Parallel Preconditioners with Low Rank Off-Diagonal Blocks [1]

Randall Bramley and Vladimir Meñkov [a]

[a] *Department of Computer Science*
Indiana University–Bloomington
{bramley,vmenkov}@cs.indiana.edu

A common approach for blocked dense linear systems occurring in integral equations is to replace off–diagonal blocks with low rank approximations. In [3] this idea was applied to sparse linear systems occuring in PDE discretizations. Diagonal blocks are replaced with approximate factorizations and off–diagonal blocks with low rank approximations, and the resulting matrix is used as a preconditioner for an iterative method. This paper shows that high parallel efficiencies can be obtained when applying the Sherman–Morrison–Woodbury formula with these methods. Order estimates are developed showing that preconditioners based on low rank approximations can be implemented with roughly twice the cost per iteration of a block-diagonal preconditioner. The results apply as well to BSSOR–like preconditioning where the off–diagonal blocks of the coefficient matrix are replaced by low rank approximations. The order estimates also provide an operational definition of "low rank". Testing results are presented for the proposed approaches showing excellent performance on a shared memory multiprocessor.

Key words: Low-rank approximation,
Sherman–Morrison–Woodbury formula, parallel preconditioning.

1 Introduction

Solving a large sparse linear system of equations $Au = f$ is a common subproblem encountered in the numerical solution of differential and integral

[1] Work supported by NSF grants CDA-9303189, ASC-9502292 and Rome Labs AF 30602-92-C-0135.

equations. Such linear systems are often solved with preconditioned conjugate gradient–like methods, which require solving on each iteration the auxillary system $Cx = y$ with a preconditioner C. In parallel computing block diagonal preconditioning is commonly used. If diagonal blocks are assigned to processors, this preconditioner can be applied without interprocessor synchronization or communication. Unfortunately, block diagonal preconditioned systems often fail to converge, especially for difficult nonsymmetric matrices A. Block symmetric Gauss-Seidel is significantly more robust, but the block lower and upper triangular system solves are inherently sequential. One way to recover parallelism is to increase the number of blocks in the partitioning of A and use level scheduling [1,2]. However, this also causes the quality of preconditioning to fall, and in the limiting case becomes pointwise symmetrized Gauss-Seidel, which typically provides poor quality of preconditioning. A second way is to find a representation of the inverse of C, and apply this approximate inverse preconditioner with easily parallelized matrix–vector products [12]. However, for a given amount of storage the quality of approximate inverse preconditioners is typically lower than methods that solve $Cx = y$.

This paper borrows an idea from the solution of the *dense* systems that occur in integral equations, and uses low rank approximations (LRA) of the off–diagonal blocks of the *sparse* matrix A. Diagonal blocks of A are implicitly approximated using standard preconditioners such as ILU(s) [10]. This leads to the direct low rank off–diagonal block (DLRA) preconditioner

$$C_{\mathrm{DIR}} = D + Q_C \tag{1}$$

and the block SSOR with low rank off–diagonal blocks (SLRA) preconditioner

$$C_{\mathrm{SOR}} = (D + Q_L)D^{-1}(D + Q_R), \tag{2}$$

where D is a nonsingular block diagonal matrix, Q_L and Q_R are block strictly lower and upper triangular matrices composed of low rank blocks, and Q_C has zero diagonal blocks and low rank off–diagonal blocks. Earlier work [3] analyzed the effects of replacing off–diagonal blocks by lower rank approximations for PDE's on a regular mesh. This work examines the parallel implementation of those preconditioners, and shows under what circumstances they can achieve high parallel efficiency. In particular they can be implemented with parallelism almost matching that of block diagonal preconditioning. By varying the rank of the off–diagonal blocks the new methods can be parametrized to vary in quality between that of block diagonal and block SSOR [13] preconditioning, providing robustness for difficult systems.

2 Notation and Data Distribution

Both preconditioners require solving one or two block linear systems of the form $Bx = y$, where the nonsingular $n \times n$ matrix B has the splitting $B = D+Q$ with a nonsingular block-diagonal matrix $D = \mathrm{diag}\{D_{11}, D_{22}, \ldots, D_{pp}\}$ and some block matrix Q. The n variables are divided into p blocks so that B and other matrices of the same dimension have block dimensions $p \times p$. A typical source of the blocking is from domain decomposition, with $p \ll n$.

Let $r_{kl} = \mathrm{rank}\, Q_{kl}$, let $M = \sum_{k,l} r_{kl}$ be the sum of ranks of all blocks of Q, let $m_l = \sum_k r_{kl}$ be the sum of ranks of the blocks of Q in the l-th block column, and let m be the maximum of the sum of the ranks of the blocks in any one block column or block row of Q. If no block of Q has rank higher than one, then M becomes the number of nonzero blocks in Q, and m the maximum number of nonzero blocks in any one row or column of Q. Let $\mathcal{P}$ be the set of all the pairs (k, l) such that $Q_{kl} \neq 0$.

Each block of Q can be represented as

$$Q_{kl} = \sum_{i=1}^{r_{kl}} u_{kl}^i {v_{kl}^i}^{\mathrm{T}} = U_{kl} V_{kl}^T, \tag{3}$$

with $U_{kl} = [\, u_{kl}^1, \quad u_{kl}^2, \quad \cdots, \quad u_{kl}^{r_{kl}} \,]$ and $V_{kl} = [\, v_{kl}^1, \quad v_{kl}^2, \quad \cdots, \quad v_{kl}^{r_{kl}} \,]$.

Since it is the preconditioning step for an iterative solver, the system $Bx = y$ needs to be solved for many right-hand sides y, which are not all available at the same time.

Each block of D is represented as a product of a lower and an upper triangular matrices obtained by incomplete LU factorization of the diagonal blocks of A, but the analysis here only requires that systems with coefficient matrices D_{kk} be easily solved. The data are distributed among the processors with processor k storing D_{kk}, U_{kl}, and V_{kl}. Here processor k may be a virtual processor, and if fewer than p processors are available, several virtual processors are mapped to one physical processor.

3 Representation of B^{-1}

Equation (3) implies that Q can be written as UV^{T}, where U and V have M columns each:

$$U = \begin{bmatrix} U_{11} & 0 & \cdots & 0 & U_{12} & 0 & \cdots & 0 & \cdots & U_{1p} & 0 & \cdots & 0 \\ 0 & U_{21} & \cdots & 0 & 0 & U_{22} & \cdots & 0 & \cdots & 0 & U_{2p} & \cdots & 0 \\ \vdots & & \ddots & \vdots & \vdots & & \ddots & \vdots & \cdots & \vdots & & \ddots & \vdots \\ 0 & & 0 & U_{p1} & 0 & & 0 & U_{p2} & \cdots & 0 & & 0 & U_{pp} \end{bmatrix} \tag{4}$$

$$V = \begin{bmatrix} V_{11} & V_{21} & \cdots & V_{p1} & 0 & 0 & \cdots & 0 & \cdots & 0 & 0 & \cdots & 0 \\ 0 & 0 & \cdots & 0 & V_{12} & V_{22} & \cdots & V_{p2} & \cdots & \vdots & 0 & & \vdots \\ \vdots & & & & \vdots & & & & \cdots & 0 & & & 0 \\ 0 & 0 & \cdots & 0 & 0 & 0 & \cdots & 0 & \cdots & V_{1p} & V_{2p} & \cdots & V_{pp} \end{bmatrix} \tag{5}$$

This block column ordering is chosen so that G will have the same block structure as Q (see Section 4). Block columns in (4) and (5) corresponding to zero blocks Q_{kl} are absent.

Since $B = D + UV^{\mathrm{T}}$, the Sherman–Morrison–Woodbury (SMW) formula [11] gives

$$B^{-1} = (D + UV^{\mathrm{T}})^{-1} = D^{-1} - D^{-1}U(I + G)^{-1}V^{\mathrm{T}}D^{-1}, \tag{6}$$

where $G = V^{\mathrm{T}}D^{-1}U$ is of order M. This SMW formula is the basis of the implementations we use. Since the nonzero eigenvalues of G are the same as those of $D^{-1}UV^{\mathrm{T}}$,

$$\sigma(I + G) \setminus \{1\} = \sigma(D^{-1}B) \setminus \{1\},$$

where $\sigma(H)$ is the set of eigenvalues of the square matrix H. Since B is nonsingular, so is $I + G$. Furthermore, if $M < n$ (i.e. if the ranks of the blocks of Q are low enough), then $\operatorname{rank}(UV^{\mathrm{T}}) \leq M < n$, and $1 \in \sigma(D^{-1}B)$. Therefore $\sigma(I + G) \subset \sigma(D^{-1}B)$, and $\operatorname{cond}(I + G) \leq \operatorname{cond}(D^{-1}B)$. So no additional numerical stability difficulties are introduced from using SMW.

4 Structure of G

Introducing the matrix G from the SMW formula could potentially cause storage problems, if its block structure is significantly denser than that of Q. The following proposition shows that the block sparsity pattern of G is contained in that of Q:

Proposition 1 *If $Q_{kl} = 0$, then $G_{kl} = 0$.*

PROOF. G can be partitioned as

$$G = \begin{bmatrix} G_{11} & G_{12} & \cdots & G_{1p} \\ G_{21} & G_{22} & \cdots & G_{2p} \\ \vdots & & & \vdots \\ G_{p1} & G_{p2} & \cdots & G_{pp} \end{bmatrix},$$

where an $m_k \times m_l$ block G_{kl} has the structure

$$\begin{aligned} G_{kl} &= \begin{bmatrix} 0 & \cdots & 0 & V_{1k}^{\mathrm{T}} & 0 & \cdots & 0 \\ 0 & \cdots & 0 & V_{2k}^{\mathrm{T}} & 0 & \cdots & 0 \\ \vdots & & \vdots & \vdots & \vdots & & \vdots \\ 0 & \cdots & 0 & V_{pk}^{\mathrm{T}} & 0 & \cdots & 0 \end{bmatrix} \cdot \begin{bmatrix} D_{11}^{-1}U_{11} & 0 & \cdots & 0 \\ 0 & D_{22}^{-1}U_{21} & & \vdots \\ \vdots & & \ddots & 0 \\ 0 & \cdots & 0 & D_{pp}^{-1}U_{p1} \end{bmatrix} \\ &= \begin{bmatrix} 0 & \cdots & 0 & H_{1kl} & 0 & \cdots & 0 \\ 0 & \cdots & 0 & H_{2kl} & 0 & \cdots & 0 \\ \vdots & & \vdots & \vdots & \vdots & & \vdots \\ 0 & \cdots & 0 & H_{pkl} & 0 & \cdots & 0 \end{bmatrix}, \end{aligned} \tag{7}$$

with zeros in the first $k-1$ block columns (of the total width $\sum_{j=1}^{k-1} r_{jl}$ columns) and the last $p-k$ block columns (of the total width $\sum_{j=k+1}^{p} r_{jl}$ columns). The block $H_{jkl} = V_{jk}^{\mathrm{T}} D_{kk}^{-1} U_{kl}$ is of the size $r_{jk} \times r_{kl}$. The equation for H_{jkl} shows that the block G_{kl} can be nonzero only if U_{kl} is nonzero, which, in its turn, can be nonzero only if Q_{kl} is nonzero. $\square$

If *no exact cancellation* [6, Section 2.2.2] is assumed Proposition 1 shows that G has the same nonzero block structure as Q.

Corollary 2 *If Q is strictly block upper or lower triangular, then so is G.*

Note that, as (7) shows, all nonzeros in G are concentrated in p^2 or fewer dense rectangular blocks. All nonzeros of G_{kl} are in a $m_k \times r_{kl}$ block

$$\hat{G_{kl}} \equiv \begin{bmatrix} H_{1kl} \\ \vdots \\ H_{pkl} \end{bmatrix}.$$

When G is block triangular as it is in SLRA, Section 5.3 shows that this makes it possible to use BLAS-2 or BLAS-3 operations in solving systems with coefficient matrix $I + G$. Note that the total number of nonzero elements in G is $\mathrm{nnz}(G) = \sum_{j,k,l} \mathrm{nnz}(H_{jkl}) = \sum_{k,l} m_k r_{kl} \le mM$. Furthermore, if no block of Q has rank higher than one, then $\mathrm{nnz}(G) \le mM \le p^3$. If Q is block-tridiagonal, with all nonzero blocks of rank one, then $\mathrm{nnz}(G) \le mM < 3p$.

5 The Parallel Method

The SMW formula, and the characteristics of the matrix G from Section 4, form the basis of the parallel algorithm for solving $Bx = y$. This algorithm is first stated for the direct low rank approximation (DLRA) preconditioner (1), and then the modifications needed for the BSSOR–like preconditioner SLRA (2) are described. For DLRA, there are three variants depending on how G is distributed among the processors and how systems with coefficient matrix $I + G$ are solved.

5.1 Initializing Preconditioner Structures for DLRA

Given the distribution of the data for D and Q from Section 2, initialization for DLRA is:

begin DLRA: Init
1. **for** each $k = 1 : p$ (in parallel)
   Preprocess D_{kk} for solving $D_{kk}x_k = y_k$ if necessary
2. **for** each $k = 1 : p$ (in parallel) /* Compute entries of G */
   for $l = 1 : p$ such that $(k, l) \in \mathcal{P}$
2a. Solve $D_{kk}\tilde{U}_{kl} = U_{kl}$
   for $j = 1 : p$ such that $(j, k) \in \mathcal{P}$
2b. Set $H_{jkl} = V_{jk}^{\mathrm{T}}\tilde{U}_{kl}$
2c. If desired, gather all block rows of G on a single processor, or redistribute them for parallel processing.
3. LU-factor $I + G$

end

Computing G consists of two steps. On step 2a, processor k computes $\tilde{U}_{kl}$ for all l's by solving the linear system with $\sum_l r_{kl}$ right-hand sides

$$D_{kk}\,[\,\tilde{U}_{k1} \quad \tilde{U}_{k2} \quad \cdots \quad \tilde{U}_{kp}\,] = [\,U_{k1} \quad U_{k2} \quad \cdots \quad U_{kp}\,]\,.$$

On step 2b, processor k computes the blocks G_{kl} for all values l, i.e. the blocks H_{jkl} in the matrix

$$\begin{bmatrix} H_{1k1} & \cdots & H_{1kp} \\ H_{2k1} & \cdots & H_{2kp} \\ \vdots & & \vdots \\ H_{pk1} & \cdots & H_{pkp} \end{bmatrix} = \begin{bmatrix} V_{1k}^{\mathrm{T}} \\ V_{2k}^{\mathrm{T}} \\ \vdots \\ V_{pk}^{\mathrm{T}} \end{bmatrix} \cdot [\,\tilde{U}_{k1} \quad \tilde{U}_{k2} \quad \cdots \quad \tilde{U}_{kp}\,]$$

As a result of this step, G is distributed among the processors by block row. An additional gather step is needed if G is to be stored on a single processor.

Unless G is triangular, some LU-factorization of $I+G$ is needed in step 3. If G is stored on a single processor, factoring is done by that processor; otherwise, it is done in parallel.

5.2 Applying the DLRA Preconditioner

$Bx = y$ with $B = C_{\mathrm{DIR}} = D+Q_C$ is solved using the SMW formula as follows:

begin DLRA: Solving $Bx = y$
1. **for** each $k = 1 : p$ (in parallel) /* $z = D^{-1}y$ */
   Solve $D_{kk}z_k = y_k$
2. **for** each $l = 1 : p$ (in parallel) /* $t = V^{\mathrm{T}}z$ */
   for $k = 1 : p$ such that $(k, l) \in \mathcal{P}$
   Set $t_{kl} = V_{kl}^{\mathrm{T}} z_l$
3. Solve $(I + G)s = t$ /* $s = (I + G)^{-1}t$ */
4. **for** each $k = 1 : p$ (in parallel) /* $y' = y - Us$ */
   Set $y'_k = y_k - \sum_l U_{kl}s_{kl}$
5. **for** each $k = 1 : p$ (in parallel) /* $x = D^{-1}y'$ */
   Solve $D_{kk}x_k = y'_k$

end

Only step 3 involves interprocessor communications. On step 2, the j-th processor produces all components of t in its block column (t_{kj} for all k); on step 4, the j-th processor needs all components of s in its block row (s_{jl} for all l). Depending on how $(I + G)s = t$ is solved, step 3 can be implemented in three different ways:

(i) Store and solve the $I+G$ system on a single processor. First, t is gathered on this processor; after solving, components of s are scattered to the processors that will need them at the next step. Both steps 2 and 4 send M floating-point values between the processors. One of the processors $1, \ldots, p$ that perform the parallel operations can be used for this, or $p+1$ processors can be used with processor $(p + 1)$ assigned the task.

(ii) Have each processor redundantly solve the $I + G$ system. This is suitable if the total M of ranks in the approximation is extremely small.

(iii) Use a parallel dense method, distributing the factors of $I + G$ among all processors. At most pM floating-point numbers are sent between processors. The factors can be distributed among the p processors by block row. If this results in poor load balance (as in the case of a block-triangular or block-tridiagonal Q), load balancing can be improved by distributing the factors of $I+G$ cyclically by row.

The preconditioner can be applied with one D-solve per iteration instead of two, by replacing steps 4 and 5 with

4$'$. **for** each $k = 1 : p$ (in parallel) $\qquad$ /* $x = z - \tilde{U}s$ */
 Set $x_k = z_k - \sum_l \tilde{U}_{kl} s_{kl}$

where $\tilde{U} = D^{-1}U$ (see step 2a of the preconditioner initialization algorithm). The disadvantage is that the DAXPY operation in 4$'$ uses dense vectors, while 5 uses sparse ones. The analysis in the remainder of the article assumes that 4 and 5 are used instead of 4$'$.

5.3 Modifications for the SLRA Preconditioner

The SMW formula can also be used for applying the SLRA preconditioner. Let $Q_L = U_L V_L^{\mathrm{T}}$ and $Q_R = U_R V_R^{\mathrm{T}}$, where U_L and V_L are assembled from the factors of Q_L, and U_R and V_R are assembled from the factors of Q_R in the manner analogous to (4–5). Then

$$C_{\mathrm{SOR}}^{-1} = D^{-1}\left[I - U_R(I+G_R)^{-1}V_R^{\mathrm{T}}D^{-1}\right]\left[I - U_L(I+G_L)^{-1}V_L^{\mathrm{T}}D^{-1}\right],$$

with $G_R = V_R^{\mathrm{T}}D^{-1}U_R$ and $G_L = V_L^{\mathrm{T}}D^{-1}U_L$. From Corollary 2, G_R and G_L are strictly block upper triangular and strictly block lower triangular, respectively.

The solution x of $C_{\mathrm{SOR}}x = y$ can be computed by the following procedure:

$$\begin{aligned} z &= D^{-1}y \\ t_L &= V_L^{\mathrm{T}}z \\ t_R &= V_R^{\mathrm{T}}z \\ s_L &= (I+G_L)^{-1}t_L \\ s_R &= (I+G_R)^{-1}(t_R - G_{RL}s_L) \\ x &= D^{-1}[y - U_R s_R - U_L s_L], \end{aligned} \tag{8}$$

with $G_{RL} = V_R^{\mathrm{T}}D^{-1}U_L$. The dimensions of the matrices G_R, G_L, and G_{RL} are $M_R \times M_R$, $M_L \times M_L$, and $M_R \times M_L$, where the M_R and M_L are the sums of the ranks of blocks of Q_R and Q_L, respectively. The dimension of the vectors t_R and s_R is M_R, and the dimension of t_L and s_L is M_L.

The SLRA initialization algorithm must compute the elements of G_R, G_L, and G_{RL}. None of these require factorization, since G_{RL} is used only in matrix-vector multiplications and the other two are strictly block-triangular.

However, parallel solution of systems with coefficient matrices $(I+G_R)$ and $(I+G_L)$ and parallel multiplication by G_{RL} require redistribution of G_R, G_L, and G_{RL} over the processors in the initialization phase. The load can be balanced during $(I+G)$-solves and G_{RL}-multiplications if each block row $[G_{k1}, G_{k2}, \cdots, G_{kp}]$ of these three matrices is partitioned into block sub-rows of approximately equal width, one block sub-row per processor. This way, each rectangular block of nonzeros $\hat{G_{kl}}$ is partitioned equally among the processors.

Like the DLRA algorithm, the SLRA solve involves two D-solves.

begin SLRA: Solving $C_{\mathrm{SOR}}x = y$
1. **for** each $k = 1:p$ (in parallel) /* $z = D^{-1}y$ */
   Solve $D_{kk}z_k = y_k$
2. **for** each $l = 1:p$ (in parallel) /* $t = V^{\mathrm{T}}z$ */
   for $k = 1:p$ such that $(k,l) \in \mathcal{P}$
      Set $t_{L,kl} = V_{L,kl}^{\mathrm{T}} z_l$ /* $t_L = V_L^{\mathrm{T}} z$ */
      Set $t_{R,kl} = V_{R,kl}^{\mathrm{T}} z_l$ /* $t_R = V_R^{\mathrm{T}} z$ */
3. Solve $(I+G_L)s_L = t_L$ /* $s_L = (I+G_L)^{-1}t$ */
4. Set $t_R = t_R - G_{RL}s_L$ /* $t'_R = t_R - G_{RL}s_L$ */
5. Solve $(I+G_R)s_R = t_R$ /* $s_R = (I+G_R)^{-1}t'_R$ */
6. **for** each $k = 1:p$ (in parallel) /* $y' = y - U_R s_R - U_L s_L$ */
   Set $y'_k = y_k - \sum_l U_{R,kl}s_{R,kl} - \sum_l U_{L,kl}s_{L,kl}$
7. **for** each $k = 1:p$ (in parallel) /* $x = D^{-1}y'$ */
   Solve $D_{kk}x_k = y'_k$

end

The two $(I+G)$-solves involve only block-triangular matrices. Together with the matrix-vector multiplication by G_{RL}, only these operations involve inter-processor communication.

6 Timing Models

The number of operations in applying the DLRA preconditioner is

$$C_{\mathrm{DIR}} = C_D + C_V + C_G + C_U + C_D = 2C_D + C_m + C_G.$$

The five terms of the first sum correspond to the five steps of the algorithm. C_D is for solving $Dz = w$, C_V is for one local dot product for each column of the blocks V_{kl}, C_U is for one DAXPY with each column of the blocks U_{kl}'s, and C_G is for solving $(I+G)s = t$. The sum $C_m = C_V + C_U$ is the number of operations for a matrix-vector multiplication $Qw = UV^{\mathrm{T}}w$.

A similar estimate for the SLRA solve algorithm gives

$$C_{\mathrm{SOR}} = 2C_D + 2C_m + C_{LUG},$$

where C_{LUG} is the sum of the numbers of operations needed for solving the linear systems with G_U and G_L, and for a matrix-vector multiplication with G_{UL}.

When solving $Bx = y$, only the $I + G$ system requires interprocessor communications; all other operations are parallel. From the operation count data above, the time t_{DIR} needed for $Bx = y$ is approximately

$$t_{\mathrm{DIR}} = 2t_D + t_V + t_U + t_G = 2t_D + t_m + t_G, \tag{9}$$

where t_D, t_V, t_U, and t_G are the times required to

t_D: solve $Dz = w$,
t_V: calculate a dot product with each column of each block V_{kl},
t_U: perform a DAXPY operation with each column of each block U_{kl},
t_G: solve a system with coefficient matrix $I + G$,

The sum $t_m = t_V + t_U$ is the time needed to calculate a matrix-vector product $Qw = UV^{\mathrm{T}}w$.

For the SLRA preconditioner,

$$t_{\mathrm{SOR}} = 2t_D + t_m + t_{LUG}, \tag{10}$$

where t_{LUG} is the time needed to solve the systems with G_L and G_U and to multiply by G_{UL}.

Cost of Solving $(I + G)s = t$: When Q is strictly block triangular (as in SLRA), so is G; then the system $(I + G)s = t$ can be solved in time $t_G^{\mathrm{seq}} = O(Mm)$ sequentially, or in time $t_G^{\mathrm{par}} = O(\max\{Mm/p, M\})$ in parallel. The max operator is needed because if $p > m$, only p processors may have work to do on a given step.

When G is not triangular, it still does not depend on the right-hand side y, and so $I + G$ can be LU-factored once for all solves with the preconditioner. The sequential solve time t_G^{seq} is $O(M^2)$; parallel solve time $t_G^{\mathrm{par}} = O(mM)$. This can be reduced if the block structure of Q is such such that fill-in is limited in some way. E.g., for a block-tridiagonal Q, t_G^{seq} is $O(mM)$ and $t_G^{\mathrm{par}} = O(\max\{Mm/p, M\})$.

For SLRA we can use the triangular-matrix estimate, having $t_{LUG}^{\text{seq}} = O(Mm)$ and $t_{LUG}^{\text{par}} = O(\max\{Mm/p, M\})$.

Costs of preparing G: Computing G involves solving M linear systems of the form $D_{jj}\tilde{u}_{ji} = u_{ji}$, and then calculating no more than Mm dot products of the form $v_{kj}^{\mathrm{T}}\tilde{u}_{ji}$. The total number of operations involved is under $m(C_D + C_U) \approx \frac{m}{2} C_{\text{DIR}}$. Since both D-solves and dot products can be done in parallel,

$$t_{\text{prepG}} \approx \frac{m}{2} t_{\text{DIR}}. \tag{11}$$

The cost of LU-factoring the matrix $I + G$ depends greatly on the structure of this matrix and thus on the structure of Q. For a general Q, the worst case operation count is $C_{\text{fact(I+G)}} = \mathcal{O}(M^3)$, which implies $t_{\text{fact(I+G)}}^{\text{seq}} = \mathcal{O}(M^3)$ for sequential processing and

$$t_{\text{fact(I+G)}}^{\text{par}} = \mathcal{O}(mM^2) \tag{12}$$

for factoring on p processors in parallel, where the p-th processor calculates the p-th block row of the factors. This result is better if Q has some special structure making LU-factoring G less expensive than $\mathcal{O}(M^3)$ operations. For example, if Q is block-tridiagonal, $t_{\text{fact(I+G)}}^{\text{seq}} = \mathcal{O}(m^2 M)$ and

$$t_{\text{fact(I+G)}}^{\text{par}} = \mathcal{O}(\max\{Mm, Mm^2/p\}). \tag{13}$$

7 Which Ranks Are Low?

The term "low rank" for blocks of Q has been used until now without specification. The timing model provides an operational definition:

(a) The time spent preparing and factoring $I + G$ must be small compared to the iterative solve time for $Au = f$, and
(b) t_G must be small relative to $t_D + t_U$, so that

$$t_{\text{DIR}} \approx 2t_D + t_m. \tag{14}$$

Below we will show what restrictions these conditions impose on m for a general matrix Q as well as for the special cases of block-triangular and block-tridiagonal Q. We suppose that:

(i) p processors are used.
(ii) The maximum dimension of a block D_{kk} is $\mathcal{O}(n/p)$.

(iii) $m = \mathcal{O}(M/p)$, i.e. the maximum sum of block ranks in a block row of Q is of the same order as the average sum.
(iv) Factoring $I + G$ and solving systems with it are done in parallel on p processors. (Estimates for sequential operations can be obtained in a similar way.)
(v) t_D is the dominant term in $t_D + t_m$, i.e. $t_D + t_m = \mathcal{O}(t_D)$. This is usually the case when discretized PDEs are solved.

The time t_D spent solving the system $Dz = w$ depends on the structure of D. The two obvious limiting cases are $t_D = \Theta(n/p)$, which occurs when D_{kk} is a diagonal matrix or a product of LU-factors with few nonzeros per row; and $t_D = \Theta(n^2/p^2)$, which occurs when D_{kk} is a product of dense LU factors. The estimates below are for Q of arbitrary of block-triangular, and block-tridiagonal block structure,

Equation (11) indicates that t_{prepG} is on the order of the time taken by m iterations, so criterion (a) implies that m needs to be small in comparison to the total number of iterations.

In order for the term $t_{\mathrm{fact(I+G)}}$ not to make this ratio much worse, it needs to be of the same order as t_{prepG}, i.e., $t_{\mathrm{fact(I+G)}} = \mathcal{O}(mt_D)$. For a block-triangular Q this holds trivially. For a general Q, the parallel LU-factoring time estimate (12) is used to rewrite the condition as

$$m = \begin{cases} \mathcal{O}\left(n^{1/2}p^{-3/2}\right), & \text{if } t_D = \Theta\left(n/p\right), \\ \mathcal{O}\left(np^{-2}\right), & \text{if } t_D = \Theta\left(n^2/p^2\right) \end{cases} \tag{15}$$

For a block-tridiagonal Q, the parallel LU-factoring time estimate (13) gives

$$m = \begin{cases} \mathcal{O}\left(\min\{\frac{n}{p^2}, \frac{n^{1/2}}{p^{1/2}}\}\right), & \text{if } t_D = \Theta\left(n/p\right), \\ \mathcal{O}\left(\min\{\frac{n^2}{p^3}, \frac{n}{p}\}\right), & \text{if } t_D = \Theta\left(n^2/p^2\right). \end{cases} \tag{16}$$

For an arbitrary Q, the time t_G needed to solve $I + G$ by a parallel dense method is $\mathcal{O}(mM)$. Thus the criterion (b) requires that

$$m = \begin{cases} o\left(n^{1/2}p^{-1}\right), & \text{if } t_D = \Theta\left(n/p\right), \\ o\left(np^{-3/2}\right), & \text{if } t_D = \Theta\left(n^2/p^2\right). \end{cases} \tag{17}$$

For a block-triangular or a block-tridiagonal Q we similarly obtain

$$m = \begin{cases} o\left(\min\{\frac{n}{p^2}, \frac{n^{1/2}}{p^{1/2}}\}\right), & \text{if } t_D = \Theta\left(n/p\right), \\ o\left(\min\{\frac{n^2}{p^3}, \frac{n}{p}\}\right), & \text{if } t_D = \Theta\left(n^2/p^2\right). \end{cases} \tag{18}$$

Obviously, condition (15) is stronger than condition (17), but for a block-tridiagonal Q (18) is stronger than (16).

If Q is ...	To ensure that...	If $t_D = \Theta(n/p)$	If $t_D = \Theta(n^2/p^2)$
arbitrary	$t_{\text{fact(I+G)}} = \mathcal{O}(mt_D)$	$m = \mathcal{O}\left(\frac{n^{1/2}}{p^{3/2}}\right)$	$m = \mathcal{O}\left(\frac{n}{p^2}\right)$
	$t_G = o(t_D + t_m)$	$m = o\left(\frac{n^{1/2}}{p}\right)$	$m = o\left(\frac{n}{p^{3/2}}\right)$
block-tridiag.	$t_{\text{fact(I+G)}} = \mathcal{O}(mt_D)$	$m = \mathcal{O}\left(\min\{\frac{n}{p^2}, \frac{n^{1/2}}{p^{1/2}}\}\right)$	$m = \mathcal{O}\left(\min\{\frac{n^2}{p^3}, \frac{n}{p}\}\right)$
	$t_G = o(t_D + t_m)$	$m = o\left(\min\{\frac{n}{p^2}, \frac{n^{1/2}}{p^{1/2}}\}\right)$	$m = o\left(\min\{\frac{n^2}{p^3}, \frac{n}{p}\}\right)$
blk-triang.	$t_G = o(t_D + t_m)$	$m = o\left(\min\{\frac{n}{p^2}, \frac{n^{1/2}}{p^{1/2}}\}\right)$	$m = o\left(\min\{\frac{n^2}{p^3}, \frac{n}{p}\}\right)$

Table 1
Low-rank criteria when $(I + G)s = t$ is solved in parallel.

If Q is ...	To ensure that...	If $t_D = \Theta(n/p)$	If $t_D = \Theta(n^2/p^2)$
arbitrary	$t_{\text{fact(I+G)}} = \mathcal{O}(mt_D)$	$m = \mathcal{O}\left(\frac{n^{1/2}}{p^2}\right)$	$m = \mathcal{O}\left(\frac{n}{p^{5/2}}\right)$
	$t_G = o(t_D + t_m)$	$m = o\left(\frac{n^{1/2}}{p^{3/2}}\right)$	$m = o\left(\frac{n}{p^2}\right)$
block-tridiagonal	$t_{\text{fact(I+G)}} = \mathcal{O}(mt_D)$	$m = \mathcal{O}\left(\frac{n^{1/2}}{p}\right)$	$m = \mathcal{O}\left(\frac{n}{p^{3/2}}\right)$
	$t_G = o(t_D + t_m)$	$m = o\left(\frac{n^{1/2}}{p}\right)$	$m = o\left(\frac{n}{p^{3/2}}\right)$
block-triangular	$t_G = o(t_D + t_m)$	$m = o\left(\frac{n^{1/2}}{p}\right)$	$m = o\left(\frac{n}{p^{3/2}}\right)$

Table 2
Low-rank criteria when $(I + G)s = t$ is solved sequentially.

Restrictions on m to satisfy criterion (b) can be derived straightforwardly from the earlier estimates of t_G. If $n > p^3$, which is a reasonable assumption for most parallel architectures and practical problem sizes, only the second values in the formulas (16,18) with the min operator apply.

Table 1 summarizes the order estimates for parallel factorization of and solution with $I + G$, and Table 2 presents results for the sequential case.

Suppose that A is the matrix for a discretized PDE on a cubic domain divided into cubic subdomains, and Q_C (or $Q_L + Q_R$) is composed of the original off-diagonal blocks of A. In this case, $m = \mathcal{O}(n^{1/2}/p^{1/2})$ in 2D or $m = \mathcal{O}(n^{2/3}/p^{2/3})$ in 3D. So only for SLRA with parallel $(I + G)$-solves will the term t_D be dominant. For DLRA or for SLRA with sequential $(I+G)$-solves, the ranks of the blocks of Q must be reduced from those of A in order for t_D to be dominant in (9) or (10), i.e. for the DLRA or SLRA iteration time to be comparable with that of a Jacobi iteration. Some techniques for designing such low rank approximations are discussed in [3].

8 Testing Results

The numerical experiments use the preconditioners within a biconjugate gradients stabilized [4] iterative method. The program is written in pC++, a parallel extension [5] of C++, and run on a ten-processor SGI Power Challenge with 75 MHz R8000 chips running the 64 bit IRIX 6.1 operating system. The machine has 2Gbytes of main memory, and each processor accesses memory through a 16 Kbyte primary and 4 Mbyte secondary cache. Optimized BLAS were used for dense vector operations. The algorithms were implemented with a parallel factorization and solution of $I + G$ sytems. Table 3 summarizes the methods tested. The preconditioners either used no off–diagonal blocks, the original off–diagonal blocks of A, or a low rank approximation obtained by a lumping-based low rank approximation method [3] of the respective blocks of A, with rank $Q_{kl} \leq 3$. For comparison, the BSSOR–like preconditioners SORG and SLRA were also tested using conventional block triangular forward and backward solves; these are called SORG-LU and SLRA-LU respectively. All other preconditioners with off–diagonal blocks use the SMW formula for solving the systems defined by the "preconditioned system matrix" in Table 3. The matrix D is obtained from an ILU(3) factorization of the corresponding diagonal blocks, computed in parallel.

		Off–Diagonal Blocks:		
		None	Original	Low Rank Approximation
Precond	D	Jacobi		
System	$D + Q_C$		DORG	DLRA
Matrix:	$(D + Q_L)D^{-1}$		SORG	SLRA
	$*(D + Q_R)$		SORG-LU	SLRA-LU

Table 3
Methods Tested

The test problem is from the fluid dynamics steady state modeling of a backward facing step in 2D [9]. The resulting matrix is of order $n = 20284$ and has 452752 nonzeros. This test problem was chosen because simple block diagonal (Jacobi) preconditioning allows convergence, and one purpose of our testing is to compare the parallelization of the low rank approximation method with Jacobi preconditioning. Tests were run using $p = 8$ and 1,2,4, and 8 processors. The system was partitioned into eight subdomains using a spectral method from the Chaco [7,8] package for domain decomposition.

For A and the preconditioners with A's original off–diagonal blocks, the sum of the ranks of the off–diagonal blocks was $M = 2214$; for the preconditioners with low rank approximations, $M = 66$.

Figure 1 compares the cost per iteration of the different preconditioners. The vertical axis is CPU time per iteration multiplied by the number of processors. This time includes matrix-vector products, vector operations, preconditioner application, and synchronization and communication time between processors. The product of time and the number of processors P measures parallelism: for a linear speed-up, it will be a constant independent of P; for an inherently non-parallelizable algorithm, it will grow with P.

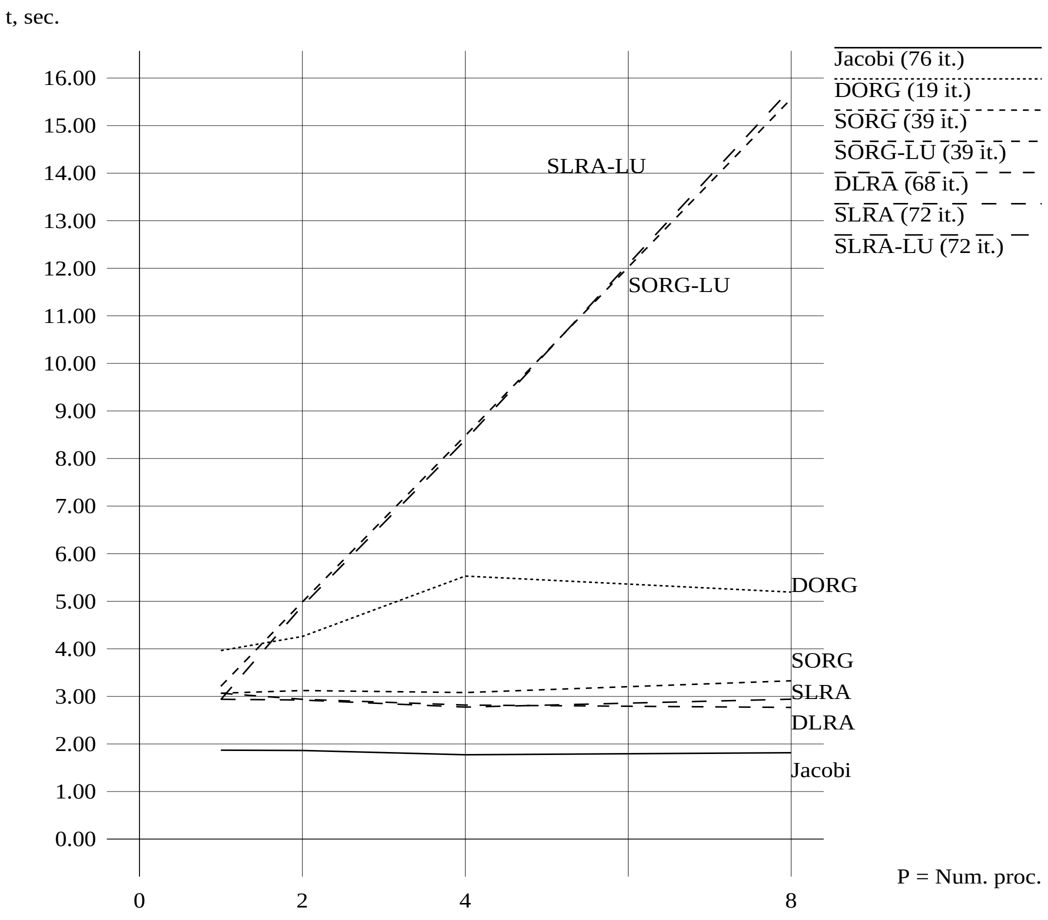

Fig. 1. Product of P and time per iteration for methods.

In Figure 1 the conventional BSSOR preconditioned method does achieve some speedup because the other operations of the iterative method are efficiently parallelized. However, the block dependencies clearly make it unsuitable for parallel computing since the parallel efficiency E_P for $P = 2, 4$, and 8 processors is $E_2 = 64\%$, $E_4 = 37\%$, and $E_8 = 20\%$, respectively.

By comparison, the SMW-based method provide effective parallelization for both SLRA ($E_p > 99\%$) and SORG ($E_p > 92\%$) preconditioning. The cost of an iteration with these preconditioners was 1.5–1.9 times that of block diagonal preconditioning, bearing out the analysis of Section 6 which bounded the

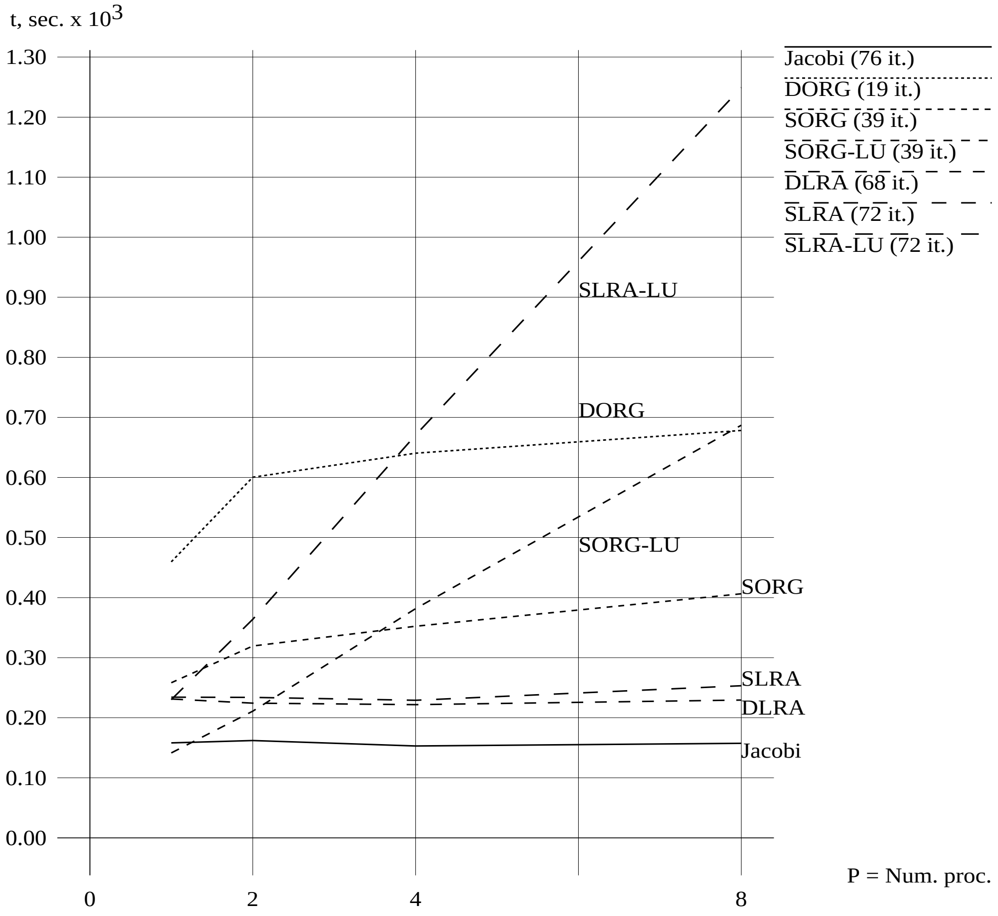

Fig. 2. Product of P and total time for methods.

preconditioning application by the cost of two applications of block diagonal preconditioning, plus a small amount for low rank matrix-vector products. The cost ratio is actually below 2.0 because of the parallelization of operations other than preconditioning in the iterative method.

The SMW-based method is also highly parallel when the DLRA preconditioner is applied; for it, the measured speed-up was even superlinear. Superlinear speedups typically occur on the SGI Power Challenge because each processor has its own two–level cache system, and using more processors brings more cache memory to bear. The cost of an iteration with DLRA was about 1.65 that of the block-diagonal preconditioner. However, for the test problem used here its convergence was not much faster than that of block diagonal preconditioning: 68 iterations vs. 76.

Solving $(I+G)s = t$ becomes more costly when DORG, direct preconditioning with A's original off–diagonal blocks, is used. Communication overhead drops the parallel efficiency E_p to 71%–76%, and the cost of an iteration was 2–3 times as high as that of a block diagonal preconditioned iteration. However, convergence was significantly faster: only 19 iterations vs. 72 iterations for block-Jacobi or 39 iterations for BSSOR with original blocks.

These results confirm that the SMW-based method can provide good parallel efficiency while maintaining and in the case of DLRA even improving the quality of preconditioning compared to BSSOR.

The total time for a solution, shown in Figure 2, includes the cost of preparing and factoring G, but not factoring the diagonal blocks since that cost is identical for all the methods. The graph for the SMW-based method of applying the DORG preconditioner indicates its high setup cost. Still, even at 8 processors its parallelism has allowed it to outperform conventionally applied BSSOR.

9 Summary

This paper has shown that in the parallel iterative solution of large sparse linear systems of equations, high parallel efficiencies can be obtained by using low rank approximations of off-diagonal blocks combined with a SMW formula approach. Order estimates have been developed showing when this approach is practical, and indicate that for systems whose order n is significantly larger than the number P of processors, preconditioners based on low rank approximations can be implemented with roughly twice the cost per iteration of a block diagonal preconditioner. The results also apply to BSSOR–like preconditioning, with the off diagonal blocks of A replaced by low rank approximations. The order estimates also provide a practical definition of "low rank" for this application.

Numerical testing both validates the analysis and shows good parallel efficiencies for this approach. The tests also show that when block diagonal Jacobi preconditioning works, it is the cheapest and most parallel of the methods – hardly a surprising result. However, the test problem used here was chosen to allow Jacobi preconditioning to succeed in order to allow comparisons of parallelism. In the steady state fluid flow application from which it was drawn we often encounter systems for which block Jacobi preconditioning fails, while methods such as ours that account for inter-subdomain interactions succeed.

References

[1] F. Alvarado, *Ordering schemes for partitioned sparse inverses*, 1989. SIAM Symposium on Sparse Matrices, Salishan Lodge, Gleneden Beach, OR, May 22–24 1989.

[2] E. Anderson, *Parallel implementation of preconditioned conjugate gradient methods for solving sparse systems of linear equations*, Master's thesis, Univ. of Illinois at Urbana-Champaign, Center for Supercomputing Res. & Dev., 1988.

[3] R. Bramley and V. Meñkov, *Low rank off-diagonal block preconditioners for solving sparse linear systems on parallel computers*, Tech. Rep. 446, Department of Computer Science, Indiana University, Bloomington, IN, 1996. Available via World Wide Web at http://ftp.cs.indiana.edu /pub/vmenkov/lowrank/tr446.dvi.

[4] H. V. der Vorst, *Bi–CGSTAB: A fast and smoothly converging variant of Bi–CG for the solution of nonsymmetric linear systems*, SIAM Journal of Scientific and Statistical Computing, 13 (1992), pp. 631–644.

[5] D. Gannon, S. X. Yang, and P. Beckman, *User Guide for a Portable Parallel C++ Programming System, pC++*, Department of Computer Science and CICA, Indiana University, Bloomington, IN, 1994. Available via World Wide Web at http://www.extreme.indiana.edu/sage/pcxx_ug/pcxx_ug.html.

[6] A. George and W.-H. Liu, *The Computer Solution of Large Sparse Positive Definite Systems*, Prentice-Hall, Engelwood Cliffs, NJ, 1981.

[7] B. Hendrickson and R. Leland, *The Chaco User's Guide Version 1.0*, Tech. Rep. SAND 93-2339, Sandia National Laboratories, Albuquerque, N.M., October 1993.

[8] ——, *An Improved Spectral Graph Partitioning Algorithm for Mapping Parallel Computations*, SIAM Journal on Scientific Computing, (1995), pp. 452–469.

[9] J.-F. Hétu and D. Pelletier, *Fast, adaptive finite element scheme for viscous incompressible flow*, AIAA Journal, 30 (1992), pp. 2677–2681.

[10] J. Meijerink and H. van der Vorst, *An iterative solution method for linear systems of which the coefficient matrix is a symmetric M-matrix*, Math. Comp., 31 (1977), pp. 148–162.

[11] J. M. Ortega and W. C. Rheinboldt, *Iterative Solution of Nonlinear Equations in Several Variables*, Academic Press, New York, 1970.

[12] Y. Saad, *Iterative methods for sparse linear systems*, PWS publishing, New York, 1996.

[13] A. Yeremin and L. Kolotilina, *On a family of two–level preconditionings of the incomplete block factorization type*, Sov. J. Numer. Anal. Math. Modeling, 1 (1986), pp. 293–320.