Advances in Engineering Software 147 (2020) 102837

Contents lists available at ScienceDirect

Advances in Engineering Software

journal homepage: www.elsevier.com/locate/advengsoft

Research paper

Reducing the effect of global synchronization in delayed gradient methods for symmetric linear systems

Qinmeng Zou[a,b], Frédéric Magoulès*[,b,c]

[a] School of Science, Beijing University of Posts and Telecommunications, Beijing 100876, China
[b] Université Paris-Saclay, CentraleSupélec, Mathématiques et Informatique pour la Complexité et les Systèmes, 91190, Gif-sur-Yvette, France
[c] Faculty of Engineering and Information Technology, University of Pécs, Boszorkány street 2, Pécs 7624, Hungary

ARTICLE INFO

Keywords:
Lagged gradient methods
Reducing synchronization
S-dimensional gradient methods
Cyclic gradient methods
Parallel computing
Asynchronous iterations

MSC:
65F10
68Q85

ABSTRACT

Compared with arithmetic operation, communication cost is often the bottleneck on modern computers, and thus should be paid increasing attention when choosing algorithms. Lagged gradient methods are known for their error tolerance and fast convergence. However, it appears that their parallel behavior is not well understood. In this paper, we explore the cyclic formulations of lagged gradient methods and *s*-dimensional methods for reducing global synchronizations. We provide parallel implementations for these methods and propose some new variants. A comparison is then reported for different gradient iterative schemes. To illustrate the performance, we run a number of experiments, from which we conclude that our formulations perform better than traditional methods in view of both iteration count and computing time.

1. Introduction

Real-life industrial applications often lead to large-scale linear systems, for which parallel iterative solvers have been studied for decades. The performance of a parallel algorithm is impacted by both floating point operations and communications performed during its execution. On modern computers, however, communication is generally much slower than computation, and this trend is likely to accelerate in future systems.

One way to remedy this issue is to recast parallel synchronous algorithms into asynchronous formulations by breaking up the data dependencies, in which iterations and communications are no longer synchronized. Asynchronous iterations were formally proposed by Chazan and Miranker [1] and generalized by many later authors. Recently, much interest has been focused on programming libraries [2,3], GPU implementations [4,5], space domain decomposition methods [6,7] and time domain decomposition methods [8,9]. Another direction to reduce communication is based on the *s*-step Krylov methods, which can often reduce the amount of communications per iteration by a factor of $O(s)$ (see, e.g., [10]). Chronopoulos and Gear [11] suggested an early specimen for the *s*-step formulation of the conjugate gradient (CG) method [12]. Hoemmen [10] gave a complete treatise of the relevant theory and proposed several new methods, which were later generalized by Carson et al. [13]. There are other strategies for reducing communication in iterative methods that can be found in much recent literature; see, e.g. [14,15].

Krylov subspace methods are often the methods of choice for large practical problems. For example, CG is optimal for symmetric positive definite (SPD) systems and has finite termination property in exact arithmetic (see, e.g., [16]). Nevertheless, CG is sensitive to rounding errors, and thus any derivation such as small initial perturbation in the Hessian matrix can seriously degrade its performance. An alternative is to use lagged gradient methods. They are more resilient to variations than the traditional CG method and tend to perform better when low precision is required (see, e.g., [17–19]). Barzilai and Borwein [20] proposed the first lagged gradient method. Friedlander et al. [17] discussed a general framework and introduced a cyclic formulation. In this paper, we suggest some variants of gradient methods for the purpose of reducing synchronization cost, for which parallel algorithms are provided by complying the message passing interface (MPI) specification. Our work closely follows that of a conference paper [21]. However, it additionally includes much useful information: (i) complete discussion about an alignment method; (ii) parallel implementation of a new cyclic method; (iii) damped formulation of a cyclic method; (iv) comparison of lagged and asynchronous gradient schemes; (v) discussion about the equilibration technique; (vi) more practical considerations and

* Corresponding author.
E-mail address: frederic.magoules@hotmail.com (F. Magoulès).

https://doi.org/10.1016/j.advengsoft.2020.102837
Received 14 November 2019; Received in revised form 9 May 2020; Accepted 26 May 2020

experimental results.

In Section 2 we give some parallel algorithms of lagged gradient methods. In Section 3 we discuss the s-dimensional formulation and give a parallel implementation. In Section 4 we propose some promising methods in terms of communication cost. Section 5 focuses on the comparison of the parallel lagged gradient scheme and the asynchronous gradient scheme. Section 6 provides some details about implementation and performance, as well as a technique for improving the conditioning of intermediate matrices generated by s-dimensional methods. Numerical results are presented in Section 7 and concluding remarks are drawn in Section 8.

2. Lagged gradient methods

We consider the linear system

$$Ax = b, \tag{1}$$

where A is an SPD real matrix of size N. This is equivalent to minimizing the quadratic function

$$f(x) = \frac{1}{2}x^{\intercal}Ax - b^{\intercal}x. \tag{2}$$

Several gradient-based iterative techniques have been proposed to solve such system. We shall denote by g_n the gradient vector at nth iteration such that $g_n = Ax_n - b$. The standard gradient iteration is as follows:

$$x_{n+1} = x_n - \alpha_n g_n, \tag{3}$$

where α_n is the step defined by a specific method. The gradient vector may also be updated by

$$g_{n+1} = g_n - \alpha_n A g_n. \tag{4}$$

Given an initial vector x_0, the gradient methods generate a sequence $\{x_n\}$ converging to the exact solution x_*.

The steepest descent (SD) method is a simple gradient method introduced by Cauchy [22]

$$\alpha_n^{\mathrm{SD}} = \frac{g_n^{\intercal}g_n}{g_n^{\intercal}Ag_n},$$

which minimizes the function value in (2) or the A-norm error $\|e_{n+1}\|_A = \|x_* - x_{n+1}\|_A$. In practice, the stagnation of convergence is a serious concern, and this led to the work of Akaike [23] to explain the convergence behavior. Let κ be the condition number of matrix A. The convergence rate can be established by

$$\|e_{n+1}\|_A \leq \left(\frac{\kappa - 1}{\kappa + 1}\right)\|e_n\|_A.$$

It was shown, by a nonstandard analysis based on the transformation of probability distribution, that the SD directions tend to asymptotically alternate between two orthogonal vectors. As a result, the asymptotic convergence rate is almost as bad as predicted [23].

In 1988, Barzilai and Borwein [20] suggested the first lagged gradient method that was motivated by the observation that the quasi-Newton property can be exploited by the two-point approximation. This yields the steplength

$$\alpha_n^{\mathrm{BB}} = \frac{g_{n-1}^{\intercal}g_{n-1}}{g_{n-1}^{\intercal}Ag_{n-1}}.$$

The global convergence has been proved in Raydan [24] by an induction. Numerical results show that the directions generated by BB span the whole space, i.e., it does not sink into the lower subspace spanned by some eigenvectors and reduce the gradient components more or less at the same asymptotic rate (see, e.g., [25]).

Concerning numerical implementation, one observes that the above methods have similar operations: matrix-vector multiplications, dot products, and vector updates. In parallel environment, dot products require a combination of local dot products and a reduction over the processors, which incur a costly global synchronization. Let ν be the number of processors and $i \in \{1, \cdots, \nu\}$. Let $x_{i,n}$ and $g_{i,n}$ be the subvectors updated in the ith processors. Assume that the matrix A and the vector b are partitioned into the submatrices A_i and the subvectors b_i, respectively, where A_i are non-overlapping blocks of rows of A. Assume that $x_0 = 0$. This yields the parallel SD method described in Algorithm 1. The potential bottleneck is in the reduction and the gather operations. Note that if the sparse matrix A has a suitable sparsity structure, one may exploit the specific topology and use neighborhood collective operations to accelerate the algorithm. The parallel BB method can be similarly described. If one combines SD and BB, an alternate step (AS) method can be derived (see [26]).

1: set $g_{i,0} = -b_i$
2: $\gamma = g_{i,0}^{\intercal}g_{i,0}$
3: `Allreduce`(γ, SUM)
4: compute initial residual
5: **for** $n = 0, 1, \ldots$ **do**
6: `Allgather`(g_n)
7: $p_{i,n} = A_i g_n$
8: $\delta = p_{i,n}^{\intercal}g_{i,n}$
9: `Allreduce`(δ, SUM)
10: $\alpha_n = \gamma/\delta$
11: $x_{i,n+1} = x_{i,n} - \alpha_n g_{i,n}$
12: $g_{i,n+1} = g_{i,n} - \alpha_n p_{i,n}$
13: $\gamma = g_{i,n+1}^{\intercal}g_{i,n+1}$
14: `Allreduce`(γ, SUM)
15: compute residual
16: **end for**

Algorithm 1. Parallel SD method.

A famous publication by Friedlander et al. [17] considered a framework called gradient methods with retards (GMR) and proved its global convergence. Let m be a positive integer. Let $\bar{n} = \max\{0,\ n - m\}$. This framework is of the form

$$\alpha_n^{\mathrm{GMR}} = \frac{g_{\tau(n)}^{\intercal}A^{\rho(n)}g_{\tau(n)}}{g_{\tau(n)}^{\intercal}A^{\rho(n)+1}g_{\tau(n)}}, \tag{5}$$

with

$$\tau(n) \in \{\bar{n},\ \bar{n}+1, \cdots, n-1,\ n\}, \quad \rho(n) \in \{q_1, \cdots, q_m\}, \quad q_j \geq 0.$$

Notice that choosing $\tau(n) = n$ and $\rho(n) = 0$ yields SD, while $\tau(n) = n - 1$ yields BB. Assume that m is the maximum delay. Let $d = m + 1$. A synchronization-reducing gradient method can be derived from this framework by periodically fixing and recovering $\tau(n) = n$ within d iterations, leading to the cyclic steepest descent (CSD) method

$$\alpha_n^{\mathrm{CSD}} = \alpha_{\tau(n)}^{\mathrm{SD}}, \quad \tau(n) = \max\{j \leq n : j \bmod d = 0\},$$

with $d \geq 1$. AS is indeed a special case of CSD when choosing $d = 2$. Further experiments in Friedlander et al. [17] confirmed that CSD is competitive with BB in the sequential case. The parallel implementation is described in Algorithm 2. Notice that $\tau(n) = n - r$, which indicates that the value of steplength does not change within a cycle. In the parallel case, CSD reduces $d - 1$ or $2(d - 1)$ dot product operations in every d iterations depending on the implementation. If one computes only local residual in each iteration, then the algorithm could be further improved by moving line 14 into the code block starting from line 5 in Algorithm 2.

Let $\{v_1, \cdots, v_N\}$ be the orthonormal eigenvectors of A associated with the eigenvalues $\{\lambda_1, \cdots, \lambda_N\}$. We assume that

$$0 < \lambda_1 \leq \cdots \leq \lambda_N,$$

and thus $\kappa = \lambda_N/\lambda_1$. There exist real values $\zeta_{j,n}$ such that

```
same as lines 1 to 4 in Algorithm 1
for n = 0, 1, ... do
    Allgather(g_n)
    p_{i,n} = A_i g_n
    if n mod d = 0 then
        δ = p_{i,n}^T g_{i,n}
        Allreduce(δ, SUM)
        α_n = γ/δ
        r = 0
    end if
    x_{i,n+1} = x_{i,n} − α_{n−r} g_{i,n}
    g_{i,n+1} = g_{i,n} − α_{n−r} p_{i,n}
    r = r + 1
    γ = g_{i,n+1}^T g_{i,n+1}
    Allreduce(γ, SUM)
    compute residual
end for
```

Algorithm 2. Parallel CSD method.

$$g_n = \sum_{j=1}^{N} \zeta_{j,n} v_j.$$

Using Eq. (4), we get

$$\zeta_{j,n+1} = (1 - \alpha_n \lambda_j)\zeta_{j,n} = \zeta_{j,0} \prod_{k=0}^{n} (1 - \alpha_k \lambda_j), \tag{6}$$

where $j \in \{1, \cdots, N\}$. It is clear that choosing $\alpha_n = 1/\lambda_j$ leads to the fact that the corresponding component of the gradient vector will vanish and remain zero throughout the iteration.

It was De Asmundis et al. [27] who suggested to use the second order information based on some spectral properties of SD to accelerate the gradient iterations. They proposed in [27] and [28] two new methods called SD with alignment (SDA) and SD with constant steplength (SDC), respectively. We only discuss the latter since they share similar properties, and it is the experience of the present authors that SDC generally performs better. The step is of the form

$$\alpha_n^{\mathrm{SDC}} = \begin{cases} \alpha_n^{\mathrm{SD}}, & n \bmod (d_1 + d_2) < d_1, \\ \alpha_{\tau(n)}^{\mathrm{Y}}, & \tau(n) = \max\{j \le n : j \bmod (d_1 + d_2) = d_1\}, \end{cases}$$

where

$$\alpha_n^{\mathrm{Y}} = 2\left(\sqrt{\left(\frac{1}{\alpha_{n-1}^{\mathrm{SD}}} - \frac{1}{\alpha_n^{\mathrm{SD}}}\right)^2 + \frac{4\,\|g_n\|^2}{(\alpha_{n-1}^{\mathrm{SD}})^2 \|g_{n-1}\|^2}} + \frac{1}{\alpha_{n-1}^{\mathrm{SD}}} + \frac{1}{\alpha_n^{\mathrm{SD}}}\right)^{-1},$$

with $d_1, d_2 \ge 1$. Here, the Yuan step was introduced in Yuan [29]. De Asmundis et al. [28] has derived the following limit:

$$\lim_{n\to\infty} \alpha_n^{\mathrm{Y}} = \frac{1}{\lambda_N},$$

which is based on the spectral property of SD; see [23]. According to the relationship (6), one finds that the main feature of SDC is to foster the reduction of gradient components along the eigenvectors of A selectively, and thus reduce the search space into smaller and smaller dimensions. As a result, the problem tends to have a better and better condition number (see [28]). The parallel algorithm is illustrated in Algorithm 3. In parallel case, this method reduces $d_2 - 1$ or $2(d_2 - 1)$ dot product operations in every $d_1 + d_2$ iterations.

One finds that both CSD and SDC are cyclic gradient methods. A cycle equals d iterations in CSD and $d_1 + d_2$ in SDC. It is also possible to define minimal gradient (MG) approaches (see, e.g., [30]) in a cyclic form. We will not investigate these variants further since MG and SD share similar properties in both sequential and parallel views; see [31,32].

3. *s*-dimensional steepest descent

Now we rewrite the solution vector recurrence in the following form:

$$x_{n+1} = x_n - P(A)g_n.$$

By the Cayley-Hamilton theorem, one finds that A^{-1} could be described as a matrix polynomial of degree $t - 1$. Let $P(A)$ be a polynomial of the form

$$P(A) = \alpha_n^{(1)} I + \alpha_n^{(2)} A + \cdots + \alpha_n^{(t)} A^{t-1},$$

where I is an identity matrix of size N. We consider the s-dimensional plane

$$L_n^{(s)} = \left\{ x_n - \sum_{j=1}^{s} \alpha_n^{(j)} A^{j-1} g_n : \alpha_n^{(j)} \in \mathbb{R} \right\}. \tag{7}$$

We remark that choosing $s = 1$ will recover the search spaces of gradient methods. The s-dimensional steepest descent (s-SD) method is of the form

$$x_{n+1} = x_n - \alpha_n^{(1)} g_n - \cdots - \alpha_n^{(s)} A^{s-1} g_n,$$

where $\alpha_n^{(1)}, \cdots, \alpha_n^{(s)}$ are selected to minimize the quadratic function in Eq. (2) or the A-norm error $\|e_{n+1}\|_A$ over $L_n^{(s)}$. The recurrence of gradient vector can be written as

$$g_{n+1} = g_n - \alpha_n^{(1)} A g_n - \cdots - \alpha_n^{(s)} A^s g_n. \tag{8}$$

It is clear that g_{n+1} should be orthogonal to the s-dimensional space $L_n^{(s)}$, namely, g_{n+1} should be orthogonal to g_n, Ag_n, ..., $A^{s-1}g_n$. Along with (8), as a result, one finds the following system of equations:

$$\begin{aligned}
\alpha_n^{(1)} g_n^{\intercal} A g_n &+ \cdots + \alpha_n^{(s)} g_n^{\intercal} A^s g_n &&= g_n^{\intercal} g_n, \\
\alpha_n^{(1)} (Ag_n)^{\intercal} A g_n &+ \cdots + \alpha_n^{(s)} (Ag_n)^{\intercal} A^s g_n &&= (Ag_n)^{\intercal} g_n, \\
&\vdots \qquad\qquad\quad \vdots && \vdots \\
\alpha_n^{(1)} (A^{s-1} g_n)^{\intercal} A g_n &+ \cdots + \alpha_n^{(s)} (A^{s-1} g_n)^{\intercal} A^s g_n &&= (A^{s-1} g_n)^{\intercal} g_n.
\end{aligned}$$

Let $\omega_n^{(j)} = g_n^{\intercal} A^j g_n$. Since $(A^k g_n)^{\intercal} A^l g_n = g_n^{\intercal} A^{k+l} g_n$, it follows that

$$\begin{pmatrix} \omega_n^{(1)} & \omega_n^{(2)} & \cdots & \omega_n^{(s)} \\ \omega_n^{(2)} & \omega_n^{(3)} & \iddots & \omega_n^{(s+1)} \\ \vdots & \iddots & \iddots & \vdots \\ \omega_n^{(s)} & \omega_n^{(s+1)} & \cdots & \omega_n^{(2s-1)} \end{pmatrix} \begin{pmatrix} \alpha_n^{(1)} \\ \alpha_n^{(2)} \\ \vdots \\ \alpha_n^{(s)} \end{pmatrix} = \begin{pmatrix} \omega_n^{(0)} \\ \omega_n^{(1)} \\ \vdots \\ \omega_n^{(s-1)} \end{pmatrix}. \tag{9}$$

The coefficient matrix in the above system of equations is denoted by Ω_n, which is a Hankel matrix and, specifically, a Gram matrix.

Some early works of the s-dimensional steepest descent (s-SD) method can be found in Birman [33], Khabaza [34] and Forsythe [35]. In 1950, Birman [33] gave a proof for the convergence by virtue of the orthogonal polynomial. The original literature was written in Russian; see Equation (2.14) in [35] for an English discussion, where the asymptotic behavior is also shown therein. The following theorem gives a lower bound for the parameter which has the largest magnitude, acting as a complement to the existing theories.

Theorem 1. *Consider the linear system $Ax = b$ where A is SPD. Let $\Gamma_n = \{\alpha_n^{(1)}, \cdots, \alpha_n^{(s)}\}$. If the sequence of solution vectors $\{x_n\}$ is generated by the s-dimensional SD method, then*

$$\max_{\alpha \in \Gamma_n} |\alpha| \ge \frac{1}{\lambda_N + \cdots + \lambda_N^s}$$

Proof. Let

same as lines 1 to 4 in Algorithm 1
for $n = 0, 1, \ldots$ **do**
  `Allgather`(g_n)
  $p_{i,n} = A_i g_n$
  if $n \bmod (d_1 + d_2) < d_1$ **then**
    $\delta = p_{i,n}^{\mathsf{T}} g_{i,n}$
    `Allreduce`$(\delta,$ `SUM`$)$
    $\alpha_n = \gamma/\delta$
    $\theta = \gamma$
    $r = 0$
  else if $n \bmod (d_1 + d_2) = d_1$ **then**
    $\delta = p_{i,n}^{\mathsf{T}} g_{i,n}$
    `Allreduce`$(\delta,$ `SUM`$)$
    $\hat{\alpha} = \gamma/\delta$
    $\alpha_n = 2/(\sqrt{(1/\alpha_{n-1} - 1/\hat{\alpha})^2 + 4\gamma/(\alpha_{n-1}^2\theta)} + 1/\alpha_{n-1} + 1/\hat{\alpha})$
    $r = 0$
  end if
  same as lines 11 to 16 in Algorithm 2
end for

Algorithm 3. Parallel SDC method.

$\hat{\alpha} = \max_{\alpha \in \Gamma_n} |\alpha|$.

By assumption,

$$\omega_n^{(0)} = \omega_n^{(1)}\alpha_n^{(1)} + \cdots + \omega_n^{(s)}\alpha_n^{(s)} \le \omega_n^{(1)}\hat{\alpha} + \cdots + \omega_n^{(s)}\hat{\alpha}.$$

Since

$$\frac{\omega_n^{(j)}}{\omega_n^{(0)}} = \frac{\omega_n^{(j)}}{\omega_n^{(j-1)}} \cdots \frac{\omega_n^{(1)}}{\omega_n^{(0)}} \le \lambda_N^j,$$

we get

$$1 \le \lambda_N\hat{\alpha} + \cdots + \lambda_N^s\hat{\alpha},$$

which yields the desired result. □

Let $u_n^{(j)} = A^{(2j-1)/2} g_n$. One finds that

$$\Omega_n = \begin{pmatrix} (u_n^{(1)})^{\mathsf{T}} u_n^{(1)} & (u_n^{(1)})^{\mathsf{T}} u_n^{(2)} & \cdots & (u_n^{(1)})^{\mathsf{T}} u_n^{(s)} \\ (u_n^{(2)})^{\mathsf{T}} u_n^{(1)} & (u_n^{(2)})^{\mathsf{T}} u_n^{(2)} & \ddots & (u_n^{(2)})^{\mathsf{T}} u_n^{(s)} \\ \vdots & \ddots & \ddots & \vdots \\ (u_n^{(s)})^{\mathsf{T}} u_n^{(1)} & (u_n^{(s)})^{\mathsf{T}} u_n^{(2)} & \cdots & (u_n^{(s)})^{\mathsf{T}} u_n^{(s)} \end{pmatrix}.$$

Therefore, Ω_n is a Gram matrix, and thus a positive definite Hankel matrix, which can be solved by Cholesky factorization. This variant involves the computational work of $4sN$ multiplications, $4sN$ additions, s matrix-vector operations and the cost of solving a positive definite Hankel system, for which the Cholesky factorization requires approximately $(1/3)s^3 + (1/2)s^2$ arithmetic operations. On the other hand, a simple SD algorithm over s iterations requires $4sN$ multiplications, $4sN$ additions, s matrix-vector operations and s divisions. It is well-known that the Hankel matrix is ill-conditioned since the condition number has a lower bound $\kappa \ge 3 \cdot 2^{s-6}$ (see, e.g., [36]). In the finite precision case, the monomial basis may become linearly dependent asymptotically when s is large. As a result, the ill-conditioning of Ω_n makes the s-SD algorithm unstable. Concerning parallel implementation, however, breaking the data dependency between matrix-vector multiplications and dot products makes the s-dimensional method attractive. Now we design the parallel s-SD method; see Algorithm 4. In the parallel case, s-SD requires 2 reduction operations and s gather operations, while SD requires $2s$ reduction operations and s gather operations over s iterations. Under the reasonable assumption that the computational environment is latency-bound, the redundant operations of s-SD will not be a limiting factor. If the matrix has an appropriate sparsity pattern, the neighborhood collective operation will be much cheaper to apply than the global reduction operation. Additionally, since $N \gg s$ for large sparse matrix A, the computational cost of Cholesky factorization is unlikely to be a bottleneck.

set $g_{i,0} = -b_i$
$\omega_0^{(0)} = g_{i,0}^{\mathsf{T}} g_{i,0}$
`Allreduce`$(\omega_0^{(0)},$ `SUM`$)$
compute initial residual
for $n = 0, 1, \ldots$ **do**
  set $u_{i,n} = g_{i,n}$
  for $j = 1, \ldots, s$ **do**
    `Allgather`(u_n)
    $p_{i,n}^{(j)} = A_i u_n$
    set $u_{i,n} = p_{i,n}^{(j)}$
  end for
  compute $\omega_n^{(1)}, \ldots, \omega_n^{(2s-1)}$
  `Allreduce`$([\omega_n^{(1)} \ldots \omega_n^{(2s-1)}],$ `SUM`$)$
  solve system (9)
  $x_{i,n+1} = x_{i,n} - \alpha_n^{(1)} g_{i,n} - \alpha_n^{(2)} p_{i,n}^{(1)} - \cdots - \alpha_n^{(s)} p_{i,n}^{(s-1)}$
  $g_{i,n+1} = g_{i,n} - \alpha_n^{(1)} p_{i,n}^{(1)} - \alpha_n^{(2)} p_{i,n}^{(2)} - \cdots - \alpha_n^{(s)} p_{i,n}^{(s)}$
  $\omega_{n+1}^{(0)} = g_{i,n+1}^{\mathsf{T}} g_{i,n+1}$
  `Allreduce`$(\omega_{n+1}^{(0)},$ `SUM`$)$
  compute residual
end for

Algorithm 4. Parallel s-SD method.

4. New lagged methods

According to Friedlander et al. [17], the gradient methods with retards usually perform much better than the traditional optimal gradient methods. We have the following observations:

- in Forsythe [35], we know that both SD and s-SD have two-step invariance property, which makes the directions sink into lower subspaces, while lagged gradient steps could remedy such problem (see, e.g., [25]);
- unlike CG, gradient methods are less sensitive to the rounding error since no sequence of conjugate directions is constructed;
- both computation and communication for dot products can be cyclically curtailed by imposing lagged steps.

It may be possible to add a slight delay in successive s-SD iterations, from which the induced rounding error is generally acceptable. Recall that a scalar matrix is a diagonal matrix where all elements are equal. The following theorem reveals the signs of parameters in 2-dimensional case.

Theorem 2. *Consider the linear system $Ax = b$ where A is SPD. Assume that g_n is not an eigenvector of A. If the sequence of solution vectors $\{x_n\}$ is generated by the 2-dimensional SD method, then $\alpha_n^{(1)} > 0$ and $\alpha_n^{(2)} < 0$.*

Proof. The system (9) is reduced to

$$\begin{pmatrix} \omega_n^{(1)} & \omega_n^{(2)} \\ \omega_n^{(2)} & \omega_n^{(3)} \end{pmatrix} \begin{pmatrix} \alpha_n^{(1)} \\ \alpha_n^{(2)} \end{pmatrix} = \begin{pmatrix} \omega_n^{(0)} \\ \omega_n^{(1)} \end{pmatrix}.$$

It follows that

$$\alpha_n^{(1)} = \frac{\omega_n^{(0)}\omega_n^{(3)} - \omega_n^{(1)}\omega_n^{(2)}}{\omega_n^{(1)}\omega_n^{(3)} - (\omega_n^{(2)})^2}.$$

$$\alpha_n^{(2)} = \frac{(\omega_n^{(1)})^2 - \omega_n^{(0)}\omega_n^{(2)}}{\omega_n^{(1)}\omega_n^{(3)} - (\omega_n^{(2)})^2}.$$

By Cauchy-Schwarz inequality, we observe that

$$\begin{aligned} \frac{g_n^{\mathrm{T}} A^{j+1} g_n}{g_n^{\mathrm{T}} A^{j+2} g_n} &\leq \frac{\| A^{j/2} g_n \| \| A^{(j+2)/2} g_n \|}{\| A^{(j+2)/2} g_n \|^2} \\ &= \frac{\| A^{j/2} g_n \|^2}{\| A^{(j+2)/2} g_n \| \| A^{j/2} g_n \|} \leq \frac{g_n^{\mathrm{T}} A^{j} g_n}{g_n^{\mathrm{T}} A^{j+1} g_n}. \end{aligned} \tag{10}$$

Moreover, by assumption we know that g_n is not an eigenvector of A, yielding that $A^{j/2}g_n$ and $A^{(j+2)/2}g_n$ are linearly independent. Thus, the strict inequality holds.

Now consider the denominators

$$\omega_n^{(1)}\omega_n^{(3)} = \frac{\omega_n^{(1)}}{\omega_n^{(2)}} \cdot \frac{\omega_n^{(2)}}{\omega_n^{(3)}} \cdot (\omega_n^{(3)})^2 > \left(\frac{\omega_n^{(2)}}{\omega_n^{(3)}}\right)^2 \cdot (\omega_n^{(3)})^2 = (\omega_n^{(2)})^2.$$

Similarly, consider the numerators

$$\omega_n^{(0)}\omega_n^{(3)} = \frac{\omega_n^{(0)}}{\omega_n^{(1)}} \cdot \omega_n^{(1)}\omega_n^{(2)} \cdot \frac{\omega_n^{(3)}}{\omega_n^{(2)}} > \frac{\omega_n^{(2)}}{\omega_n^{(3)}} \cdot \frac{\omega_n^{(3)}}{\omega_n^{(2)}} \cdot \omega_n^{(1)}\omega_n^{(2)} = \omega_n^{(1)}\omega_n^{(2)},$$

and

$$\omega_n^{(0)}\omega_n^{(2)} > (\omega_n^{(1)})^2.$$

This completes our proof. □

From Theorem 2, one finds that the components obtained by system (9) may be less than zero, and thus could not serve as lagged steps. Another direction of approach is based on the steps of the traditional gradient methods, fortunately, the parameters $\omega_n^{(j)}$ being available. The real scalars

$$\hat{\alpha}_n = \frac{\omega_{n-r}^{(j)}}{\omega_{n-r}^{(j+1)}} = \frac{g_{n-r}^{\mathrm{T}} A^{j} g_{n-r}}{g_{n-r}^{\mathrm{T}} A^{j+1} g_{n-r}}$$

can be constructed where $1 \leq r < d$ and $0 \leq j < 2s-1$, and embedded in the iterative process. It is noteworthy that such kind of steps is the inverse of Rayleigh quotient and satisfies the form of Eq. (5). Like the previously described methods, we illustrate the parallel implementation of this variant in Algorithm 5, called cyclic s-SD (Cs-SD) method. We

1: same as lines 1 to 4 in Algorithm 4
2: **for** $n = 0, 1, \ldots$ **do**
3: **if** $n \bmod d = 0$ **then**
4: same as lines 6 to 16 in Algorithm 4
5: $r = 1$
6: **else**
7: `Allgather`(g_n)
8: $p_{i,n} = A_i g_n$
9: compute $\hat{\alpha}_n$ by $\omega_{n-r}^{(1)}, \ldots, \omega_{n-r}^{(2s-1)}$
10: $x_{i,n+1} = x_{i,n} - \hat{\alpha}_n g_{i,n}$
11: $g_{i,n+1} = g_{i,n} - \hat{\alpha}_n p_{i,n}$
12: $r = r + 1$
13: **end if**
14: same as lines 17 to 19 in Algorithm 4
15: **end for**

Algorithm 5. Parallel Cs-SD method.

observe that a cycle involves d iterations in which an s-SD process is performed at the beginning of the cycle and $d-1$ lagged gradient iterations are performed thereafter.

There exist several versions of the algorithm that can be effectively used for updating the solution and the gradient vectors due to the redundant elements of $\omega_n^{(j)}$. We give examples for the CSD-like version and the damped version. The CSD-like step is defined by choosing $j = 0$ for all $r \in \{1, \cdots, d-1\}$, which is equivalent to the CSD method except when $n \bmod d = 0$. The damped step means that the steplengths become smaller and smaller during lagged iterations. From Eq. (10) and by the fact that g_n is not an eigenvector of A,

$$\frac{\omega_n^{(2s-2)}}{\omega_n^{(2s-1)}} < \cdots < \frac{\omega_n^{(0)}}{\omega_n^{(1)}},$$

and thus choosing $j = r-1$ yields the damped Cs-SD method.

In average, the computational work in each iteration consists of $2(d+1)sN/d$ multiplications, $2(d+1)sN/d$ additions, s matrix-vector products and $O(s^3/d)$ operations to solve the positive definite Hankel system. The construction of scalars in lagged iterations also brings the computational cost of divisions. Nevertheless, we are more concerned with the communication cost in the parallel case, which on average consists of $(d+1)/d$ reduction operations and $(s+d-1)/d$ gather operations. We remark that $d = 1$ leads to exactly the case of s-SD in terms of the communication cost. If we move the line 14 into the code block starting from line 3, merge the reduction operations and compute local residual instead in Algorithm 5, then Cs-SD requires $1/d$ reduction operations and $(s+d-1)/d$ gather operations, while 1 and s operations for s-SD, respectively.

It is challenging to calculate the potential speedup. For example, under the assumption that both reduction and gather operations have the same latency, which is assumed as the major bottleneck that impacts the performance of algorithms, then we expect to see a $(sd+d)/(s+d)$ times less communication of Algorithm 5 with respect to Algorithm 4 when computing local residual per iteration. The drawback of this analysis is that an s-SD iteration is generally not equivalent to a lagged step iteration. A lagged iteration usually gives some orders of magnitude more contributions to the convergence speed than an s-SD iteration when n is large. On the other hand, the comparison of classical gradient methods is obvious. Here, we only discuss the most communication-hiding situation. If we merge the reduction operations, then both SD and BB require 1 reduction operations and 1 gather operation. For the CSD method, on average, the communication cost consists of $1/d$ reduction operations and 1 gather operation. Under the above assumption, choosing $d = 2$ yields a 1.3 times less communication.

Similar to classical CSD, classical SDC generates cyclic iterations

1: same as lines 1 to 4 in Algorithm 4
2: **for** $n = 0, 1, \ldots$ **do**
3: **if** $n \bmod d = 0$ **then**
4: same as lines 6 to 16 in Algorithm 4
5: $\tilde{\alpha} = \omega_n^{(0)}/\omega_n^{(1)}$
6: $\theta = \omega_n^{(0)}$
7: **else**
8: $\texttt{Allgather}(g_n)$
9: $p_{i,n} = A_i g_n$
10: **if** $n \bmod d = 1$ **then**
11: $\delta = p_{i,n}^{\intercal} g_{i,n}$
12: $\texttt{Allreduce}(\delta, \texttt{SUM})$
13: $\hat{\alpha} = \omega_n^{(0)}/\delta$
14: $\alpha_n = 2/(\sqrt{(1/\tilde{\alpha} - 1/\hat{\alpha})^2 + 4\omega_n^{(0)}/(\tilde{\alpha}^2\theta)} + 1/\tilde{\alpha} + 1/\hat{\alpha})$
15: $r = 0$
16: **end if**
17: $x_{i,n+1} = x_{i,n} - \alpha_{n-r} g_{i,n}$
18: $g_{i,n+1} = g_{i,n} - \alpha_{n-r} p_{i,n}$
19: $r = r + 1$
20: **end if**
21: same as lines 17 to 19 in Algorithm 4
22: **end for**

Algorithm 6. Parallel s-SDC method.

which allow a reduction in communication cost. The following inequality is easy to show:

$$\alpha_n^{\mathrm{Y}} < \min\{\alpha_{n-1}^{\mathrm{SD}}, \alpha_n^{\mathrm{SD}}\}.$$

This leads to a rather smooth convergence. The idea of the s-SD with constant steplength (s-SDC) method can be interpreted as a process in which s-SD takes place at the beginning of a cycle, and then one proceeds with the cyclic Yuan step over $d - 1$ iterations. We illustrated the parallel algorithm of s-SDC in Algorithm 6. An obvious drawback of this method is that one more dot product operation is required over a cycle. This will increase the average cost in terms of both computation and communication. Since Yuan step plays an important role in the alignment methods, we expect to see that the superior convergence behavior can remedy the additional cost per cycle.

Now we make the following simplifying assumptions about algorithms:

- communication is the major bottleneck compared with computation, in which latency plays a dominant role compared with bandwidth;
- we count reduction and gather operations as yielding the same communication cost;
- we do not use blocking synchronization for the computation of residual;
- $1/s$ iteration in s-SD process is regarded as equivalent to 1 iteration in SD or CSD process.

Although the methods discussed above are pairwise distinct in a mathematical point of view, these assumptions enable a comparison of them, which is shown in Table 1. Since we do not exploit sparsity pattern in matrix A, we assume that each processor performs a gather operation in each step explicitly. Concerning reduction operations, both parameters s and d lead to gains in communication avoidance. We observe that SDC is special because it was motivated by the spectral properties of SD, involving d_1 iterations to foster alignment (see [28]), and thus has a parameter in the numerator. It appears that this approach could be more efficient, albeit at the cost of increasing communication with respect to CSD.

Table 1
Average number of reduction and gather operations (ro and go). 1 iteration here equals 1 iteration in CSD process or $1/s$ iteration in s-SD process.

	SD	s-SD	CSD	SDC	Cs-SD	s-SDC
ro	1	$1/s$	$1/d$	$(d_1+1)/(d_1+d_2)$	$1/(s+d-1)$	$2/(s+d-1)$
go	1	1	1	1	1	1

1: compute $A_i g_n$
2: compute and synchronize coefficients
3: compute α_n
4: compute $x_{i,n+1}$, $g_{i,n+1}$
5: **for** $j \in$ dependent neighbors **do**
6: send g_i to processor j
7: **end for**
8: **for** $j \in$ essential neighbors **do**
9: receive g_j from processor j
10: **end for**
11: compute residual
12: wait for requests to finish

Algorithm 7. Parallel gradient scheme.

5. Comparison of parallel gradient schemes

The straightforward parallel gradient scheme, illustrated in Algorithm 7, consists of two global synchronizations: one before the calculation of α_n and the other for g_n. Here, processor j is a dependent neighbor of processor i if the next computation in j depends on g_i; similarly, processor j is an essential neighbor of processor i if the next computation in i depends on g_j. The coefficients mentioned in line 2 are the necessary components for computing α_n. For the SD method, as shown in Algorithm 1, this line is equivalent to computing $(A_i g_n)^{\intercal} g_{i,n}$

1: compute $A_i g_n$
2: **if** n mod $d = 0$ **then**
3: compute and synchronize coefficients
4: compute α_n
5: **end if**
6: compute $x_{i,n+1}$, $g_{i,n+1}$
7: **for** $j \in$ dependent neighbors **do**
8: send g_i to processor j
9: **end for**
10: **for** $j \in$ essential neighbors **do**
11: receive g_j from processor j
12: **end for**
13: compute residual
14: wait for requests to finish

Algorithm 8. Parallel cyclic gradient scheme.

followed by a reduction operation. Unlike previous algorithms, we show here explicitly the nonblocking version, resulting in limited performance gains by overlapping communication with computation. It is known that the data dependencies in gradient methods preclude further improvement of performance. The parallel cyclic gradient scheme is shown in Algorithm 8. This is a synchronization-reducing strategy which allows an $O(d)$ reduction in communication cost. It can also accelerate convergence with respect to the number of iterations, although the corresponding methods do not satisfy optimality properties. In practice, such scheme often results in drastic improvements in convergence compared to the classical methods like SD.

If we remove lines 2 and 3 in Algorithm 7 or lines 2 to 5 in Algorithm 5, i.e., a constant parameter $\hat{\alpha} = \alpha_n$ is used for updating $x_{i,n+1}$ and $g_{i,n+1}$, then we get the parallel constant gradient scheme. The sequential case was first considered by Richardson [37] in a more general form, and thus could be called stationary Richardson iteration (see, e.g., [38]). This scheme can be further improved by removing the waiting statement, yielding the so-called asynchronous gradient scheme, as described in Algorithm 9. Asynchronous iterations were first proposed by Chazan and Miranker [1]. Let $\mathcal{F}_i$ be a mapping in processor i. Assume that $\mathcal{P}_n \subset \{1, \cdots, \nu\}$ is a subset of processors in the nth iteration. Such iterative scheme can be expressed in the following form:

$$x_{i,n+1} = \begin{cases} \mathcal{F}_i(x_{1,\tau_{i,1}(n)}, \cdots, x_{\nu,\tau_{i,\nu}(n)}), & i \in \mathcal{P}_n, \\ x_{i,n}, & \text{otherwise}, \end{cases}$$

where $\tau_{i,j}(n)$ is a positive integer satisfying $\tau_{i,j}(n) \leq n$ for each element j in each processor i. It avoids communication by breaking up the data dependencies in iterative methods but still allows updates when messages arrive. In order to hide global synchronization, a well-designed non-blocking technique is required to evaluate residual; see [39] for a discussion.

Although the asynchronous gradient scheme can minimize waiting time among ν processors, the convergence conditions may be more stringent than those for the lagged gradient scheme. The basic

1: compute $A_i g_n$
2: compute $x_{i,n+1}$, $g_{i,n+1}$
3: **for** $j \in$ dependent neighbors **do**
4: send g_i to processor j
5: **end for**
6: **for** $j \in$ essential neighbors **do**
7: receive g_j from processor j
8: **end for**
9: compute residual

Algorithm 9. Asynchronous gradient scheme.

conditions for A at the beginning of Section 2 should suffice for the latter in exact arithmetic, while the condition $\rho(|I - \hat{\alpha}A|) < 1$ is required for the former (see Section 6.3.2 in [40] for a discussion). Another drawback is that the asynchronous formulation is based on the stationary Richardson method, which is generally far less efficient than the lagged gradient methods. Hoemmen [10] argued that asynchronous methods tend to be based on slow relaxation-type iterations. In other words, synchronous advanced methods tend to perform better than asynchronous basic methods. However, as mentioned in Section 1, asynchronous iterations are still viewed as promising, especially when communication costs become prohibitive. We will not pursue this scheme further in this paper. For a good overview, see [39].

6. Some practical considerations

The algorithms mentioned in Sections 2 to 5 are described in an MPI-like manner. If the matrix size is not a power of two, then one should invoke `Allgatherv`, a routine defined in the MPI specification for assembling data with displacement of indices, to collect matrix-vector multiplication results. For the global synchronization before α_n, it appears that the use of nonblocking communication does not lead to performance gains due to the data dependencies. If the MPI programming libraries are optimally implemented, then the collective routine `Allreduce` is somewhat preferred to the point-to-point operations. Thakur et al. [41] has shown how to improve the performance of collective communication operations. On the other hand, 2 reduction operations result in 2 times latency costs, and thus could be avoided by constructing 1 message for 2 values. The termination detection may become complicated. For example, global convergence could be evaluated by a nonblocking reduction of the local residual. In this case, however, a delayed snapshot of global state without redundant solution vector being stored may lead to a suspicion about the final residual, as well as a delay in the termination detection. A more robust solution is based on the blocking reduction of residual required by α_j where $j > n$. One could certainly make a snapshot of solution vector at the cost of increasing computer storage. For the gather operation, overlapping is an effective technique for improving performance.

A concern with lagged gradient methods is the stability. The choice of d between 1 and 2 can be viewed as a compromise between stability and convergence speed. Choosing larger d may however yield no gain in convergence speed but a heavy price in instability, which may decrease the maximum attainable accuracy. For the CSD-like methods, from the experience of the present authors, choosing $d > 5$ can occasionally lead to serious accuracy problems when $\kappa = 10^4$. For SDC, however, it depends on the choice of both d_1 and d_2. Choosing $d_1 = 1$ and $d_2 > 8$ sometimes yields an oscillation or a divergence, while choosing $d_1 = 5$ and $d_2 = 15$ seems to be stable when $\kappa = 10^4$. Raydan and Svaiter [42] studied the relaxed SD step and observed its effectiveness. Yuan [43] suggested that a good gradient method should use at least one SD step in every few iterations. van den Doel and Ascher [19] argued that faster gradient methods should use occasionally larger steplengths. It appears that SDC satisfies all the arguments given above.

Another serious concern is the conditioning of Hankel matrices. As mentioned in Section 3, Hankel matrices are ill-conditioned even for small orders. Hoemmen [10] discussed a technique called equilibration which entails applying a transformation $\hat{A} = D^{-1/2}AD^{-1/2}$, where the diagonal elements of D equal the diagonal elements of A. This technique is indeed a special case of preconditioning and has been successfully applied to the communication-avoiding Krylov subspace methods (see, e.g., [10,13]). We will discuss the impact of equilibration for Hankel matrices in the next section. On the other hand, there exist faster solvers for Hankel systems with a complexity of $O(s^2)$ compared with $O(s^3)$ for the Cholesky factorization (see, e.g., [44]). Nevertheless, the limiting behavior ignores the importance of large constant terms. Since s must be chosen small enough to maintain the conditioning, a

traditional $O(s^3)$ solver seems to be preferred. Additionally, advanced methods are more complex to implement.

7. Numerical experiments

The goal of this section is to show the numerical behaviors of the methods discussed in this paper. Among the experiments we have run, we would like to illustrate a few that are representative. Concerning parallel experiments, we have implemented the algorithms by using Alinea [45], an advanced linear algebra library developed for massively parallel computations, on a cluster of nodes comprising Intel Xeon CPU E5-2670 v3 2.30 GHz connected with FDR Infiniband network 56 Gbit/s. The MPI environment is supported by SGI MPT 2.12. Other experiments have been performed in MATLAB R2018b.

Some of the linear systems that we attempt to solve are randomly generated by MATLAB, while others are collected from the University of Florida Sparse Matrix Collection [46]. In all tests, the elements of initial vector x_0 are randomly selected from $(-1, 1)$. The right-hand side is fixed with $b = 0$ and thus $x_* = 0$. The iteration is stopped whenever $\|g_n\| < 10^{-6}\|g_0\|$ and, for the sake of simplicity, the blocking `Allreduce` operation is employed for computing the global residual.

In the first test, we give a general comparison of the aforementioned methods, shown in Fig. 1. The first plot (top-left) shows that SD and s-SD converge much slower than other methods. As expected, monotone methods are not so effective, even for a moderately well-conditioned system. The second plot (top-right) gives an example for nonmonotone methods. Along with other results, we conclude that SDC is a robust and fast solver, while s-SDC is generally less efficient. For CSD and Cs-SD, we performed more tests with different parameters (e.g., the third and the fourth plots) but could not come to a general conclusion. What seems to be clear is that lagged gradient methods are highly sensitive to initial conditions, of which a fair comparison could not be outlined by several curves.

Table 2
Average results among 10 tests with 16 and 32 processors. The matrix is ill-conditioned with $N = 10848$. The threshold of iteration is 800 and the stopping criterion is $\|g_n\| < 10^{-6}\|g_0\|$.

	16 processors			32 processors		
method	iter	time(s)	residual	iter	time(s)	residual
SD	800	0.616	5.6×10^{-5}	800	0.462	5.6×10^{-5}
BB	800	0.515	2.8×10^{-6}	800	0.364	4.6×10^{-5}
CSD(4)	800	0.468	3.2×10^{-4}	800	0.311	2.5×10^{-2}
SDC(4,4)	800	0.507	1.1×10^{-5}	800	0.350	2.9×10^{-6}
C5-SD(4)	800	0.495	6.1×10^{-6}	800	0.341	4.4×10^{-6}

In Tables 2 to 5, we perform a set of parallel experiments for two structural problems. The first one is ill-conditioned with $N = 10848$ and $\kappa = 9.967 \times 10^9$. The matrix name is `msc10848` and the matrix ID is 361. The second one has a larger size $N = 141347$. The matrix name is `bmw7st_1` and the matrix ID is 1253. The condition number of this matrix is $\kappa = 2.570 \times 10^{18}$, which is much more ill-conditioned than the first one. However, we will see later that this problem is easier to solve

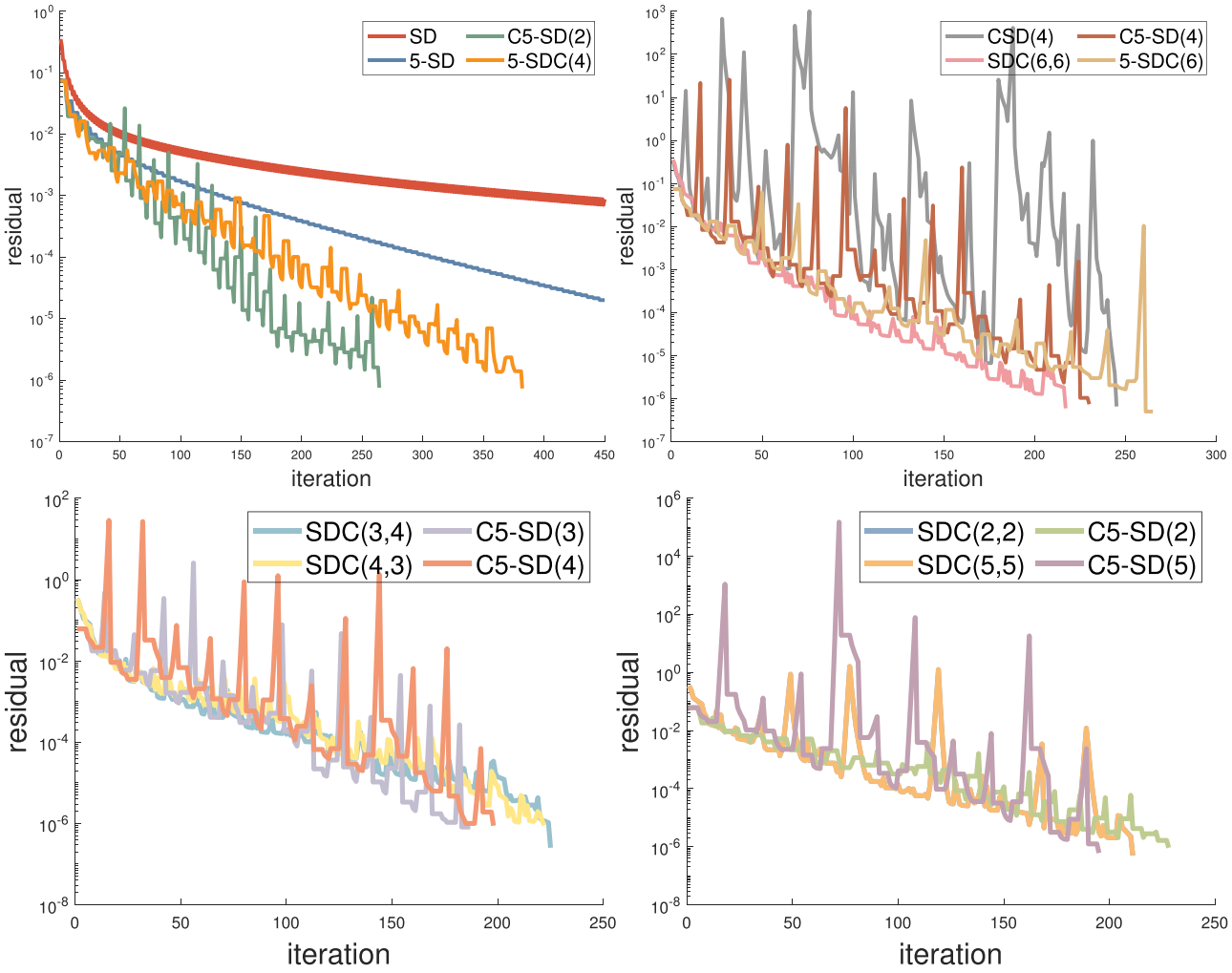

Fig. 1. Experiments with random matrices: $N = 10^2$, $\kappa = 10^3$ (top), $N = 10^2$, $\kappa = 10^2$ (bottom).

Table 3
Average results among 10 tests with 64 and 128 processors. The matrix is ill-conditioned with $N = 10848$. The threshold of iteration is 800 and the stopping criterion is $\|g_n\| < 10^{-6}\|g_0\|$.

	64 processors			128 processors		
method	iter	time(s)	residual	iter	time(s)	residual
SD	800	0.402	5.6×10^{-5}	800	0.408	5.6×10^{-5}
BB	800	0.304	1.1×10^{-3}	800	0.305	1.8×10^{-4}
CSD(4)	800	0.256	4.2×10^{-3}	800	0.252	5.4×10^{-6}
SDC(4,4)	800	0.301	2.2×10^{-6}	800	0.301	2.3×10^{-6}
C5-SD(4)	800	0.298	1.8×10^{-4}	800	0.300	8.1×10^{-5}

Table 4
Average results among 10 tests with 16 and 32 processors. The matrix is larger with $N = 141347$. The threshold of iteration is 10^4 and the stopping criterion is $\|g_n\| < 10^{-6}\|g_0\|$.

	16 processors			32 processors		
method	iter	time(s)	residual	iter	time(s)	residual
SD	$> 10^4$	\	\	$> 10^4$	\	\
BB	567	2.354	8.3×10^{-7}	1034	3.168	9.5×10^{-7}
CSD(4)	385	1.355	3.4×10^{-7}	365	0.850	1.1×10^{-6}
SDC(4,4)	468	1.927	8.7×10^{-7}	378	1.144	8.4×10^{-7}
C5-SD(4)	509	2.163	7.8×10^{-7}	405	1.153	9.9×10^{-7}

Table 5
Average results among 10 tests with 64 and 128 processors. The matrix is larger with $N = 141347$. The threshold of iteration is 10^4 and the stopping criterion is $\|g_n\| < 10^{-6}\|g_0\|$.

	64 processors			128 processors		
method	iter	time(s)	residual	iter	time(s)	residual
SD	$> 10^4$	\	\	$> 10^4$	\	\
BB	779	1.903	8.0×10^{-7}	788	1.723	7.1×10^{-7}
CSD(4)	590	1.017	9.4×10^{-7}	378	0.523	5.8×10^{-6}
SDC(4,4)	421	1.016	8.6×10^{-7}	301	0.652	9.9×10^{-7}
C5-SD(4)	605	1.392	9.0×10^{-7}	525	1.119	9.2×10^{-7}

Table 6
Average results among 10 tests with 64 and 128 processors. The matrix is very large with $N = 1564794$. The threshold of iteration is 10^4 and the stopping criterion is $\|g_n\| < 10^{-6}\|g_0\|$.

	64 processors			128 processors		
method	iter	time(s)	residual	iter	time(s)	residual
BB	720	23.649	7.9×10^{-7}	625	16.246	8.9×10^{-7}
CSD(4)	818	19.672	8.5×10^{-7}	709	13.594	8.4×10^{-7}
SDC(4,4)	453	14.503	9.4×10^{-7}	445	11.796	9.1×10^{-7}
C5-SD(4)	768	22.925	9.2×10^{-7}	545	14.002	9.7×10^{-7}

due to the choice of initial vectors and the distribution of eigenvalues. We can see in Tables 2 and 3 that in all cases actual convergence does not occur within 800 iterations. By fixing the number of iterations, we observe that the lagged strategy can significantly reduce the communication and computation costs. In Tables 4 and 5, most of the procedures are stopped before reaching the specified iteration limit. Although BB is substantially faster than SD, cyclic formulations are generally superior to BB in terms of the convergence behavior. It

Table 7
Average results among 10 tests with 256 and 512 processors. The matrix is very large with $N = 1564794$. The threshold of iteration is 10^4 and the stopping criterion is $\|g_n\| < 10^{-6}\|g_0\|$.

	256 processors			512 processors		
method	iter	time(s)	residual	iter	time(s)	residual
BB	601	14.179	9.4×10^{-7}	762	18.017	8.0×10^{-7}
CSD(4)	569	9.591	8.9×10^{-7}	598	9.844	8.0×10^{-7}
SDC(4,4)	429	10.505	8.9×10^{-7}	429	10.269	9.8×10^{-7}
C5-SD(4)	671	14.998	8.0×10^{-7}	700	15.522	9.9×10^{-7}

appears that Cs-SD in Tables 4 and 5 requires more iterations than other competing methods. However, Cs-SD is still better than BB, both from the point of view of convergence speed and robustness. In Tables 2 and 3, we find that Cs-SD and SDC are indistinguishable in terms of computing time. Additionally, SDC can ensure a good balance between efficiency and stability, while CSD requires less arithmetic operations but may decrease accuracy. Each of the two has its own dynamics and we could not provide a general recommendation. It is important to note that SD is much more robust than the nonmonotone methods in terms of residual, while CSD seems to be the most unstable one from Tables 4 and 5, in which the gap of iteration count among different situations is fairly large.

The next test proceeds along the same lines but considers a large-scale problem. We draw another matrix from the University of Florida Sparse Matrix Collection with $N = 1564794$ and $\kappa = 1.225 \times 10^8$, which is obtained from a 3D mechanical problem and discretized with the finite element method. The matrix name is `Flan_1565` and the matrix ID is 2544. We perform several experiments on a cluster using 64, 128, 256 and 512 cores. Tables 6 and 7 illustrate the results. Note that we will not show the results of SD since it requires always more than 10^4 iterations. From these tables, we find that SDC is still efficient in the large-scale case, while BB generally takes a long time to finish the job. It is noteworthy that there is almost no gain in using 512 cores compared with the case of 256 cores. This may be due to the communication costs and the limit of problem size. As mentioned in Section 6, lagged strategy enhances convergence speed at the expense of stability, namely, a small numerical perturbation may lead to large difference in numbers of iterations. From this point of view, SDC is more preferable than other alternatives.

One may wonder if Cs-SD could be further improved. One possible way is the damped formulation as mentioned in Section 4. It is often observed that the iteration count of damped Cs-SD is smaller than the original method. Two examples are shown in Fig. 2, in which we use two 100×100 random matrices with $\kappa = 10^3$ and $\kappa = 10^4$, respectively. Note that the opposite results could also be obtained, depending on the initial condition and the implementation. In contrast, if we only select the smallest possible step $\omega_n^{(2s-2)}/\omega_n^{(2s-1)}$, then a worse convergence result will generally be obtained.

In the next case, we discuss the use of equilibration for managing the conditioning of Hankel matrices. Numerical experiments in Fig. 3 indicate a marked improvement. The matrix size is 7102 with a condition number of 1.6×10^4. The left plot shows how the condition number changes over iteration, in which a two-step invariance property can be clearly confirmed. The right plot shows the changes of conditioning over s. For each s, we choose the 49th and the 50th Hankel matrices and calculate their condition numbers. The larger one is illustrated in the right plot. We observe that the equilibration technique improves the quality of Hankel matrices.

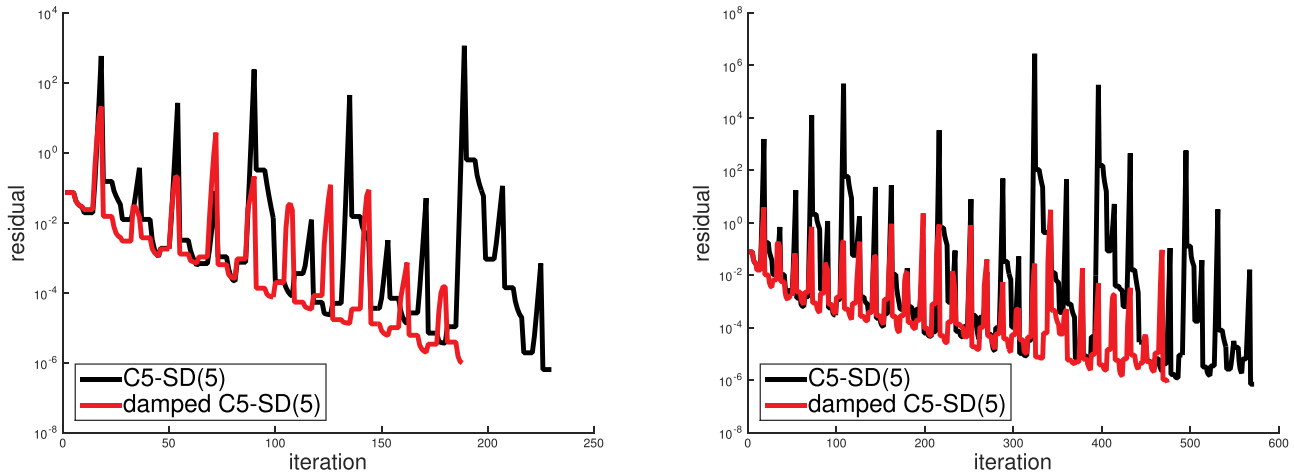

Fig. 2. A comparison of C*s*-SD and damped C*s*-SD with $s = 5$ and $d = 5$ for random problems: $N = 10^2$, $\kappa = 10^3$ (left), $N = 10^2$, $\kappa = 10^4$ (right).

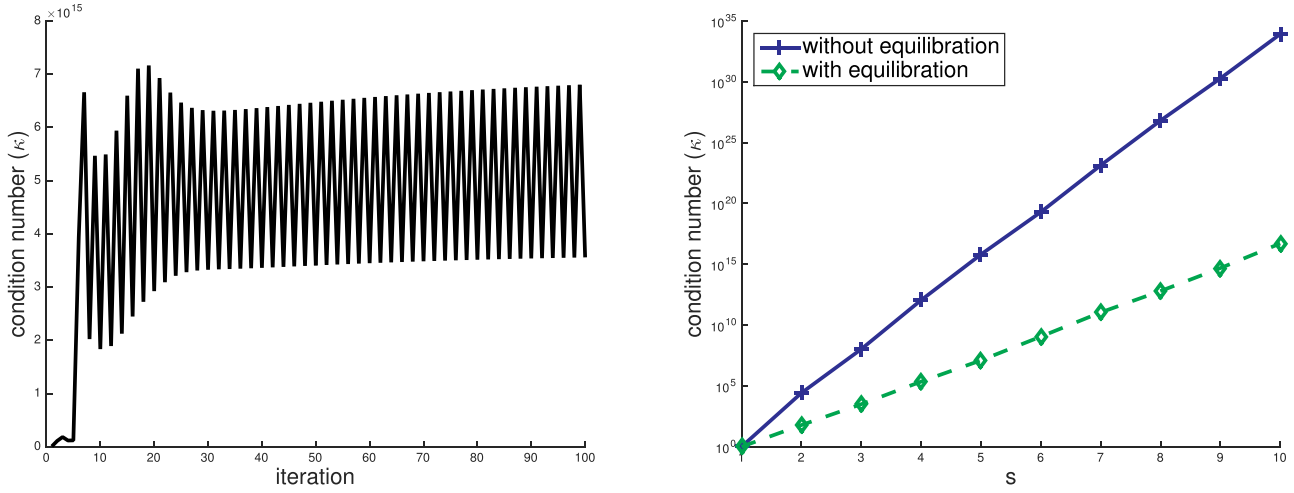

Fig. 3. Experiments for the conditioning of Hankel systems. The matrix is generated from a structural problem with $N = 7102$ and $\kappa = 1.6 \times 10^4$. The left plot shows the condition number when $s = 5$. The right plot shows the impact of equilibration.

8. Conclusion

Lagged gradient method is a useful tool for large-scale linear systems. For example, they can sometimes be competitive with the conjugate gradient method [31] and are appealing when initial perturbations occur in the original matrices. Concerning parallel performance, SDC is still viewed as the most robust method. The behavior of our new cyclic formulation C*s*-SD is indistinguishable from that of CSD. On the other hand, SD, *s*-SD, BB and *s*-SDC are generally less or far less efficient than these methods. In the largest test case reported in the preceding section, the best performance was obtained by taking 128 or 256 processors, while there is no benefit from using 512 processors. The damped formulation of C*s*-SD and the equilibration technique could be used with success in most cases. We have also shown parallel algorithms of all these methods, as well as some simplified schemes for the purpose of comparison. Future areas of exploration include comparing parallel lagged gradient methods with communication-avoiding Krylov subspace methods. It will also be interesting to exploit sparsity pattern in the coefficient matrix for parallel gradient methods.

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgements

This work was supported by the French national programme LEFE/INSU and the project ADOM (Méthodes de décomposition de domaine asynchrones) of the French National Research Agency (ANR).

Supplementary material

Supplementary material associated with this article can be found, in the online version, at 10.1016/j.advengsoft.2020.102837

References

[1] Chazan D, Miranker W. Chaotic relaxation. Linear Algebra Appl 1969;2(2):199–222.
[2] Magoulès F, Gbikpi-Benissan G. JACK: An asynchronous communication kernel library for iterative algorithms. J Supercomput 2017;73(8):3468–87.
[3] Magoulès F, Gbikpi-Benissan G. JACK2: an MPI-based communication library with non-blocking synchronization for asynchronous iterations. Adv Eng Softw 2018;119:116–33.
[4] Anzt H, Tomov S, Dongarra JJ, Heuveline V. A block-asynchronous relaxation method for graphics processing units. J Parallel Distrib Comput 2013;73(12):1613–26.
[5] Chow E, Anzt H, Dongarra JJ. Asynchronous iterative algorithm for computing incomplete factorizations on GPUs. High performance computing: 30th international conference, ISC high performance 2015. Springer; 2015. p. 1–16.
[6] Magoulès F, Venet C. Asynchronous iterative sub-structuring methods. Math Comput Simul 2018;145:34–49.
[7] Magoulès F, Szyld DB, Venet C. Asynchronous optimized schwarz methods with and without overlap. Numer Math 2017;137(1):199–227.

[8] Magoulès F, Gbikpi-Benissan G, Zou Q. Asynchronous iterations of parareal algorithm for option pricing models. Mathematics 2018;6(4):1–18.
[9] Magoulès F, Gbikpi-Benissan G. Asynchronous parareal time discretization for partial differential equations. SIAM J Sci Comput 2018;40(6):C704–25.
[10] Hoemmen M. Communication-avoiding Krylov subspace methods. UC Berkeley; 2010. Ph.D. thesis.
[11] Chronopoulos AT, Gear CW. *s*-step iterative methods for symmetric linear systems. J Comput Appl Math 1989;25(2):153–68.
[12] Hestenes MR, Stiefel E. Methods of conjugate gradients for solving linear systems. J Res Natl Bur Stand 1952;49(6):409–36.
[13] Carson EC, Knight N, Demmel JW. Avoiding communication in nonsymmetric Lanczos-based Krylov subspace methods. SIAM J Sci Comput 2013;35(5):S42–61.
[14] Ghysels P, Ashby TJ, Meerbergen K, Vanroose W. Hiding global communication latency in the GMRES algorithm on massively parallel machines. SIAM J Sci Comput 2013;35(1):C48–71.
[15] Grigori L, Moufawad S, Nataf F. Enlarged Krylov subspace conjugate gradient methods for reducing communication. SIAM J Matrix Anal Appl 2016;37(2):744–73.
[16] Saad Y. Iterative methods for sparse linear systems. 2nd SIAM; 2003.
[17] Friedlander A, Martínez JM, Molina B, Raydan M. Gradient method with retards and generalizations. SIAM J Numer Anal 1999;36(1):275–89.
[18] Dai Y-H, Fletcher R. On the asymptotic behaviour of some new gradient methods. Math Program 2005;103(3):541–59.
[19] van den Doel K, Ascher UM. The chaotic nature of faster gradient descent methods. J Sci Comput 2012;51(3):560–81.
[20] Barzilai J, Borwein JM. Two-point step size gradient methods. IMA J Numer Anal 1988;8(1):141–8.
[21] Zou Q, Magoulès F. Parallel iterative methods with retards for linear systems. In: Iványi P, Topping BHV, editors. Proceedings of the sixth international conference on parallel, distributed, grid and cloud computing for engineering Stirlingshire, UK: Civil-Comp Press; 2019. https://doi.org/10.4203/ccp.112.31.
[22] Cauchy AL. Méthode générale pour la résolution des systèmes d'équations simultanées. Comp Rend Sci Paris 1847;25(1):536–8. (in French)
[23] Akaike H. On a successive transformation of probability distribution and its application to the analysis of the optimum gradient method. Ann Inst Stat Math 1959;11(1):1–16.
[24] Raydan M. On the Barzilai and Borwein choice of steplength for the gradient method. IMA J Numer Anal 1993;13(3):321–6.
[25] Dai Y-H, Yuan Y-X. Analysis of monotone gradient methods. J Ind Manag Optim 2005;1(2):181–92.
[26] Dai Y-H. Alternate step gradient method. Optimization 2003;52(4–5):395–415.
[27] De Asmundis R, di Serafino D, Riccio F, Toraldo G. On spectral properties of steepest descent methods. IMA J Numer Anal 2013;33(4):1416–35.
[28] De Asmundis R, di Serafino D, Hager WW, Toraldo G, Zhang H. An efficient gradient method using the Yuan steplength. Comput Optim Appl 2014;59(3):541–63.
[29] Yuan Y-X. A new stepsize for the steepest descent method. J Comput Math 2006;24(2):149–56.
[30] Dai Y-H, Yuan Y-X. Alternate minimization gradient method. IMA J Numer Anal 2003;23(3):377–93.
[31] Zou Q., Magoulès F.. Fast gradient methods with alignment for symmetric linear systems without using Cauchy step. J Comput Appl Math. In press.
[32] Zou Q., Magoulès F.. Parameter estimation in the Hermitian and skew-Hermitian splitting method using gradient iterations. Numer Linear Algebra Appl. In press.
[33] Birman MS. Some estimates for the method of steepest descent. Uspekhi Mat Nauk 1950;5(3)(37):152–5. (in Russian)
[34] Khabaza IM. An iterative least-square method suitable for solving large sparse matrices. Comput J 1963;6(2):202–6.
[35] Forsythe GE. On the asymptotic directions of the *s*-dimensional optimum gradient method. Numer Math 1968;11(1):57–76.
[36] Tyrtyshnikov EE. How bad are Hankel matrices? Numer Math 1994;67(2):261–9.
[37] Richardson LF. The approximate arithmetical solution by finite differences of physical problems involving differential equations, with an application to the stresses in a masonry dam. Philos Trans R Soc A 1911;210(459–470):307–57.
[38] Young DM. Iterative solution of large linear systems. Academic Press; 1971.
[39] Magoulès F, Gbikpi-Benissan G. Distributed convergence detection based on global residual error under asynchronous iterations. IEEE Trans Parallel Distrib Syst 2018;29(4):819–29.
[40] Bertsekas DP, Tsitsiklis JN. Parallel and distributed computation: numerical methods. Prentice-Hall; 1989.
[41] Thakur R, Rabenseifner R, Gropp W. Optimization of collective communication operations in MPICH. Int J High Perform Comput Appl 2005;19(1):49–66.
[42] Raydan M, Svaiter BF. Relaxed steepest descent and Cauchy-Barzilai-Borwein method. Comput Optim Appl 2002;21(2):155–67.
[43] Yuan Y-X. Step-sizes for the gradient method. Third international congress of chinese mathematicians. AMS/IP; 2008. p. 785–96.
[44] Heinig G, Rost K. Fast algorithms for Toeplitz and Hankel matrices. Linear Algebra Appl 2011;435(1):1–59.
[45] Magoulès F, Cheik Ahamed A-K. Alinea: an advanced linear algebra library for massively parallel computations on graphics processing units. Int J High Perform Comput Appl 2015;29(3):284–310.
[46] Davis TA, Hu Y. The University of Florida sparse matrix collection. ACM Trans Math Softw 2011;38(1):1–125.