and 7 new Dot Products will dominate the time (similar to when PIPECG performs worse than PCG). Further we notice that as the problem becomes more compute intense (9-pt to 27-pt to 125-pt), the performance improvement given by optimized VecOps decrease because then the total execution time begins to be dominated by PC and SPMV. If we have a highly compute-intensive problem, unoptimized VecOps PIPECG-OATI and optimized VecOps PIPECG-OATI will give the same performance.

VII. Conclusion and Future Work

In this work, we developed a novel PIPECG-OATI method for Distributed Memory Systems which 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 more VMAs and Dot Products. We use iteration combination and also introduce new non-recurrence computations to achieve this aim. We provide an optimized implementation of PIPECG-OATI which helps in efficient memory accesses and hence gives performance improvements. Our method PIPECG-OATI gives up to 3x speedup wrt PCG, 1.73x speedup wrt PIPECG, 1.33x speedup wrt to PIPECG3 and 5x speedup wrt PIPELCG. We conclude that PIPECG-OATI gives performance benefits over PCG and PIPECG 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 problem.

In the 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 use Residual Replacement Strategy to increase the attainable accuracy of the PIPECG-OATI method while trying to bring down the number of PCs and SPMVs associated with it. We plan to test and analyse the behaviour of our method PIPECG-OATI on multi-node multi-GPU systems.

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

[4] 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

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

[6] W. Gropp, "Update on libraries for blue waters," Tech. Rep. [Online]. Available: http://jointlab.ncsa.illinois.edu/events/workshop3/pdf/presentations/Gropp-Update-on-Libraries.pdf

[7] 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

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

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

[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.

[11] 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

[12] S. Balay, S. A. a, 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 users manual," Argonne National Laboratory, Tech. Rep. ANL-95/11 - Revision 3.13, 2020. [Online]. Available: https://www.mcs.anl.gov/petsc

[13] S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith, "Efficient management of parallelism in object oriented numerical software libraries," in *Modern Software Tools in Scientific Computing*, E. Arge, A. M. Bruaset, and H. P. Langtangen, Eds. Birkhäuser Press, 1997, pp. 163–202.

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

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

Authorized licensed use limited to: J.R.D. Tata Memorial Library Indian Institute of Science Bengaluru. Downloaded on December 23,2022 at 14:29:55 UTC from IEEE Xplore. Restrictions apply.