Computers and Mathematics with Applications 57 (2009) 1369–1376

Contents lists available at ScienceDirect

Computers and Mathematics with Applications

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

A distributed memory parallel Gauss–Seidel algorithm for linear algebraic systems☆

Yueqiang Shang*

Faculty of Science, Xi'an Jiaotong University, Xi'an 710049, PR China
School of Mathematics and Computer Science, Guizhou Normal University, Guiyang 550001, PR China

ARTICLE INFO

Article history:
Received 26 June 2008
Received in revised form 17 December 2008
Accepted 12 January 2009

Keywords:
Parallel computing
Linear algebraic system
Gauss–Seidel method
Distributed memory system
Parallel algorithm

ABSTRACT

A distributed memory parallel Gauss–Seidel algorithm for linear algebraic systems is presented, in which a parameter is introduced to adapt the algorithm to different distributed memory parallel architectures. In this algorithm, the coefficient matrix and the right-hand side of the linear algebraic system are first divided into row-blocks in the natural rowwise-order according to the performance of the parallel architecture in use. And then these row-blocks are distributed among local memories of all processors through torus-wrap mapping techniques. The solution iteration vector is cyclically conveyed among processors at each iteration so as to decrease the communication. The algorithm is a true Gauss–Seidel algorithm which maintains the convergence rate of the serial Gauss–Seidel algorithm and allows existing sequential codes to run in a parallel environment with a little investment in recoding. Numerical results are also given which show that the algorithm is of relatively high efficiency.

1. Introduction

The solving of linear algebraic systems lies at the core of many scientific and engineering simulations. Many practical problems can be translated into a large scale linear algebraic system. Therefore the parallel solving of large scale linear algebraic systems is of great importance in the scientific computation field. Methods for a linear algebraic system generally fall into two categories: direct methods and iterative methods. The direct methods can arrive at the solution within a limited number of steps. However, direct methods may be impractical if the coefficient matrix of the linear algebraic system to be solved is large and sparse because the sought-after factor can be dense [1]. This is the reason why the iterative methods, which are able to take advantage of sparse systems, are often preferable compared with the direct methods in engineering fields. Through iterative methods, an approximate solution within the error tolerance could be obtained under the assumption that the algorithm is convergent. Among classical iterative methods, the Gauss–Seidel method has several interesting properties. In general, if the Jacobi method also converges, the Gauss–Seidel method will converge faster than the Jacobi method. Even though the SOR method with the optimal relaxation parameter is faster than the Gauss–Seidel method, however, choosing an optimal SOR relaxation parameter can be difficult for many problems of practical interest [2]. Therefore, the Gauss–Seidel method is very attractive in practice and it is usually used as the smoother of the multigrid method for partial differential equations which typically yields good multigrid convergence properties.

Parallel implementations of Gauss–Seidel method have generally been developed for regular problems, for example, the solution of Laplace's equations by finite differences [3,4], where a red-black coloring scheme is used to provide independence

☆ This work was supported by the Science and Technology Foundation of Guizhou Province, China [2008] 2123.
* Corresponding address: Faculty of Science, Xi'an Jiaotong University, P.O. Box 2325, Xi'an 710049, PR China.
E-mail address: yueqiangshang@gmail.com.

0898-1221/$ – see front matter © 2009 Elsevier Ltd. All rights reserved.
doi:10.1016/j.camwa.2009.01.034

in the calculations and some parallelism. This scheme has been extended to multi-coloring for additional parallelism in more complicated regular problems in [5]. In [6], Koester, Ranka and Fox have presented a parallel Gauss–Seidel algorithm for sparse power system matrices, in which a two-part matrix ordering technique has been developed—first to partition the matrix into block-diagonal bordered form using diakoptic techniques and then to multi-color the data in the last diagonal block using graph-coloring techniques. Unfortunately, the elegant simplicity of structured grid multi-color Gauss–Seidel is lost on 3D unstructured finite element applications as the number of required colors increases dramatically. Motivated by the above observation, Adams has proposed an efficient parallel Gauss–Seidel algorithm for unstructured problems suiting distributed memory computers in [7]. This algorithm takes advantage of the domain decomposition provided by the distribution of the stiffness matrix that is common in parallel computing. It first colors the processors and uses an ordering of the colors to provide a processor inequality operator. A weakness of this parallel-block multi-color Gauss–Seidel algorithm is that it requires different orderings depending on the number of processors. In [8,9], Amodio and Mazzia have developed a parallel Gauss–Seidel method to solve block-banded systems. They used a parallel structure which consists of one-dimensional logically connected processors. While in [10], Kim and Lee have used a parallel structure which consists of two-dimensional logically connected processors. On the other hand, to avoid parallelization difficulties, a processor-localized Gauss–Seidel is often employed instead of a true Gauss–Seidel method, in which each processor performs the Gauss–Seidel method as a subdomain solver for a block Jacobi method (see, e.g., [11]). While a processor-localized Gauss–Seidel is easy to parallelize, as shown in [11], its convergence rate usually suffers and can even lead to divergence when the number of processors increases.

In this paper, we present a distributed memory parallel Gauss–Seidel algorithm for general linear algebraic systems, in which a parameter is introduced to adapt the algorithm to different parallel architectures. The general idea of this algorithm is to divide the coefficient matrix and the right-hand side of the linear algebraic system into row-blocks in the natural rowwise-order, then to distribute these blocks among local memories of all processors through a torus-wrap mapping technique and cyclically convey the solution iteration vector at each iteration to decrease the communication. It is a true parallel Gauss–Seidel method and hence the convergence rate of the serial Gauss–Seidel algorithm is maintained. Furthermore, when the linear algebraic system arises from the discretization of a partial differential equation, this algorithm does not depend on the domain decomposition, the topological structure of grids, the sequencing skills of elements and other skills, hence it allows existing sequential codes to run in a parallel environment with a little investment in recoding.

The remainder of the paper is organized as follows. In Section 2, the Gauss–Seidel method and its convergence properties are recalled. In Section 3, the parallel implementation of Gauss–Seidel method is discussed and the corresponding parallel Gauss–Seidel algorithm is described and analyzed. A numerical experiment and analysis on numerical results are given in Section 4. Finally, some conclusions are drawn in Section 5.

2. The Gauss–Seidel method

Consider the following real linear algebraic system $Ax = b$:

$$\begin{bmatrix} a_{00} & a_{01} & \cdots & a_{0\,n-1} \\ a_{10} & a_{11} & \cdots & a_{1\,n-1} \\ \vdots & \vdots & & \vdots \\ a_{n-1\,0} & a_{n-1\,1} & \cdots & a_{n-1\,n-1} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{n-1} \end{bmatrix} = \begin{bmatrix} b_0 \\ b_1 \\ \vdots \\ b_{n-1} \end{bmatrix}, \tag{1}$$

where $A = (a_{ij})_{n\times n}$ is a known nonsingular $n \times n$ matrix with nonzero diagonal entries, $b = (b_0, b_1, \ldots, b_{n-1})^{\mathrm{T}}$ is the right-hand side and $x = (x_0, x_1, \ldots, x_{n-1})^{\mathrm{T}}$ is the vector of unknowns.

The Gauss–Seidel method for the above linear algebraic system reads: Given an initial guess $x^{(0)} = (x_0^{(0)}, x_1^{(0)}, \ldots, x_{n-1}^{(0)})^{\mathrm{T}}$, $x^{(k+1)} = (x_0^{(k+1)}, x_1^{(k+1)}, \ldots, x_{n-1}^{(k+1)})^{\mathrm{T}}$ is obtained by the following iterative procedure

$$\begin{cases} x_i^{(k+1)} = \dfrac{1}{a_{ii}} \left(b_i - \displaystyle\sum_{j=0}^{i-1} a_{ij} x_j^{(k+1)} - \sum_{j=i+1}^{n-1} a_{ij} x_j^{(k)} \right), \\ i = 0, 1, \ldots, n-1. \end{cases} \tag{2}$$

By introducing vectors $z = (z_0, z_1, \ldots, z_{n-1})^{\mathrm{T}}$ and $t = (t_0, t_1, \ldots, t_{n-1})^{\mathrm{T}}$, where

$$z_i = \begin{cases} 0 & i = 0 \\ -\displaystyle\sum_{j=0}^{i-1} a_{ij} x_j & i = 1, 2, \ldots, n-1, \end{cases} \tag{3}$$

$$t_i = b_i - \sum_{j=i+1}^{n-1} a_{ij} x_j \quad i = 0, 1, \ldots, n-1. \tag{4}$$

scheme (2) can be written as

$$x_i^{(k+1)} = \frac{1}{a_{ii}}(z_i^{(k+1)} + t_i^{(k)}). \tag{5}$$

As for the convergence of scheme (2) or (5), we have the following well-known result that if the coefficient matrix A is strictly diagonally dominant or irreducibly diagonally dominant, scheme (2) or (5) converges for any initial guess $x^{(0)}$ (see, e.g., [1]).

3. Parallel Gauss–Seidel algorithm

In this section, we detailedly discuss the parallel implementation of the Gauss–Seidel method described in Section 2 on distributed-memory parallel architectures.

3.1. Strategy for data distribution and storage

Firstly, we divide the linear algebraic system (1) into $m = n/g$ row-blocks in the natural rowwise order (for simplicity of presentation, we assume n can be divided exactly by g) as follows.

$$\begin{bmatrix} A_0 \\ A_1 \\ \vdots \\ A_{m-1} \end{bmatrix} \begin{bmatrix} X_0 \\ X_1 \\ \vdots \\ X_{m-1} \end{bmatrix} = \begin{bmatrix} B_0 \\ B_1 \\ \vdots \\ B_{m-1} \end{bmatrix}, \tag{6}$$

where A_i $(i = 0, 1, \ldots, m-1)$ are $g \times n$ matrices each consisting of the successive rows of the coefficient matrix A, X_i and B_i $(i = 0, 1, \ldots, m-1)$ are g-dimensional vectors consisting of the successive components of the unknown vector x and the right-hand side vector b, respectively. Then, we distribute and store these row-blocks of (6) among local memories of all processors in the parallel system through a torus-wrap mapping technique. Specifically, assuming p is the number of processors in the parallel system, the 0th processor stores the 0th, pth, $2p$th,... row-blocks of (6), the 1st processor stores the 1st, $(p+1)$th, $(2p+1)$th, ... row-blocks of (6), the 2nd processor stores the 2nd, $(p+2)$th, $(2p+2)$th, ... row-blocks of (6), etc. If $p = 3$, $m = 8$, for example, then the 0th processor stores the A_0, A_3, A_6 and B_0, B_3, B_6 of (6), the 1st processor stores the A_1, A_4, A_7 and B_1, B_4, B_7 of (6), and the 2nd processor stores the A_2, A_5 and B_2, B_5 of (6), respectively.

3.2. Parallel implementation

From scheme (2) or (5), we can see that the Gauss–Seidel iterative process has a very strong sequential property in the sense that x_{j+1} cannot be computed until x_j is known. To obtain good load-balancing, we adopt the above strategy to distribute and store the coefficient matrix and the right-hand side of the linear algebraic system (1). The Master/Slave model is employed in which the master processor (it also performs the task of a slave processor) partitions the computation task, initializes the iteration vector, collects and prints the computed results, while the slave processors perform the actual computation involved. Meanwhile, we cyclically convey the solution iteration vector at each iteration to decrease the communication and adopt the technique of overlapping communication and computations to increase the parallel efficiency. Concretely speaking, the $(k+1)$th iteration is performed as follows, where ϵ is the error tolerance.

(1) All the slave processors summate the data at the right-hand side of the dominant diagonal of A in parallel, i.e., compute $t_i^{(k)}(i = 0, 1, \ldots, n-1)$ in parallel, and set $sign = 0$.

(2) The 0th slave processor first computes $x_0^{(k+1)}, x_1^{(k+1)}, \ldots, x_{g-1}^{(k+1)}$, and checks whether the conditions $|x_i^{(k+1)} - x_i^{(k)}| < \epsilon$ $(i = 0, 1, \ldots, g-1)$ hold. If there exists one of the above inequalities not being true, set $sign = 1$. Then the 0th slave processor sends $x_0^{(k+1)}, x_1^{(k+1)}, \ldots, x_{g-1}^{(k+1)}$ and $sign$ to the 1st slave processor. After that, the 0th slave processor and the 1st processor perform the following tasks in parallel: the 0th slave processor uses the newly computed $x_0^{(k+1)}, x_1^{(k+1)}, \ldots, x_{g-1}^{(k+1)}$ to compute other relevant items of the remainder local rows, i.e., to compute the partial sum in $z_i^{(k+1)}$ involving $x_0^{(k+1)}, x_1^{(k+1)}, \ldots, x_{g-1}^{(k+1)}$, while the 1st processor receives $x_0^{(k+1)}, x_1^{(k+1)}, \ldots, x_{g-1}^{(k+1)}$ and $sign$ from the 0th slave processor and computes $x_g^{(k+1)}, x_{g+1}^{(k+1)}, \ldots, x_{2g-1}^{(k+1)}$, checks whether the conditions $|x_i^{(k+1)} - x_i^{(k)}| < \epsilon$ $(i = g, g+1, \ldots, 2g-1)$ hold. Invalidity of any the above inequalities leads to $sign = 1$.

(3) After computing $x_g^{(k+1)}, x_{g+1}^{(k+1)}, \ldots, x_{2g-1}^{(k+1)}$ and $sign$, the 1st slave processor sends $x_0^{(k+1)}, x_1^{(k+1)}, \ldots, x_{2g-1}^{(k+1)}$ and $sign$ to the 2nd slave processor. Then the 1st slave processor and the 2nd slave processor carry out the following tasks in parallel: the 1st slave processor uses $x_0^{(k+1)}, x_1^{(k+1)}, \ldots, x_{g-1}^{(k+1)}$ and the newly computed $x_g^{(k+1)}, x_{g+1}^{(k+1)}, \ldots, x_{2g-1}^{(k+1)}$ to compute other relevant items of the remainder local rows, while the 2nd slave processor receives $x_0^{(k+1)}, x_1^{(k+1)}, \ldots, x_{2g-1}^{(k+1)}$ and $sign$ from the 1st slave processor and computes $x_{2g}^{(k+1)}, x_{2g+1}^{(k+1)}, \ldots, x_{3g-1}^{(k+1)}$, checks whether the conditions $|x_i^{(k+1)} - x_i^{(k)}| < \epsilon$ $(i = 2g, 2g+1, \ldots, 3g-1)$ hold. If there is one of the above inequations not satisfied, set $sign = 1$.

(4) Go on as the above process until the vth ($0 \le v \le p-1$) slave processor, which stores the last row-block of (6), receives $x_0^{(k+1)}, x_1^{(k+1)}, \ldots, x_{n-g-1}^{(k+1)}$ and *sign*, computes $x_{n-g}^{(k+1)}, x_{n-g+1}^{(k+1)}, \ldots, x_{n-1}^{(k+1)}$, and then checks whether $|x_i^{(k+1)} - x_i^{(k)}| < \epsilon$ ($i = n-g, n-g+1, \ldots, n-1$) hold, if there is one of the above inequations not valid, set $sign = 1$.

(5) The vth slave processor checks whether $sign = 0$ is satisfied, if it is true, sends $x_0^{(k+1)}, x_1^{(k+1)}, \ldots, x_{n-1}^{(k+1)}$ to master processor and stops the iteration, or else, broadcasts $x_0^{(k+1)}, x_1^{(k+1)}, \ldots, x_{n-g-1}^{(k+1)}$ to other slave processors and starts the next iteration.

3.3. Parallel algorithm

Based on the above parallel implementation of Gauss–Seidel method, we describe our parallel Gauss–Seidel algorithm as follows, in which *total* is the number of the row-blocks of (6) stored on the processor.

Algorithm. Parallel Gauss–Seidel algorithm.

Sub-algorithm for master processor.

(1) Initialize the iteration vector and the parameter g according to the size of the problem and the performance of the parallel architecture in use.
(2) Broadcast the parameter g, initial iteration vector and other information about partitioning the computation task to slave processors.
(3) Receive the approximate solution vector and print it.

Sub-algorithm for the jth ($j > 0$) *slave processor.*

(1) Receive the parameter g, initial iteration vector and other information about partitioning the computation task from master processor, read my local data.
(2) For $k = 1, 2, \ldots$, until convergence:
  (i) Set $sign = 0$, compute my local $t_i^{(k)}$ in parallel with other slave processors.
  (ii) For $r = 0, 1, \ldots, total - 1$, do:
    (a) Receive $x_0^{(k+1)}, x_1^{(k+1)}, \ldots, x_{r*p+j*g-1}^{(k+1)}$ and *sign* from the $(j-1)$th processor, compute $x_{r*p+j*g}^{(k+1)}, x_{r*p+j*g+1}^{(k+1)}, \ldots, x_{r*p+(j+1)*g-1}^{(k+1)}$ and check whether they satisfy the accurate requirement. If the accurate requirement is not satisfied, set $sign = 1$.
    (b) Send $x_0^{(k+1)}, x_1^{(k+1)}, \ldots, x_{r*p+(j+1)*g-1}^{(k+1)}$ and *sign* to the $(j+1)$th processor, then compute other relevant items of the remainder local rows.
  (iii) Broadcast or receive $x_0^{(k+1)}, x_1^{(k+1)}, \ldots, x_{n-1}^{(k+1)}$.

3.4. Speedup and parallel efficiency

The performance of parallel algorithms in a homogeneous distributed memory parallel environment is measured by *speedup* and *parallel efficiency*, defined as

$$S_{par}(p) = \frac{T_s}{T(p)}, \qquad E_{par}(p) = \frac{S_{par}(p)}{p}, \tag{7}$$

where T_s is the wall (elapsed) time for the best sequential algorithm, $T(p)$ stands for the wall (elapsed) time for the parallel algorithm using p processors, while $S_{par}(p)$ and $E_{par}(p)$ denote the parallel speedup and parallel efficiency for parallel computation, respectively. Because the best sequential algorithm is not known in general, in practical applications, T_s is usually replaced by $T(1)$, the wall time of the same parallel program on one processor (i.e., both master and slave tasks are performed on one processor). Thus, the speedup $S_{par}(p)$ and parallel efficiency $E_{par}(p)$ are commonly computed as follows:

$$S_{par}(p) = \frac{T(1)}{T(p)}, \qquad E_{par}(p) = \frac{S_{par}(p)}{p}. \tag{8}$$

The parallel efficiency takes into account of the loss of efficiency due to the data communication and other extra costs, such as on a partitioning computation task, on scheduling processes and on data management, introduced by the parallelization. The wall time for parallel computation with p processors can be decomposed into the following form

$$T(p) = T_{par}(p) + T_{seq}(p) + T_{comu}(p) + T_{extra}(p), \tag{9}$$

where $T_{par}(p)$ is the average CPU time spent on computation on slave processors satisfying $T_{par}(p) = \frac{T_{par}(1)}{p}$ for our algorithm, $T_{seq}(p)$ is the CPU time taken on the serial components of the computation satisfying $T_{seq}(p) = T_{seq}(1)$, $T_{comu}(p)$ is the

communication time with p processors and $T_{extra}(p)$ denotes other extra time (for example, idle time) introduced by the parallelization. Therefore, the speedup $S_{par}(p)$ can be written as

$$\begin{aligned} S_{par}(p) = \frac{T(1)}{T(p)} &= \frac{T_{par}(1) + T_{seq}(1) + T_{comu}(1) + T_{extra}(1)}{T_{par}(1)/p + T_{seq}(1) + T_{comu}(p) + T_{extra}(p)} \\ &< \frac{T_{par}(1) + T_{seq}(1) + T_{comu}(1) + T_{extra}(1)}{T_{seq}(1) + T_{comu}(p) + T_{extra}(p)}. \end{aligned} \tag{10}$$

This is called Amdahl's law.

At each iteration of our parallel algorithm, There are m times communication: $m-1$ for local communication and one for global communication. The amount of transferred data at the kth ($k = 0, 1, \ldots, m-2$) local communication is $(k+1)g$ units (float or double) data plus an integer *sign*, while that of the global communication is n units (float or double) data. Therefore, assuming the network bandwidth is big enough to transfer n units data at a time, the total communication time with p ($p > 1$) processors for our parallel algorithm is approximatively

$$\begin{aligned} T_{comu}(p) &= \log_2(p)t_w + (n+1)\log_2(p)t_c + \left\{(m-1)t_w + \left[\frac{gm(m-1)}{2} + m - 1\right]t_c\right\} \times IC \\ &\quad + \{\log_2(p)t_w + n\log_2(p)t_c\} \times (IC-1) + t_w + nt_c \\ &= \{[\log_2(p) + m - 1] \times IC + 1\}t_w + \left\{\left[n\log_2(p) + \frac{(m-1)(gm+2)}{2}\right] \times IC + \log_2(p) + n\right\}t_c, \end{aligned} \tag{11}$$

where IC is the iteration count satisfying the stopping criterion, t_c is the transfer time of a unit (float or double) datum and t_w is the communication startup time. Assume that t_a is the time of a (float or double) number operation of type multiplication, division, addition or subtraction and that the time of reading, valuing and printing a unit datum is also equal to t_a, then for our algorithm

$$\begin{aligned} &T_{par}(1) = (n^2+n)t_a + 2n^2t_a \times IC, \qquad T_{seq}(1) = 2nt_a, \\ &T_{extra}(p) = \left\{\frac{g^2(p-1)^2t_a}{2} + O(g(p-1)t_a)\right\} \times IC. \end{aligned} \tag{12}$$

Therefore, noting $n = mg$ and neglecting the communication time $T_{comu}(1)$ when only one processor is employed, the speedup and parallel efficiency of our parallel algorithm can be written as

$$\begin{aligned} S_{par}(p) &= \left\{(n^2+n)t_a + 2n^2t_a \times IC + 2nt_a\right\}\left\{\frac{(n^2+n)t_a + 2n^2t_a \times IC}{p} + 2nt_a\right. \\ &\quad + [(\log_2(p) + m - 1) \times IC + 1]t_w + \left[\left(n\log_2(p) + \frac{(m-1)(gm+2)}{2}\right) \times IC + \log_2(p) + n\right]t_c \\ &\quad \left. + \left[\frac{g^2(p-1)^2}{2} + O(g(p-1))\right]t_a \times IC\right\}^{-1} \\ &= \left\{2n^2 \times IC + n^2 + 3n\right\}\left\{\frac{2n^2 \times IC + n^2 + n}{p} + 2n + \left[\left(\log_2(p) + \frac{n}{g} - 1\right) \times IC + 1\right]\frac{t_w}{t_a}\right. \\ &\quad \left. + \left[\left(n\log_2(p) + \frac{(\frac{n}{g} - 1)(n+2)}{2}\right) \times IC + \log_2(p) + n\right]\frac{t_c}{t_a} + \left[\frac{g^2(p-1)^2}{2} + O(g(p-1))\right] \times IC\right\}^{-1} \end{aligned} \tag{13}$$

$$\begin{aligned} E_{par}(p) &= \left\{2n^2 \times IC + n^2 + 3n\right\}\left\{2n^2 \times IC + n^2 + n + 2np\right. \\ &\quad + \left[\left(\log_2(p) + \frac{n}{g} - 1\right) \times IC + 1\right]p\frac{t_w}{t_a} + \left[\left(n\log_2(p) + \frac{\left(\frac{n}{g} - 1\right)(n+2)}{2}\right) \times IC + \log_2(p) + n\right] \\ &\quad \left. \times p\frac{t_c}{t_a} + \left[\frac{g^2(p-1)^2}{2} + O(g(p-1))\right]p \times IC\right\}^{-1}. \end{aligned} \tag{14}$$

Remark. The above analysis is for the case where the coefficient matrix A is dense. When A is sparse, the corresponding speedup and parallel efficiency can be easily obtained by a similar procedure if we know the number of nonzero entries of A.

From (13) and (14), we can see that for a size-fixed problem and a given parallel system, the speedup of the algorithm varies with the row-block-partitioning parameter g. In general, the bigger the parameter g, the fewer the times of the communication (which equals $\frac{n}{g}$ at each iteration) and hence the less the communication startup time leading to a decreased total communication time. Meanwhile, when the parameter g grows, the load-balancing becomes worse leading to more idle time among processors and then leading to a degraded parallel efficiency. Therefore, to obtain good parallel performance, the parameter g should be suitably chosen.

3.5. The choice of parameter g

In parallel computing, due to the different structures and performance of parallel architectures, an algorithm achieving good performance on a parallel architecture does not necessarily suit and perform well on another one. Therefore, portability and flexibility are very important for a parallel algorithm. Among the attractive features of our parallel algorithm is the great flexibility in the choice of the parameter g to adapt the algorithm to different parallel architectures. From the above analysis, we can see that for a size-fixed problem, the speedup and then the parallel efficiency are the functions of the row-block-partitioning parameter g which can be suitably valued according to the performance of the parallel system in use. For an application of our parallel Gauss–Seidel algorithm, one may obtain good parallel performance by suitably choosing the value of the row-block-partitioning parameter g according to the size of the problem and the performance (mainly, the values of $\frac{t_c}{t_a}$ and $\frac{t_w}{t_a}$) of the parallel systems in use. Specifically speaking, for a given parallel system, we can first use some standard routines (for example, in PVM and MPI softwares) to measure its communication startup time t_w, transfer time of unit datum t_c and the unit arithmetic operation time t_a. Then from (13) and through a calculus approach, we may obtain good speedups by choosing the parameter g as

$$g \approx \sqrt[3]{\frac{\frac{n(n+2)}{2} \times \frac{t_c}{t_a} + n \times \frac{t_w}{t_a}}{(p-1)^2}}. \tag{15}$$

4. Numerical experiment

4.1. Numerical results

In our experiment, the parallel platform is made up of 4 PCs, each with a Pentium IV 2.4 GHz CPU, 80G hard disk and 512M DRAM (the master processor has 2G DRAM), installed Microsoft Windows2000 operation system, Microsoft VC 6.0 and connected together by 100 Mbps Ethernet. Message-passing is supported through PVM 3.4, which is a public-domain software from Oak Ridge National Laboratory [12]. PVM is a software system that enables a collection of homogeneous or heterogeneous computers to be used as a coherent and flexible concurrent computational resource, or a "parallel virtual machine". PVM consists of two parts: a daemon process that any user can install on a machine, and a user library that contains routines for initiating processes on other machines, for communicating between processes, and for changing the configuration of machines. User's programs written in C, C++ or Fortran can access PVM through provided library routines for functions such as process initiation, message transmission, synchronization and coordination via barriers or rendezvous. Our codes for experiment were written in C++ making use of a PVM message-passing library.

For simplicity, the linear algebraic system $Ax = b$ in our experiment is chosen as the following special form.

$$A = (a_{i,j})_{n\times n} = \begin{cases} n & i = j, \\ 0.8^{|i-j|} & 0 < |i-j| \le n/4, \\ 0 & \text{other}, \end{cases} \tag{16}$$

$i, j = 0, 1, \ldots, n-1$. The vector b of right-hand side is randomly valued. Processors compute with double precision. The initial iteration vector is chosen as zero vector and the stopping criterion is

$$|x_i^{(k+1)} - x_i^{(k)}| < 10^{-6} \quad (i = 0, 1, \ldots, n-1).$$

The numerical results are listed in Tables 1 and 2.

4.2. Analysis based on numerical results

From Table 2, we can see that our parallel Gauss–Seidel algorithm is of relatively high efficiency. Good speedups were obtained with a certain row-block-partitioning parameter g in the experiment.

As predicted by the previous analysis, Tables 1 and 2 show that for a size-fixed problem and a given parallel system, the wall (elapsed) time and the speedup of the algorithm are closely correlated with the row-block-partitioning parameter g. When the row-block-partitioning parameter g is smaller than a certain value, the increasing speed of the communication time is higher than the decreasing speed of the computation time as the number of the processors increases and hence

Table 1
The wall (elapsed) time in seconds of the parallel Gauss–Seidel algorithm with double precision, where p denotes the number of processors, n the order of the coefficient matrix and g the row-block-partitioning parameter.

p	$n = 16\,000$				$n = 24\,000$			
	$g = 100$	$g = 500$	$g = 1000$	$g = 2000$	$g = 100$	$g = 500$	$g = 1000$	$g = 2000$
1	59.093	50.609	50.109	49.109	132.938	114.110	112.500	112.438
2	59.547	30.656	28.953	29.204	124.250	68.797	63.422	62.344
3	66.922	34.094	27.281	31.360	130.516	66.407	55.125	53.969
4	79.938	36.953	30.391	33.641	150.797	68.235	56.094	55.954

Table 2
The speedup of the parallel Gauss–Seidel algorithm.

p	$n = 16\,000$				$n = 24\,000$			
	$g = 100$	$g = 500$	$g = 1000$	$g = 2000$	$g = 100$	$g = 500$	$g = 1000$	$g = 2000$
2	0.992	1.651	1.731	1.711	1.070	1.659	1.774	1.804
3	0.883	1.484	1.837	1.578	1.019	1.718	2.041	2.083
4	0.739	1.370	1.649	1.485	0.881	1.672	2.006	2.009

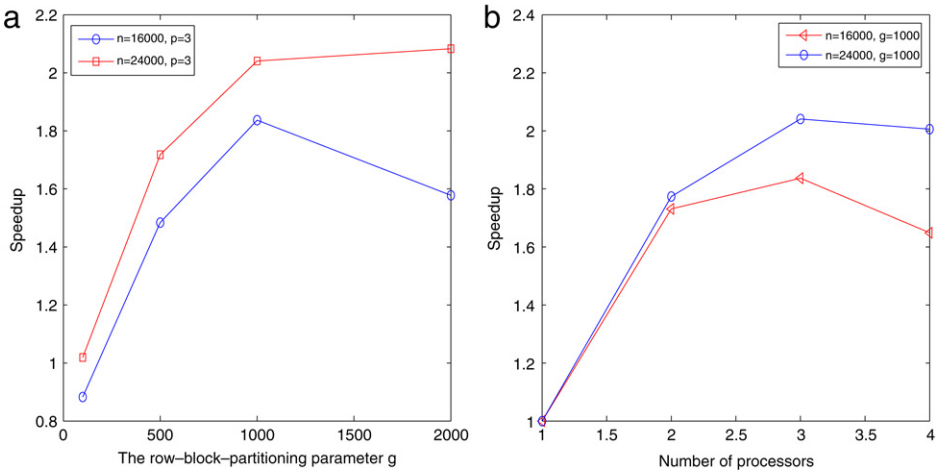

Fig. 1. Speedup versus (a) the row-block-partitioning parameter g and (b) number of processors.

results in a increased rather than decreased wall time for the algorithm. Consequently, the speedups are smaller than one and decrease as the number of the processors increases; see the second columns of Tables 1 and 2 for the case $n = 16\,000$, $g = 100$. Only within a certain range of the row-block-partitioning parameter g, does the speedup increase with growing g. However, when the row-block-partitioning parameter g exceeds another certain value, the speedup decreases instead of increasing as g further increases; see Fig. 1(a) for the case $p = 3$. A possible reason may be that with the growing row-block-partitioning parameter g, the time of communication decreases; at the same time, however, the load imbalance among processors becomes worse, i.e., there is more waiting time among processors. Therefore, when the increase in the waiting time is larger than the decrease in the communication time, decreased speedups and then decreased parallel efficiency are observed. As is shown in Table 2 and Fig. 1(a), the "optimal" value of g for the cases $n = 16\,000$ and $n = 24\,000$ are 1000 and 2000, respectively. Here and hereafter, "optimal" value means that after which, the speedup decreases rather than increases as the corresponding parameter increases in our experiment.

It is also found that for a certain value of the parameter g, when the size of the problem is fixed and the number of processors is small, the speedup increases with the number of processors; nevertheless, when the number of processors exceeds a certain value, the speedup decreases instead of increasing as the number of processors increases; see Fig. 1(b) for the case $g = 1000$. The reason may be that, on one hand, as the number of processors grows, the communication and the idle time among processors increases sharply; on the other hand, with more processors, the granularity of computation task for each processor becomes smaller making the relative ratio of the time spent on communication and waiting to that spent on the computation higher. When this decrease in computation time less than the increase in communication and waiting time introduced by these additional processors, a declining speedup, and then decreasing parallel efficiency are observed.

On the other hand, when the parameter g and the number of processors are fixed, the speedup increases as the size of the problem increases; see Table 2. A possible explanation is that the relative ratio of time spent on computation to that spent on communication and waiting becomes higher as the size of the problem increases when the parameter g and the number of processors are fixed and consequently, the speedups increase.

5. Conclusion

In this paper, we have discussed a distributed memory parallel Gauss–Seidel algorithm for general linear algebraic systems. A numerical experiment performed on a local area of 4 PCs with PVM for the message-passing is also given. The conclusions are as follows.

- By introducing a row-block-partitioning parameter g, the algorithm is flexible adapting to different distributed memory parallel systems. With a suitable value of the parameter g depending on the performance (mainly, the relative ratio of communication to computation speeds) of the parallel system in use and the size of the problem, good parallel performance may be obtained.
- The technique of cyclically conveying the solution iteration vector leads to a large decrease in the communication time and hence results in a high speedup and parallel efficiency of the algorithm with a certain value of the parameter g.
- For a parameters-given problem, there is an "optimal" number of processors which maximizes the parallel efficiency. The larger the size of the problem, the larger the "optimal" number of processors.

Acknowledgements

The author would like to thank the editor and referees for their valuable comments and suggestions which helped to improve the results of this paper. The free software PVM was also extensively used in this study.

References

[1] G.H. Golub, C.F. Van Loan, Matrix Computations, The Johns Hopkins University Press, Baltimore, 1983.
[2] D.M. Young, Iterative Solution of Large Linear Systems, Academic Press, New York, 1971.
[3] L.M. Adams, H.F. Jordan, Is SOR color-blind? SIAM J. Sci. Statist. Comput. 7 (2) (1986) 490–506.
[4] G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, D. Walker, Solving Problems on Concurrent Processors, Prentice Hall, 1988.
[5] G. Golub, J.M. Ortega, Scientific Computing with an Introduction to Parallel Computing, Academic Press, Boston, MA, 1993.
[6] D.P. Koester, S. Ranka, G.C. Fox, A parallel Gauss–Seidel algorithm for sparse power system matrices, in: Supercomputing '94 Proceedings, 14–18 Nov, 1994, 184–193.
[7] M.F. Adams, A distributed memory unstructured Gauss–Seidel algorithm for multigrid smoothers, in: ACM/IEEE Proceedings of SC01: High Performance Networking and Computing, 2001.
[8] P. Amodio, F. Mazzia, A parallel Gauss–Seidel method for block tridiagonal linear systems, SIAM J. Sci. Comput. 6 (1995) 1451–1461.
[9] P. Amodio, F. Mazzia, Parallel iterative solvers for boundary value methods, Math. Comput. Modelling 23 (1996) 29–43.
[10] T. Kim, C.O. Lee, A parallel Gauss–Seidel method using NR data flow ordering, Appl. Math. Comput. 99 (1999) 209–220.
[11] M. Adams, M. Brezina, J. Hu, R. Tuminaro, Parallel multigrid smoothing: Polynomial versus Gauss–Seidel, J. Comput. Phys. 188 (2003) 593–610.
[12] A. Geist, A. Beguelin, J. Dongarra, et al., PVM: Parallel Virtual Machine—A Users' Guide and Tutorial for Networked Parallel Computing, MIT Press, Cambridge, 1994.