A Parallel Implementation of Communication-Avoiding Conjugate Gradient Method

Manyuan Tao

Jun 9, 2017

1 Introduction

The speed of an algorithm is determined by two factors: the number of floating-point operations performed (computation) and the amount of data movement (communication). In the sequential case, communication refers to movement of data between levels of the memory hierarchy. In the parallel case, communication means movement of data between processors. Trends in hardware technology indicate that processor speed is increasing at an accelerating rate versus memory speed, making communication a bottleneck in many numerical algorithms. Whereas algorithmic performance optimizations have traditionally attempted to decrease the number of floating point operations, we now shift to a new paradigm: *avoiding communication*. [1]

People are led into research on a class of algorithms in which the amount of communication or the number of messages are reduced at the cost of increased amount of computations, compared to existing algorithms. Such algorithms are collectively called *Communication-Avoiding* or *CA* algorithms. My final project implemented a distributed-memory parallel version of Communication-Avoiding Conjugate Gradient method (CA-CG), where the algorithm was originally introduced by Carson, Knight and Demmel [2].

The rest of the report is organized as follows. In Section 2 we first review the Conjugate Gradient (CG) method and point out its communication. Then we introduce the general idea and algorithm of CA-CG [2]. In Section 3 we discuss strategies for implementing the distributed-memory parallelism in detail. These implementations are evaluated on Stampede in Section 4, on a 2D Poisson problem. We test the strong scalability and weak scalability. Also we would like to see whether there is a speedup by avoiding communication (CA-CG vs. CG). Lastly, we summarize our findings in Section 5.

2 Communication-Avoiding Krylov Subspace Methods

2.1 Conjugate Gradient (CG) and its Communication

We briefly review classical CG for solving $Ax = b$, where A is a sparse positive definite (SPD) matrix. The following Algorithm 1 is the pseudo-code for CG.

Algorithm 1 Conjugate Gradient (CG)

```
Input: initial approximation $x_0$ for solving $Ax = b$
1: Let $p_0 = r_0 = b - Ax_0$.
2: for $i = 0, 1, ...,$ until convergence do
3:     $\alpha_i = \frac{r_i^T r_i}{p_i^T A p_i}$
4:     $x_{i+1} = x_i + \alpha_i p_i$
5:     $r_{i+1} = r_i + \alpha_i A p_i$
6:     $\beta_i = \frac{r_{i+1}^T r_{i+1}}{r_i^T r_i}$
7:     $p_{i+1} = r_{i+1} + \beta_i p_i$
8: end for
```

In parallelizing CG method, communication is needed to compute one sparse matrix-vector multiplication (SpMV) Ap_i required by lines 3 and 5, and two inner products in lines 3 and 6 in each iteration. The other computations are scalar-vector products, vector additions/subtractions, and scalar operations, which require no communication.

2.2 Reduction of the Number of Inner Products [2] [3]

We split iteration loops into an inner loop over $0 \leqslant j < s$ and an outer loop over k, whose range depends on the number of steps until convergence. We index iteration m in CG as iteration $m = sk + j$ in CA-CG. By induction on lines 4, 5 and 7 of Algorithm 1, we can write

$$p_{sk+j},\ r_{sk+j},\ x_{sk+j} - x_{sk} \in \mathcal{K}_{s+1}(A, p_{sk}) + \mathcal{K}_s(A, r_{sk}) \quad \text{for } 0 \leqslant j < s,$$

where $\mathcal{K}_i(A, v)$ denotes the i-th Krylov subspace of A with respect to v, i.e.,

$$\mathcal{K}_i(A, v) = \text{span}\{v, Av, A^2v, \ldots, A^{i-1}v\}.$$

Therefore, in outer iteration k, for $0 \leqslant j < s$, we can represent $x_{sk+j} - x_{sk},\ r_{sk+j}$ and p_{sk+j} as linear combinations of vectors spanning the space $\mathcal{K}_{s+1}(A, p_{sk}) + \mathcal{K}_s(A, r_{sk})$. We let length-$(2s+1)$ vectors $x'_{k,j},\ r'_{k,j}$ and $p'_{k,j}$ denote the coordinates for $x_{sk+j} - x_{sk},\ r_{sk+j}$ and p_{sk+j}, resp., in the columns of

$$V_k = [P_k,\ R_k] = [\rho_0(A)p_{sk}, \ldots,\ \rho_s(A)p_{sk},\ \rho_0(A)r_{sk}, \ldots,\ \rho_{s-1}(A)r_{sk}],$$

where ρ_i is a polynomial of degree i. That is, we have

$$x_{sk+j} - x_{sk} = V_k x'_{k,j},\ r_{sk+j} = V_k r'_{k,j},\ \text{and } p_{sk+j} = V_k p'_{k,j},$$

for $0 \leqslant j < s$. For brevity, we will refer to V_k as a basis, although the columns of V_k need not be linearly independent.

We assume the polynomials ρ_i can be computed via a three-term recurrence in terms of parameters $\gamma_i, \theta_i,$ and σ_i, as

$$\rho_0(A) = 1, \ \rho_1(A) = (A - \theta_0 I)\rho_0(A)/\gamma_0, \text{ and}$$

$$\rho_{i+1}(A) = ((A - \theta_i I)\rho_i(A) - \theta_i \rho_{i-1}(A))/\gamma_i,$$

for $1 \leqslant i < s$. This three-term recurrence covers a large class of polynomials, including classical orthogonal polynomials. The monomial, Chebyshev bases are common choices for generating Krylov bases. To simplify notation, we assume the basis parameters remain the same throughout the iteration.

The basis V_k is generated at the beginning of each outer loop, using the current r_{sk} and p_{sk} vectors. Then we introduce Gram matrix G_k and matrix B_k in each outer loop to reduce communication in SpMV and inner products.

The product Ap_{sk+j} (SpMV) can be written

$$Ap_{sk+j} = AV_k p'_{k,j} = V_k B_k p'_{k,j},$$

where

$$B_k = \begin{bmatrix} [C_{k,s+1} \quad 0_{s+1,1}] & \\ & [C_{k,s} \quad 0_{s,1}] \end{bmatrix},$$

with

$$C_{k,j+1} = \begin{bmatrix} \theta_0 & \sigma_1 & & \\ \gamma_0 & \theta_1 & \ddots & \\ & \gamma_1 & \ddots & \sigma_{j-1} \\ & & \ddots & \theta_{j-1} \\ & & & \gamma_{j-1} \end{bmatrix}.$$

Recall that B_k and $p'_{k,j}$ are both of dimension $O(s)$, which means they either fit in fast memory (in sequential case) or are local to each processor (in parallel case), and thus the computation $B_k p'_{k,j}$ does not require data movement.

In each outer loop, we compute the $O(s)$-by-$O(s)$ Gram matrix $G_k = V_k^T V_k$. Then the inner products in lines 3 and 6 of Algorithm 1 can be written

$$r_{sk+j}^T r_{sk+j} = r'^T_{k,j} G_k r'_{k,j} \quad \text{for} \quad 0 \leqslant j < s, \quad \text{and}$$

$$p_{sk+j}^T A p_{sk+j} = p'^T_{k,j} G_k B_k p'_{k,j} \quad \text{for} \quad 0 \leqslant j < s.$$

Thus, after G_k has been computed in the outer loop, the inner products can be computed without additional communication.

2.3 Communication-Avoiding Conjugate Gradient (CA-CG)

The above gives the general idea behind avoiding data movement in Lanczos-based Krylov subspace methods. The resulting CA-CG method is shown below in Algorithm 2.

Algorithm 2 Communication-Avoiding Conjugate Gradient (CA-CG)

```
Input: initial approximation $x_0$ for solving $Ax = b$
 1: Let $p_0 = r_0 = b - Ax_0$.
 2: for $k = 0, 1, \dots,$ until convergence do
 3:     Calculate $P_k,\ R_k$, bases for $\mathcal{K}_{s+1}(A, p_{sk}),\ \mathcal{K}_s(A, r_{sk})$, resp.
 4:     Let $V_k = [P_k,\ R_k] = [\rho_0(A)p_{sk}, \dots,\ \rho_s(A)p_{sk},\ \rho_0(A)r_{sk}, \dots,\ \rho_{s-1}(A)r_{sk}]$
 5:     Compute Gram matrix $G_k = V_k^T V_k$; assemble $B_k$
 6:     Coordinates $x'_0 = 0_{2s+1},\ r'_0 = [0_{s+1}^T, 1, 0_{s-1}^T]^T,\ p'_0 = [1, 0_{2s}^T]^T$
 7:     for $j = 0, ..., s-1$ do
 8:         $\alpha_{sk+j} = \frac{r_j'^T G_k r'_j}{p_j'^T G_k B_k p'_j}$
 9:         $x'_{j+1} = x'_j + \alpha_{sk+j} p'_j$
10:         $r'_{j+1} = r'_j - \alpha_{sk+j} B_k p'_j$
11:         $\beta_{sk+j} = \frac{r_{j+1}'^T G_k r'_{j+1}}{r_j'^T G_k r'_j}$
12:         $p'_{j+1} = r'_{j+1} + \beta_{sk+j} p'_j$
13:     end for
14:     Compute $x_{sk+s} = V_k x'_s + x_{sk},\ r_{sk+s} = V_k r'_s,\ p_{sk+s} = V_k p'_s$
15: end for
```

This is also called an "S-Step" CG method. We obtain 1 communication step in the outer iteration and s computation steps in inner iterations. This formulation allows an $O(s)$ reduction in communication.

Several technical strategies in CA-CG:

1. Compute V_k: read A (sequential case) / communicate vector entries with neighbors (parallel case) only $O(1)$ times, using matrix powers kernel described in [4]. But I do not implement this since it is more technically involved. To simplify things, I just do SpMVs s times instead.

2. Polynomials ρ_i:

   - Monomial basis: $\rho_m(A) = A^m$, not numerically stable;

   - Chebyshev basis: $T_m(x)$ on the interval $[\lambda_{min}, \lambda_{max}]$ (the smallest and largest eigenvalues of A) by scaling and shifting the usual Chebyshev polynomial on the interval $[-1, 1]$, still not perfect.

3. Choice of s: expect to see better convergence for smaller values of s.

3 Distributed Memory Parallelism

In this project, we implement a distributed-memory parallel version of both CG and CA-CG. The followings are my detailed MPI-parallel implementation.

1. Coefficient matrix A (implicit):

   Implement on discretization of a 2D Poisson problem (five-point stencil).

   Use a uniform domain splitting as sketched in Figure 1.

   Partition the $N \times N$ mesh into blocks of size $\frac{N}{\sqrt{p}} \times \frac{N}{\sqrt{p}}$, where p is the number of MPI processes, and we denote $N_l = \frac{N}{\sqrt{p}}$ (p must be a square number).

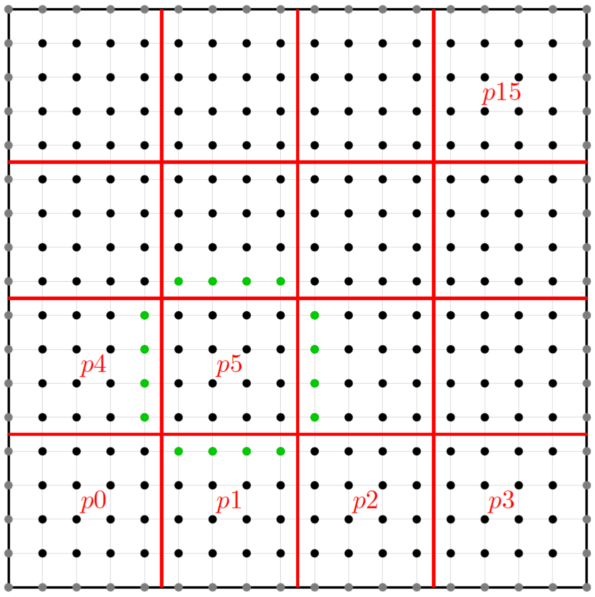


Figure 1: Uniform splitting of vector entries with 16 MPI processors, and with $N_l = 4$. Vector entries are shown as black dots, gray dots are domain boundary values. For example, the ghost nodes processor $p5$ requires for updating its values in a SpMV are shown in green. $p5$ needs to obtain these values through communication with $p1, p4, p6, p9$, where they are updated.

2. Nearest-neighbor communication in SpMV:

   Allocate $(N_l + 2)^2$ vector entries for each MPI process.

   Communicate the left/right/top/bottom ghost vectors with its neighbors.

   The "inner" N_l^2 points are updated by each MPI process. (See 8 for formula)

   The "outer" points are used to store and update the ghost point copies from neighboring MPI processes.

3. Compute basis V_k:

   I do not implement the matrix powers kernel, and just do s SpMVs instead.

   V_k will be distributed rowwise amongst the MPI processes.

4. Polynomials ρ_i:

   In sequential case, I use the monomial and Chebyshev basis separately.

   In parallel case, I use the monomial basis only.

   Obtain the corresponding matrix B_k each time.

5. Construct Gram matrix G_k:

   We use the MPI_AllReduce operation in computing $G_k = V_k^T V_k$ since V_k is distributed rowwise amongst the MPI processes.

   Every MPI process will have a copy of G_k after the MPI_AllReduce operation.

6. Inner loop:

   The whole inner loop is local computation, i.e., no MPI communication between processors.

   Implement matrix-vector multiplication, vector-vector inner products.

   Update $(2s+1)$-vectors of coordinates of $p_{sk+j},\ r_{sk+j},\ x_{sk+j} - x_{sk}$ in V_k, which replace SpMVs and inner products.

7. Choice of s:

   Use the best results we recorded after tuning over s values.

8. Five-point stencil formula for SpMV:

   In SpMV $A\vec{u}$, the sparse coefficient matrix A (block tridiagonal) is given by

$$A = \frac{1}{h^2}\begin{bmatrix} D & -I & 0 & \cdots & 0 \\ -I & D & -I & & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & & -I & D & -I \\ 0 & \cdots & 0 & -I & D \end{bmatrix}_{N^2\times N^2}, \text{ with } D = \begin{bmatrix} 4 & -1 & 0 & \cdots & 0 \\ -1 & 4 & -1 & & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & & -1 & 4 & -1 \\ 0 & \cdots & 0 & -1 & 4 \end{bmatrix}_{N\times N},$$

   I is the $N \times N$ identity matrix,

   and the vector $\vec{u}$ is discretized as

$$\vec{u} = [u_{11}, u_{12}, \cdots, u_{1N}, u_{21}, u_{22}, \cdots, u_{2N}, \cdots\cdots, u_{N1}, u_{N2}, \cdots, u_{NN}]^T .$$

   Discretizing the 2D Poisson equation gives the following formula:

$$(A\vec{u})_{ij} = \frac{-u_{i-1,j} - u_{i,j-1} + 4u_{ij} - u_{i+1,j} - u_{i,j+1}}{h^2} \quad \text{for } 1 \leqslant i,j \leqslant N,$$

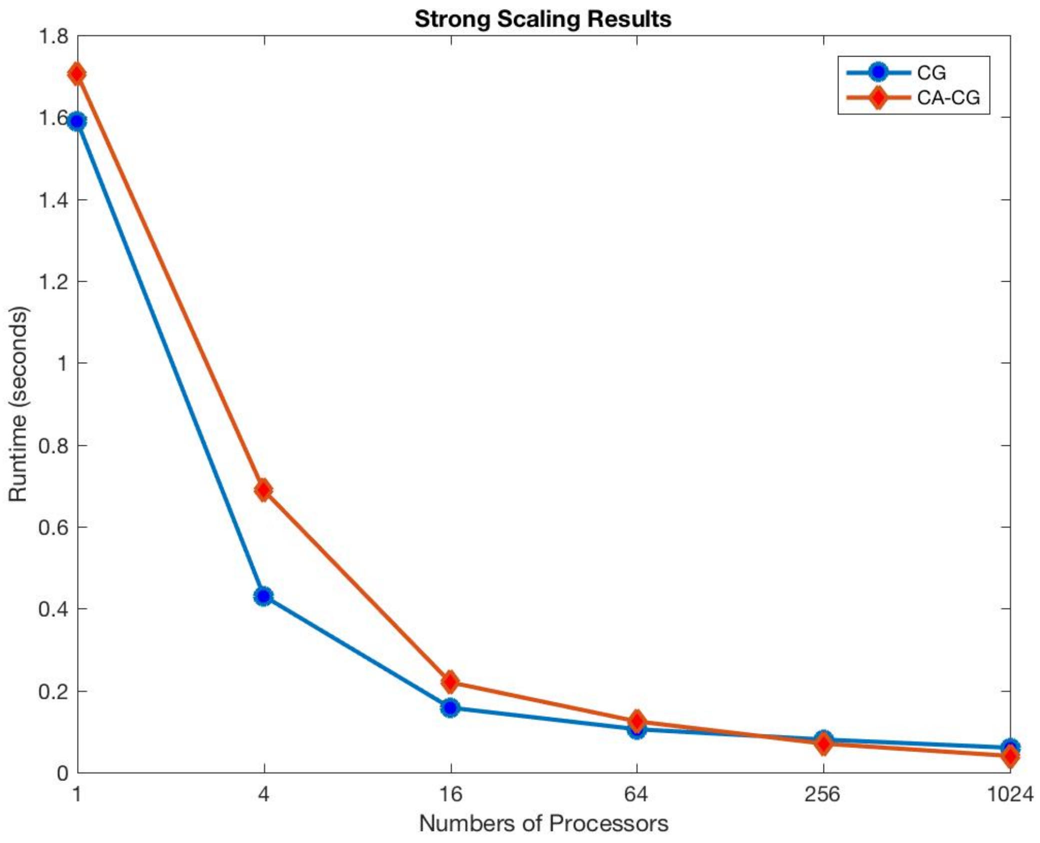
   which is used to update the inner points processed by each MPI task in 2.

4 Parallel Experiments on Stampede

In this section, we report the strong scalability and weak scalability of our numerical experiments run on the Stampede supercomputer. I have written the MPI parallel versions of both CG and CA-CG, solving a 2D Poisson equation on the unit square domain $(0,1) \times (0,1)$. Also we would like to see whether there is a speedup by avoiding communication (CA-CG vs. CG).

4.1 Strong Scaling Results

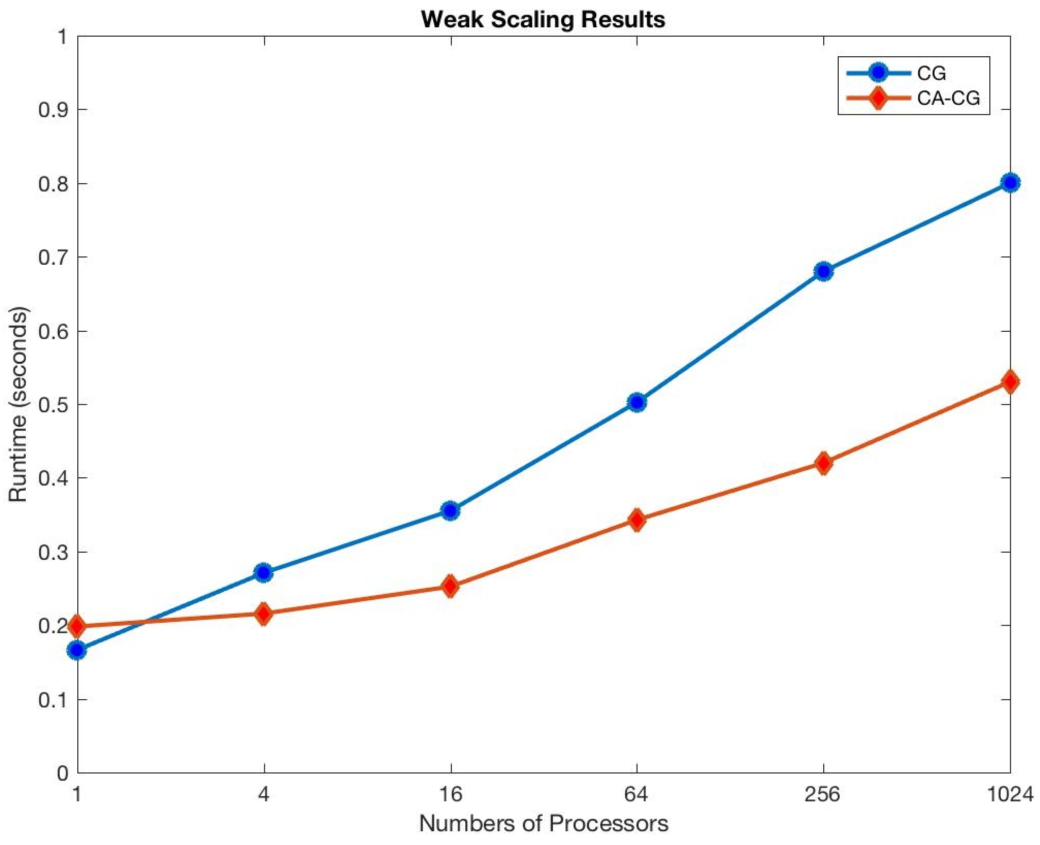
We perform strong scaling tests for our MPI parallel implementations of CG and CA-CG. We fix the number of discretization points $N = 480$ (i.e., keep the total problem size unchanged) while increasing the number of processors, so that $\frac{N}{\sqrt{p}}$ decreases with increased p.



From the above figure, we generally see good speedup results, with a perfect decrease in runtime going from 1 processor to 4 processors. The speedup in the CA-CG case is less perfect than that in CG. This is due to the sequential computations of coordinates in each iteration of the inner loop. The parallelization of calculating these coordinates is nontrivial, and is part of future work. We expect that this will further contribute to strong scaling performance.

4.2 Weak Scaling Results

We test weak scaling performance for our MPI parallel implementations of CG and CA-CG. We fix the ratio $\frac{N}{\sqrt{p}} = 100$ (i.e., keep the problem size per MPI processor unchanged) and increase the number of processors.



CA-CG shows better weak scalability. When the number of processors increases from 1 to 1024, the runtime increases only a little.

4.3 CA-CG vs. CG

Comparing the red and blue lines in the above two figures, we have the following findings. When $\frac{N}{\sqrt{p}}$ is relatively small, we can see a speedup by avoiding communication. When the problem size per processor is relative big, CG still performs better than CA-CG in runtime. This is because I did not implement the matrix powers kernel in computing basis V_k, which could contribute to further communication-avoiding speedup.

5 Conclusion

In this project, we present a parallel implementation of the CG and CA-CG algorithms on distributed memory systems. Our implementation solves a 2D Poisson problem on the unit square domain. The results show that our parallel experiments are able to scale both strongly and weakly on a large distributed memory system — Stampede, and some comparisons are discussed. We did not see an obvious speedup in CA-CG compared to CG, because we did not implement the matrix powers kernel, which is a very useful tool in avoiding communication as well. But when $\frac{N}{\sqrt{p}}$ is relatively small, we can still see a speedup just from the reduction in MPI_AllReduce operations.

My project is more of an educational purpose and we have no intention to push everything to the extreme. For a more elaborate and comprehensive study, which implements the matrix powers kernel and uses more fancy polynomials, we refer the readers to the methods and experiments in [1] [5].

References

[1] Erin Carson and Nicholas Knight. A parallel implementation of the communication-avoiding biconjugate gradient method. 2010.

[2] Erin Carson, Nicholas Knight, and James Demmel. An efficient deflation technique for the communication-avoiding conjugate gradient method. Electronic Transactions on Numerical Analysis, 2014.

[3] Reiji SUDA, Cong LI, and Daichi WATANABE. Communication-avoiding cg method: New direction of krylov subspace methods towards exa-scale computing. RIMS Kokyuroku Bessatsu, pages 102–111, 1995.

[4] Marghoob Mohiyuddin, Mark Hoemmen, James Demmel, and Katherine Yelick. Minimizing communication in sparse matrix solvers. Proceedings of the Conference on High Performance Computing Networking, 2009.

[5] Erin Carson, Nicholas Knight, and James Demmel. Efficient deflation for communication-avoiding krylovsubspace methods. NASCA, 2013.