Acta Technica 62 (2017), No. 5B, 821–828 © 2017 Institute of Thermomechanics CAS, v.v.i.

A parallel Biconjugate A-Orthonormalization algorithm for linear system

CHENG SHAOHUA[2], ZHANG LITAO[2], ZHAO JIANFENG[3]

Abstract. In this paper, based on the Biconjugate A-Orthonormalization (BiCOR) algorithm in Jing et al. (Journal of Computational Physics, 228:6376–6394,2009) and **the ideas in Gu et al.** (Applied Mathematics and Computation 186:1243–1253, 2007), the authors present a parallel Biconjugate A-Orthonormalization (PBiCOR) algorithm for linear systems. The **new algorithms reduce two** global synchronization points to one by changing the computation sequence in the Biconjugate A-Orthonormalization (BiCOR) algorithm, and all inner products per iteration are independent and communication time required for inner product can be overlapped with useful computation. **Theoretical analysis shows that** the PBiCOR method has better parallelism and scalability than the BiCOR method.

1. Introduction

One of the fundamental tasks of numerical computation is to solve linear systems. These systems arise frequently in scientific computing, for example, from finite difference or finite element discretization of partial differential equations (PDEs), as intermediate steps in finding the solution of non-linear problems or as sub-problems in linear and non-linear programming. Usually, these systems are large, sparse and

[1]Acknowledgement - This research of this author is supported by NSFC (11226337, 11501525), Excellent Youth Foundation of Science Technology Innovation of Henan Province (184100510004),Science Technology Innovation Talents in Universities of Henan Province(16HASTIT040, 17HASTIT012) , Aeronautical Science Foundation of China (2016ZG55019), Project of Youth Backbone Teachers of Colleges and Universities of Henan Province (2015GGJS-003, 2015GGJS-179), Henan Province Postdoctoral Science Foundation (2013031), Research on Innovation Ability Evaluation Index System and Evaluation Model (142400411268).

[2]Workshop 1 - College of Science, Zhengzhou University of Aeronautics, Zhengzhou, Henan, 450015, P. R. China

[3]Workshop 2 - Information Engineering Department, Henan Polytechnic, Zhengzhou, Henan, 450046, P. R. China; e-mail: 670946614@qq.com

non-symmetric and solved by iterative methods [3].

Among the iterative methods for large sparse systems, Krylov subspace methods are the most powerful. For example, conjugate gradient (CG) method for solving symmetric positive definite linear systems, the GMRES method, BiCG method [3], quasi-minimal residual(QMR)method [2], BiCGStab method [5] and biconjugate residual (BiCR) method [4] for solving non-symmetric linear systems and so on. However, the Krylov subspace methods enforce bottleneck, i.e., the global communication induced by inner product computations, when used in large-scale parallel computing.

In this paper, based on Biconjugate A Orthonor-malization (BiCOR) algorithm for large sparse linear systems in Jing et al. (2009), the authors present a parallel Biconjugate A-Orthonormalization (PBiCOR) algorithm, which is designed for distributed parallel environments. The parallel BiCOR method is reorganized without changing the numerical stability and all inner products per iteration are independent (only one single global synchronization point), and subsequently communication time required for inner products can be overlapped efficiently with computation time. The cost is only a little increased computation. Theoretical analysis shows that the PBiCOR method has better parallelism and scalability than the BiCOR method, and the parallel performance can be improved by a factor of about 3/2.

2. Material and methods

In 2009, Jing et al. (2009) proposed the Biconjugate A Orthonor-malization (BiCOR) algorithm, which is written as follows:

Algorithm 2 : Parallel BiCOR Procedure (PBiCOR)

1. Set $\beta_1 = \delta_1 = 0, w_0 = v_0 = w_1 = v_1 = 0 \in C^n$
2. For $j = 1, 2, ..., m$ Do :
3. $\alpha_j = \langle w_j, A(Av_j)\rangle, c_j = \langle w_j, A(Av_j)\rangle, f_j = \langle w_j, A(Av_{j-1})\rangle,$ $d_j = \langle w_j, Av_j\rangle, g_j = \langle w_j, Av_{j-1}\rangle, h_j = \langle w_{j-1}, A(Av_j)\rangle,$ $k_j = \langle w_{j-1}, Av_j\rangle, l_j = \langle w_{j-1}, Av_{j-1}\rangle$
4. $\pi_{j+1} = \langle \hat{w}_{j+1}, A\hat{v}_{j+1}\rangle = c_j - 2 - \beta_j f_j - \alpha_j^2 d_j$ $+ \alpha_j\beta_j g_j - \delta_j h_j + \delta_j\alpha_j k_j + \delta_j\beta_j l_j$
5. $\hat{v}_{j+1} = Av_j - \alpha_j v_j - \beta_j v_{j-1}$
6. $\hat{w}_{j+1} = A^H w_j - \alpha_j w_j - \delta_j w_{j-1}$
7. $\delta_{j+1} = |\pi_{j+1}|^{1/2}$
8. $\beta_{j+1} = \frac{1}{\delta_{j+1}}\pi_{j+1}$
9. $v_{j+1} = \frac{\hat{v}_{j+1}}{\delta_{j+1}}$
10. $w_{j+1} = \frac{\hat{w}_{j+1}}{\beta_{j+1}}$
12. EndDo

In Algorithm 1, steps 4) and 8) require inner products and have close data dependency. So there are two global synchronization points per iteration. These global communication costs become relatively more and more important when the number of parallel processors is increased and thus they have the potential to affect the scal-

ability of the algorithm in a negative way. Gu et al. (2007) proposed an improved biconjugate residual algorithm where the synchronization overhead is effectively reduced by a factor of two. Based on their similar ideas, the author propose a parallel BiCOR algorithm which is to overlap the main computational kernels such as vector updates, matrix-vector multiplications and inner products so that they can be executed in parallel per iteration of the algorithm.

Define

$$\begin{aligned} &c_j = \langle w_j, A(Av_j)\rangle, f_j = \langle w_j, A(Av_{j-1})\rangle, d_j = \langle w_j, Av_j\rangle, \\ &g_j = \langle w_j, Av_{j-1}\rangle, h_j = \langle w_{j-1}, A(Av_j)\rangle, k_j = \langle w_{j-1}, Av_j\rangle, \\ &l_j = \langle w_{j-1}, Av_{j-1}\rangle \end{aligned}$$

Then we have

$$\begin{aligned} &\pi_{j+1} = \langle \hat{w}_{j+1}, A\hat{v}_{j+1}\rangle = c_j - 2 - \beta_j f_j - \alpha_j^2 d_j \\ &+\alpha_j\beta_j g_j - \delta_j h_j + \delta_j\alpha_j k_j + \delta_j\beta_j l_j \end{aligned}$$

The parallel BiCOR method can be presented in the following:

Algorithm 2 : Parallel BiCOR Procedure (PBiCOR)

1. Set $\beta_1 = \delta_1 = 0, w_0 = v_0 = w_1 = v_1 = 0 \in C^n$
2. For $j = 1, 2, ..., m$ Do :
3. $\alpha_j = \langle w_j, A(Av_j)\rangle, c_j = \langle w_j, A(Av_j)\rangle, f_j = \langle w_j, A(Av_{j-1})\rangle,$ $d_j = \langle w_j, Av_j\rangle, g_j = \langle w_j, Av_{j-1}\rangle, h_j = \langle w_{j-1}, A(Av_j)\rangle,$ $k_j = \langle w_{j-1}, Av_j\rangle, l_j = \langle w_{j-1}, Av_{j-1}\rangle$
4. $\pi_{j+1} = \langle \hat{w}_{j+1}, A\hat{v}_{j+1}\rangle = c_j - 2 - \beta_j f_j - \alpha_j^2 d_j$ $+ \alpha_j\beta_j g_j - \delta_j h_j + \delta_j\alpha_j k_j + \delta_j\beta_j l_j$
5. $\hat{v}_{j+1} = Av_j - \alpha_j v_j - \beta_j v_{j-1}$
6. $\hat{w}_{j+1} = A^H w_j - \alpha_j w_j - \delta_j w_{j-1}$
7. $\delta_{j+1} = |\pi_{j+1}|^{1/2}$
8. $\beta_{j+1} = \frac{1}{\delta_{j+1}}\pi_{j+1}$
9. $v_{j+1} = \frac{\hat{v}_{j+1}}{\delta_{j+1}}$
10. $w_{j+1} = \frac{\hat{w}_{j+1}}{\beta_{j+1}}$
12. EndDo

Remark 2.1 The inner products of a single iteration step 3) is independent(parallel).

3. Theoretical Results

The recurrence process of the PBiCOR method shows that **the PBiCOR and the BiCOR methods** are mathematically equivalent. The computational count and number of global synchronization points per iteration for both methods without preconditioning are shown in Table 1.

Table 1. The amount of calculation per iteration

Method	**Vector update**	**Inner product**	**Matrix vector**	**Global synchro.**
BiCOR	2	2	4	2
PBiCOR	2	8	3	1

From Table 1, the readers can see that, compared with the BiCOR method, the global synchronization points per iteration of the PBiCOR method have been reduced from 2 to 1. **Similar idea with Gu et al. (2007)**, we give a performance analysis of both methods on distributed memory parallel computers, in which each processor involves memory model and corresponding operation units connected by network. All operation units execute the same program, i.e., Single-Program and Multi-Data (SPMD) model. If one of processors needs data from other processors, message passing must be performed. The following denotations will be used as follows: P is the number of processors; N is the total number of unknowns; n_z is the average number of nonzero elements per row in the matrix A; t_{fl} is the average time for a double precision floating point operation; t_s denotes the communication start-up time; t_w is the transmission time of a word between two neighboring processors.

Since the computational and communication patterns are the same per iteration, we only consider the time complexity of parallel computation and communication of one iteration. For a vector update (daxpy) or an inner product (ddot), the computation time is given by $2t_{fl}N/P$, where N/P is the local number of unknowns on a processor. The computation time of matrix-vector multiplication is $(2n_z-1)t_{fl}N/P$.

Consider a mesh-based processor grid with P processors, and assume that the communication is carried out through binary tree structure. Then, the global accumulation and broadcast time for one inner product is $2\log P(t_s + t_w)$, while the global accumulation and broadcast time for k simultaneous inner products is $2\log P(t_s + kt_w)$. The authors have assumed that coefficient matrix are mapped to processors such that for the matrix-vector a processor needs only to communicate with the nearest neighbor processors. The communication for the matrix-vector product is necessary for the exchange of so-called boundary data: sending boundary data to other processor and receiving boundary data from other processors. Assume that each processor has to send and receive n_m messages and the number of boundary data elements on a processor is given by n_b. Therefore, the total number of words that have to be communicated (sent and received) is then $2(2n_b + n_m)$ per processor. For both methods, the communication time of one matrix-vector product is $2n_mt_s + 2(2n_b + n_m)t_w$.

n summary, the time of a vector update is, since it needs no communication, that

$$t_{vec_upd} = 2t_{fl}N/P.$$

the time for k simultaneous inner products that need only one global communication is

$$t_{inn_prod}(k) = 2kt_{fl}N/P + 2\log P(t_s + kt_w),$$

and the time for a matrix-vector product is

$$t_{mat_vec} = (2n_z - 1)t_{fl}N/P + 2n_m t_s + 2(2n_b + n_m)t_w.$$

So the time per iteration of the BiCOR method is

$$\begin{aligned} T_{BiCOR} &= 2t_{vec_upd} + 2t_{inn_prod}(1) + 4t_{mat_vec} \\ &= (8n_z + 4)t_{fl}N/P + 4\log P(t_s + t_w) + 8n_m t_s + 8(2n_b + n_m)t_w \end{aligned}.$$

and of the PBiCOR method is

$$\begin{aligned} T_{PBiCOR} &= 2t_{vec_upd} + t_{inn_prod}(8) + 3t_{mat_vec} \\ &= (6n_z + 17)t_{fl}N/P + 2\log P(t_s + 8t_w) + 6n_m t_s + 6(2n_b + n_m)t_w \end{aligned}.$$

The readers know that $t_s \gg t_w$ holds for massively distributed parallel computers. Comparing T_{BiCOR} and T_{PBiCOR}, the authors get that the parallelism of the PGGl-CGS2 method is better than that of the BiCOR method since $T_{BiCOR} > T_{PBiCOR}$.

Minimizing T_{BiCOR} and T_{PBiCOR} from the above two equations, the author obtain that the number of processors for minimal parallel time of both methods is

$$P_{BiCOR} = \frac{(8n_z + 4)t_{fl}N\ln 2}{4(t_s + t_w)} = \frac{(2n_z + \frac{1}{2})t_{fl}N\ln 2}{t_s + t_w}$$

and

$$P_{PBiCOR} = \frac{(6n_z + 17)t_{fl}N\ln 2}{2(t_s + 2t_w)}) = \frac{(3n_z + \frac{17}{2})t_{fl}N\ln 2}{t_s + t_w},$$

respectively. Since $t_s \gg t_w$, therefore inequality

$$\frac{P_{PBiCOR}}{P_{BiCOR}} \approx \frac{3n_z + \frac{17}{2}}{2n_z + \frac{1}{2}} > \frac{3}{2}$$

is satisfied for any $n_z > 0$. It shows that the PBiCOR method has better scalability than the BiCOR method.

The improved ratio for the P BiCOR method against the BiCOR method is

$$\eta = \frac{T_{BiCOR} - T_{PBiCOR}}{T_{BiCOR}} \approx \frac{2t_s P\log P - (2n_z - 13)t_{fl}N}{4t_s P\log P + (8n_z + 4)t_{fl}N} \to 50\%$$

for $t_s \gg t_w$, when N is fixed and P is large enough.

4. Issoefficiency analysis about two methods

Since it is not known in advance how many iteration steps a method needs to converge, we do not consider the whole algorithm until its termination but consider a single iteration step and take N, the dimension of the coefficient matrix, as the problem size. The readers know that the sequential execution time is usually expressed as a function of problem size. So the execution time of the fastest known

sequential algorithm to perform a single BiCOR-like iteration is

$$T_{seq}(N) = cNt_{fl} = \Theta(N) \tag{1}$$

where c is a constant.

For the motivation of the isoefficiency concept, we briefly state the conventional definitions of speedup and efficiency as given in [1]. The speedup S is defined as the ratio of the time to solve a problem on a single processor using the fastest known sequential algorithm to the time required to solve the same problem on a parallel computer, $S = T_{seq}/T_{par}$. The efficiency E is defined as the ratio of the speedup to the number of processors, $E = S/P$.

One can expect to keep efficiency constant by allowing T_{seq} to grow properly with increasing number of processors. The rate at which T_{seq} has to be increased with respect to the number of processors P to maintain a fixed efficiency can serve as a measure of scalability.

Algorithm implementations on real parallel computers do not achieve optimal speedup. For example, data communication delays and synchronization are reasons for nonoptimal speedup. All causes of dropping the theoretically ideal speedup are call overhead and the total overhead function is formally defined as

$$T_{over} = (O,P) = PT_{par}(N,P) - T_{seq}(N). \tag{2}$$

i.e. the part of the total time spent in solving a problem summed over all processors PT_{par} that is not incurred by the fastest known sequential algorithm T_{seq}. So, the efficiency can be expressed as a function of the total overhead and the execution time of the fastest known sequential algorithm

$$E = \frac{S}{P} = \frac{T_{seq}(N)}{PT_{par}(N,P)} = \frac{T_{seq}(N)}{T_{seq} + T_{over}(N,P)} = \frac{1}{1 + T_{over}(N,P)/T_{seq}(N)}.$$

The rate with respect to P at which T_{seq} has to be increased to keep efficiency constant is used to asses the quality of a scalable parallel system. For example, if T_{seq} has to be increased as an exponential function of P to maintain efficiency fixed, the system is poorly scalable. A system is highly scalable if one only has to linearly increase T_{seq} with respect to P. Such growth rates can be calculated from (**??**)1) or from

$$T_{seq}(N) = \frac{E}{1-E} T_{over}(N,P). \tag{3}$$

The authors now carried out the isoefficiency analysis for a single BiCOR-like iteration step. To calculate growth rates from (**??**)3), we need to know T_{seq} and T_{over} of a single BiCOR-like iteration step. The total execution time of the fastest known sequential algorithm is given by (**??**)3). The total overhead is solely due to communication times, i.e. $T_{over} = P_{comm}$.

From Table 1 we can get the sequential time, parallel communication time and

total overhead for BiCOR method per iteration as follows, respectively

$$T^{seq}_{BiCOR} = 2t^{comp}_{vec-upd} + 2t^{comp}_{inn-prod}(1) + 4t^{comp}_{mat-vec} = (8n_z + 4)t_{fl}N,$$
$$T^{comm}_{BiCOR} = 2t^{comm}_{inn-prod}(1) + 4t^{comp}_{mat-vec} = 2\log P(t_s + t_w) + 8n_m t_s + 8(2n_b + n_m)t_w.$$
$$T^{over}_{BiCOR} = PT^{par}_{BiCOR} - T^{seq}_{BiCOR} = PT^{comm}_{BiCOR}.$$

Substitute the above equation into (**??**), we can get following equation

$$T^{seq}_{BiCOR} = \frac{E}{1-E}T^{over}_{BiCOR},$$
$$N_{BiCOR} = \frac{(t_s+t_w)E}{t_{fl}(2n_z+1)(1-E)}P\log P \approx \frac{t_s E}{t_{fl}(2n_z+1)(1-E)}P\log P.$$

The authors can also get the sequential time, parallel communication time and total overhead for PBiCOR method per iteration as follows

$$T^{seq}_{PBiCOR} = 2t^{comp}_{vec-upd} + t^{comp}_{inn-prod}(8) + 3t^{comp}_{mat-vec} = (6n_z + 17)t_{fl}N,$$
$$T^{comm}_{PBiCOR} = t^{comm}_{inn-prod}(8) + 3t^{comp}_{mat-vec} = 2\log P(t_s + t_w) + 8n_m t_s + 8(2n_b + n_m)t_w.$$
$$T^{over}_{PBiCOR} = PT^{par}_{PBiCOR} - T^{seq}_{BiCOR} = PT^{comm}_{BiCOR}.$$

Substitute the above equation into (**??**), we obtain

$$T^{seq}_{PBiCOR} = \frac{E}{1-E}T^{over}_{BiCOR},$$
$$N_{PBiCOR} = \frac{(t_s+t_w)E}{t_{fl}(3n_z+\frac{17}{2})(1-E)}P\log P \approx \frac{t_s E}{t_{fl}(3n_z+\frac{17}{2})(1-E)}P\log P.$$

From the forms of N_{BiCOR} and N_{PBiCOR}, we can see that PBiCOR method has better scalability than PBiCOR method and the parallel performance can be improved by a factor of about 2. For different values of the efficiency, the results are shown in Fig. (**??**)1) and Fig. (**??**)2), where the filled curves represent the theoretically derived isoefficiency function $N \sim O(P\log P)$, in which $t_s = 100\mu s, t_w = 20ns, t_{fl} = 10ns$ and $n_z = 5$ for our distributed memory parallel computer we do our numerical experiments on.

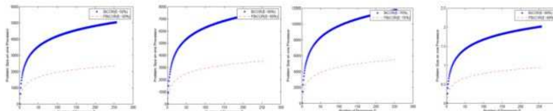

Fig. 1. Comparison of isoefficiency curve of BiCOR and PBiCOR

5. Conclusion

For further performance improvement, one can consider overlap useful computation with communication.

References

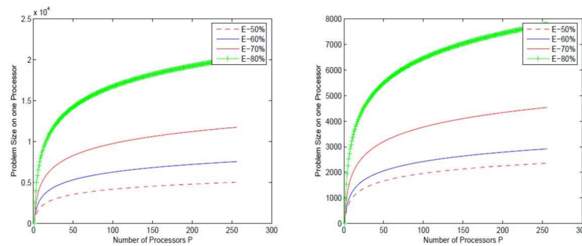

Fig. 2. Isoefficiency curve of BiCOR(left) and PBiCOR(right)

[1] A. Grama, A. Gupta, V. Kumar: *Isoefficiency function: a scalability metric for parallel algorithms and architectures.* IEEE parallel distributed technology *1* (1993), No. 3, 12-21.

[2] L. H. Chi, J. Liu, X. P. Liu, Q. F. Hu, X. M. Li: *An improved conjugate residual algorithm for large symmetric linear systems.* In Computational Physics, Proceedings of the Joint Conference of ICCP6 and CCP2003, Rinton Press, New Jersey, USA (2005), 325-328.

[3] X. P. Liu, T. X. Gu, X. D. Hang, Z. Q. Sheng: *A parallel version of QMRCGSTAB method for large linear systems in distributed parallel environments.* Applied Mathematics and Computation *172* (2006), No. 2, 744-752.

[4] Y. Saad: *Iterative methods for sparse linear systems.* PWS Publishing Company, Boston (2004).

[5] E. de Sturler: *A performance model for Krylov subspace methods on mesh-based parallel computers.* Parallel Computing *22* (1996), 57-74.

[6] L. T. Zhang, X. Y. Zuo, T. X. Gu, T. Z. Huang: *Conjugate residual squared method and its improvement for non-symmetric linear systems.* International Journal of Computer Mathematics *87* (2010),No. 7,1578-1590.

[7] L. T. Zhang, T. Z. Huang, T. X. Gu, X. Y. Zuo: *An improved conjugate residual squared algorithm suitable for distributed parallel computing.* Microelectronics and Computer *25* (2008), No. 10, 12-14.

[8] J. H. Zhang, H. Dai, J. Zhao: *Generalized global conjugate gradient squared algorithm.* Applied Mathematics and Computation*216* (2010), 326-329.

Received November 16, 2017