PARALLELIZING CONJUGATE GRADIENT METHODS

Jiannan Jiang, Yujie Wang*
Department of Mathematics, Electrical Engineering and Computer Science
University of California, Berkeley
jjn2015, yujie_wang@berkeley.edu

ABSTRACT

In this paper, we have implemented the communication-avoiding variant of conjugate gradient method and tuned its performance. We have tried various kinds of tunings for the sparse matrix-vector-multiply kernel, invented a new smoothing variant of conjugate gradient method that helps the convergence behavior, and benchmarked the resulting performance with the standard library.

All of the code used above reside in `http://github.com/~jonny97/CG_variants`.

***K*eywords** Conjugate Gradient · Parallel Computing · Communication-avoiding

1 Introduction

Conjugate gradient (CG) method is a popular method for solving large sparse linear systems [1]. The method is provably efficient when we have a sparse and well-conditioned matrix A. CG method is useful in many real-world scenarios such as solving partial differential equations and optimization problems. In this project, we have implemented a parallel version of conjugate gradient method using OpenMP and a sparse matrix-vector-multiplication (SpMV) kernel pOSKI [2]. To avoid communication, we have also implemented a communication avoiding variant of CG based on the [3]. This method is mathematically equivalent to CG and is shown much faster than CG with proper parameters [4].

2 Background

2.1 Conjugate Gradient Method

In contrast to traditional gradient-based methods, conjugate gradient method aims to reduce the number of iterations needed. The popular gradient-based framework updates the solution x along the direction of gradients. However, the number of iterations required to converge can be much higher than the problem size and the cost of computing the gradient at each iteration involves two sparse matrix-vector-multiplications.

The Conjugate gradient method relies on the Krylov subspace that provides an easy way to generate a set of A-conjugate vectors

$$K_i(A, d) = span\left\{d, Ad, A^2d, ..., A^{i-1}d\right\} \tag{1}$$

The conjugate gradient method chooses $x_i \in K_i(A, d)$ to minimize $(x - x^*)^T A(x - x^*)$ [5]. In each iteration i, we need to update the next solution x_{i+1} and residual r_{i+1}, which consists of only one SpMV, one inner product of the residue vector, and several vector-vector additions [3]. The classic conjugate gradient method is shown in Algorithm 1.

*This is the final project for CS267 Applications of Parallel Computers at University of California Berkeley in Spring 2019

Algorithm 1 Conjugate Gradient Method (CG)

Require: Approximate solution x_0 to $Ax = b$
1: $r_0 = b - Ax_0, d_0 = r_0$
2: **for** `i=0,1,...until convergence` **do**
3: $\alpha_i = (r_i^T r_i)/(d_i^T A d_i)$
4: $x_{i+1} = x_i + \alpha_i d_i$
5: $r_{i+1} = r_i - \alpha_i A d_i$
6: $\beta_{i+1} = (r_{i+1}^T r_{i+1})/(r_i^T r_i)$
7: $d_{i+1} = r_{i+1} + \beta_{i+1} d_i$
8: **end for**
9: **return** x_{i+1}

2.2 Communication-Avoiding Conjugate Gradient Method (CA-CG)

Since movement of data plays a larger role than computation in limiting performance in modern computers [3], thus multiple performance tuning has been done for this method, including devising communication avoiding variants and pipelining the methods over the past few decades. Specifically, most of the communications occur in SpMV and in calculating the inner products of vectors.

We have used an auto-tunning library pOSKI [2] for computing the SpMV occurred. This library automatically partitions the matrix and generate multiple threads to parallel.

Since there is not much space for improvement on vector operations, Communication-Avoiding Conjugate Gradient method will combine multiple iterations of CG together to reduce the communication for calculating inner products of vectors. Thus, CA-CG will be running matrix multiplications instead of multiple vector inner products, which will be faster since current BLAS-3 implementation can be much faster than the equivalent multiple executions of BLAS-2. In CA-CG, iterations are split into an inner loop over $0 \leq j < s$ and an outer loop over i, whose range depends on the number of steps until convergence. We index the iteration i in CG as the iteration $si + j$ in CA-CG [3]. The communication-avoiding conjugate gradient method is shown in Algorithm 2.

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

Require: Approximate solution x_0 to $Ax = b$
1: $r_0 = b - Ax_0, d_0 = r_0$
2: **for** `i=0,1,...until convergence` **do**
3: `Compute` P_i, R_i, `let` $V_i = [P_i, R_i]$; assemble B_i
4: $G_i = V_i^T V_i$
5: $d_{i,0}^{'T} = [1, 0_{2s}^T], r_{i,0}^{'T} = [0_{s+1}^T, 1, 0_{s-1}^T], x_{i,0}^{'} = 0_{2s+1}$
6: **for** `j=0,1,...,s-1` **do**
7: $\alpha_{si+j} = (r_{i,j}^{'T} G_k r_{i,j}^{'})/(d_{i,j}^{'T} G_i B_i d_{i,j}^{'})$
8: $x_{i,j+1}^{'} = x_{i,j}^{'} + \alpha_{si+j} d_{i,j}^{'}$
9: $r_{i,j+1}^{'} = r_{i,j}^{'} - \alpha_{si+j} B_i d_{i,j}$
10: $\beta_{si+j+1} = (r_{i,j+1}^{'T} G_i r_{i,j+1}^{'})/(r_{i,j}^{'T} G_i r_{i,j}^{'})$
11: $d_{i,j+1} = r_{i,j+1}^{'} + \beta_{si+j+1} d_{i,j}^{'}$
12: **end for**
13: $x_{si+s} = V_i x_{i,s}^{'} + x_{si}, r_{si+s} = V_i r_{i,s}^{'}, d_{si+s} = V_i d_{i,s}^{'}$
14: **end for**
15: **return** x_{si+s}

2.3 Testing Data

In this project, we choose several large sparse matrices from SuiteSparse Matrix Collection [6]. For each test, we generate a true answer x either uniformly randomly from $[-1, 1]$ or from a standard normal distribution. We then compute the right hand side b via $b = Ax$, and use our implementations to find the true answer. Since some of the problems are actually ill-conditioned, we will only compare the magnitude of residue $r := b - Ax$ for convergence.

3 Tuning Sparse-Matrix-Vector-Multiplication (SpMV)

The pOSKI framework provides us an easy way to tune the parallel performance. pOSKI supports both thread-level tuning and data-partition tuning.

3.1 Thread-level tuning

The current pOSKI implementation supports three types of threading model: THREAD POOL, PTHREAD, and OPENMP. The following is a small test of the performance of CG on dataset *cant* with 2000 iterations. The horizontal axis is the number of threads we use per node and the vertical axis is the total cpu time aross all threads. As shown from the results, due to the memory constraints, the overhead for multiple threads is not negligible on this medium sized problem (*cant* is a sparse matrix of 62451 rows, 62451 columns, and 2034917 nonzeros in the upper triangle of the matrix). As we can see from Figure1, for this problem, the thread model PTHREAD is the best.

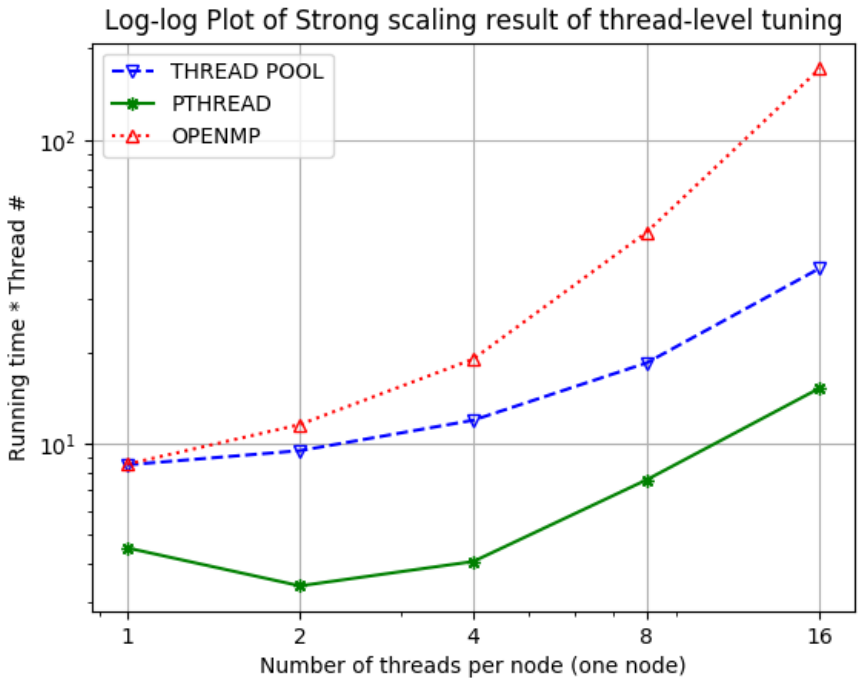


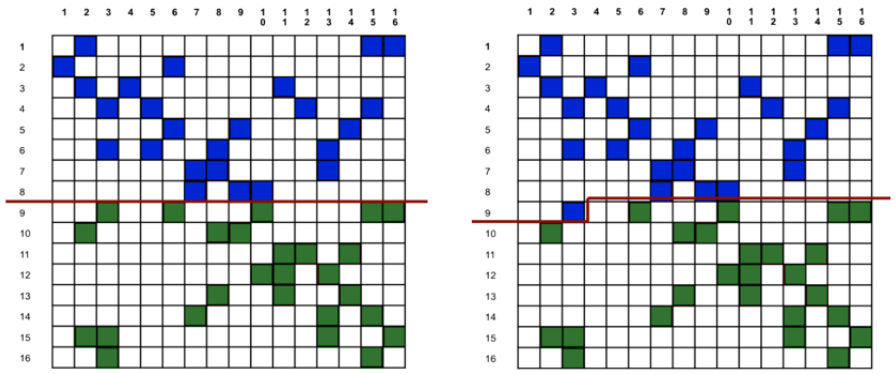
Figure 1: Log-log Plot of Strong scaling result of thread-level tuning

3.2 Data-partition tuning

pOSKI also provides a Data-partition tuning option. We can decide the number of partitions explicitly. Since the computation model is simple, we have always set the number of data partitions equal to the number of threads. We can also choose how we partition the dataset: (a) OneBdimensional row-wise partition by rows (OneD), and (b) One-dimensional row-wise partition by nonzeros (SemiOneD) as shown in Figure2. Note that, in this project, we only focus on the symmetric problems. To save memory, we only store the upper half of the matrix of A as U,and calculate $Ax = (U + U^T)x$. In this case, we should expect the second method always be faster as the OneD partition does not have a balanced load. However, OneD scheme only need one reduction whereas SemiOneD will require one more reductions. Figure3 shows the difference between two schemes. For this problem, the difference is not obvious.

4 Overall convergence of CA-CG and CG

As CG and CA-CG are both based on Krylov subspace methods, the numerical stability is always the first concern when introducing the implementations of these algorithms. Based on the size of the problem and error patterns, I have



(a) OneD: *One-Dimensional partition by rows* (b) SemiOneD: *One-Dimensional partition by nonzeros*
Figure C.1: Partitioning techniques supported in current pOSKI library.

Figure 2: Illustration of data-partition techniques in pOSKI [7]

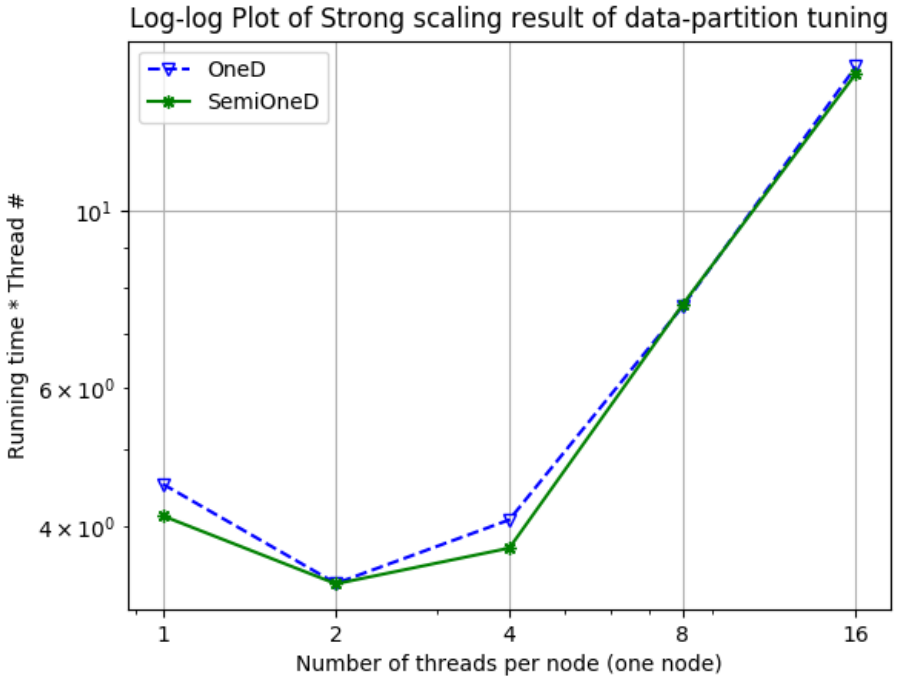


Figure 3: Log-log Plot of Strong scaling result of data-partition tuning

devised a smoothing trick to make the residue of CG going down in a more smooth manner. Figure4 is a plot of the convergence behavior of CG and the smoothing-variant of CG on dataset *cant*.

Note that this trick also smooths the performance of CA-CG as well. In the report given by Jeffery [4], we observed that CA-CG will not have a converging residue on the dataset *cant*. Use our smoothing trick, Figure5 shows the convergence of CA-CG:

As shown above, if we pack too much inner iterations, the algorithm does suffer from numerical instability. Basically the smoothing variant modifies the way of updating the directional vector in each iteration. For details about any mathematical intuitions and stabilities, please refer to the following page: `http://github.com/~jonny97/CG_variants`

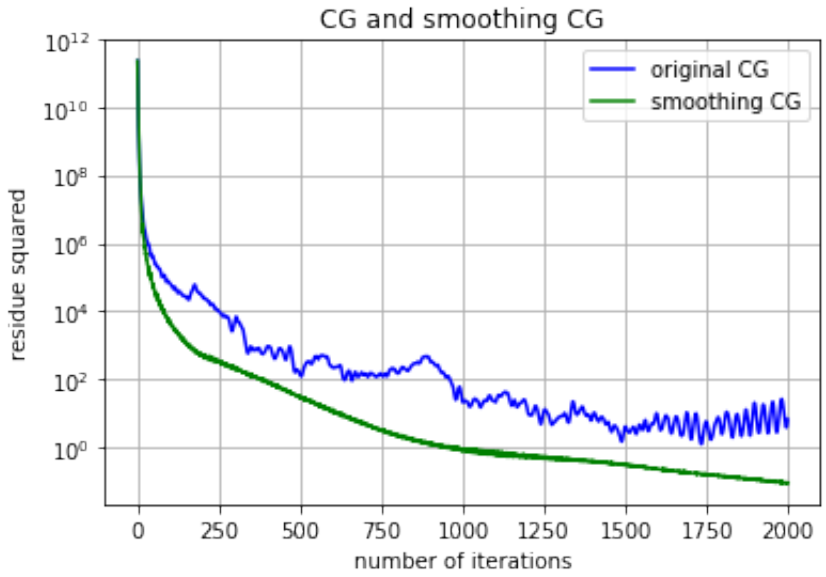


Figure 4: Convergence behavior of CG and the smoothing-variant of CG on dataset *cant*

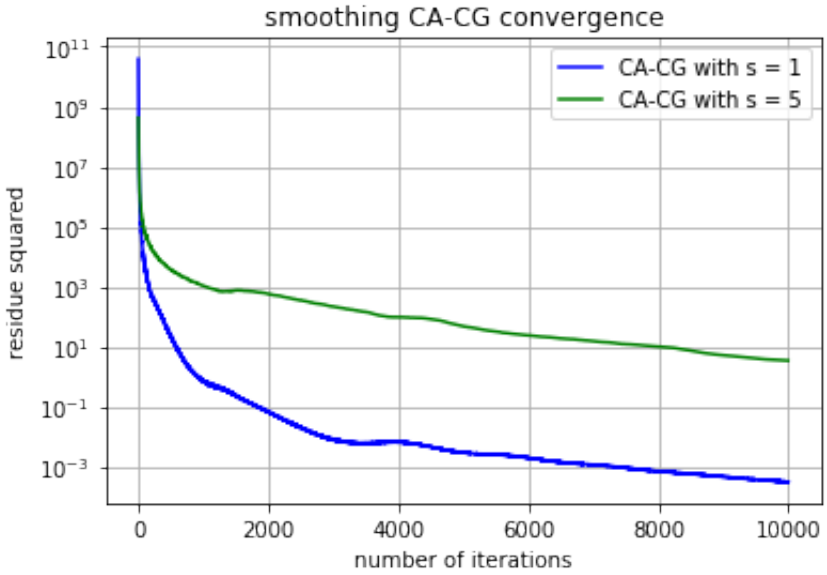


Figure 5: Convergence behavior of CA-CG with different inner loop sizes on dataset *cant*

5 Performance of CA-CG

The performance of CA-CG is hugely depending on the number of inner iterations. Given enough fast memory, the time per iteration of CA-CG will always decrease as we increase the number of inner iterations, with a potential cost of getting worse convergence behavior. The following is a plot of performance of CA-CG again the CG in SciPy. (Here we have used the best parameters. For details, please see the code section.)

6 Conclusion and Future Work

Conclusion In practice, we will not be able to know the true convergence behavior beforehand. Therefore, the trade-off decision between stability and latency is hard to make. To make the problem worse, the convergence will be problem-dependant as well and it is not cheap to get an good estimate for convergence. Therefore, we probably want to choose a small number of inner iterations when we want a good solution for the linear system while we might be using larger number of inner iterations when we are less worried about the residue of the result.
The good news is that all the tuning based on the nonzero patterns will remain work in any settings. The pOKSI tuning (both thread and data partitioning) can always be done at compile time once we know the sparsity pattern of the problem.

References

[1] Jonathan Richard Shewchuk et al. An introduction to the conjugate gradient method without the agonizing pain, 1994.

[2] Jong-Ho Byun, Richard Lin, James W Demmel, and Katherine A Yelick. poski: Parallel optimized sparse kernel interface library. In *Technical report*. University of California, 2012.

[3] Erin Carson, Nicholas Knight, and James Demmel. An efficient deflation technique for the communication-avoiding conjugate gradient method. *Electronic Transactions on Numerical Analysis*, 43(125141):09, 2014.

[4] Jeffrey Morlan, Shoaib Kamil, and Armando Fox. Auto-tuning the matrix powers kernel with sejits. In *International Conference on High Performance Computing for Computational Science*, pages 391–403. Springer, 2012.

[5] Gene H Golub and Dianne P OâĂŹLeary. Some history of the conjugate gradient and lanczos algorithms: 1948–1976. *SIAM review*, 31(1):50–102, 1989.

[6] Tim Davis, Yifan Hu, and Scott Kolodziej. The suitesparse matrix collection. *ACM Transactions on Mathematical Software*, 38(1):1–25, 2011.

[7] James W Demmel and Katherine A Yelick. poski: Parallel optimized sparse kernel interface library userâĂŹs guide for version 1.0. 0 jong-ho byun richard lin. 2012.