OVERLAPPING BLOCKS BY GROWING A PARTITION WITH APPLICATIONS TO PRECONDITIONING*

DAVID FRITZSCHE†, ANDREAS FROMMER†, STEPHEN D. SHANK‡, AND DANIEL B. SZYLD‡

Abstract. Starting from a partitioning of an edge-weighted graph into subgraphs, we develop a method which enlarges the respective sets of vertices to produce a decomposition with *overlapping* subgraphs. The vertices to be added when growing a subset are chosen according to a criterion which measures the strength of connectivity with this subset. By using our method on the (directed) graph associated with a matrix, we obtain an overlapping decomposition of the set of variables which can be used for algebraic additive and multiplicative Schwarz preconditioners. We present a complexity analysis of this block-growing method, thus proving its computational efficiency. Numerical results for problems stemming from various application areas show that with these overlapping Schwarz preconditioners we usually substantially improve GMRES convergence as compared to preconditioners based on a nonoverlapping decomposition, or an overlapping decomposition based on level sets without other criteria, as well as incomplete LU.

Key words. block partitioning, overlapping blocks, algebraic Schwarz methods, multiplicative Schwarz methods, preconditioning

AMS subject classifications. 65F10, 65F08, 05C85

DOI. 10.1137/100803821

1. Introduction. We consider a nonsingular linear system

$$(1.1) \qquad Ax = b, \qquad A \in \mathbb{R}^{n \times n}, \quad x, b \in \mathbb{R}^n,$$

with n large and A sparse. We do not assume A to be symmetric. Additive and multiplicative Schwarz preconditioners are known to be efficient iterative methods for solving (1.1) arising from discretizations of (elliptic) partial differential equations. They are based on decomposing the physical domain into a set of overlapping subdomains. Given a current approximation for the solution, appropriate restrictions of the original partial differential equation to each subdomain are solved and combined to yield the next iterate [27], [28]. In the descriptions of Schwarz preconditioners in these references it can be appreciated that the use of overlap is fundamental for the convergence of the associated methods. The idea can be applied to a general linear system of the form (1.1) where, instead of subdomains of a physical domain, we consider overlapping subsets of the variables. In this paper, we deal with such *algebraic* Schwarz methods [2].

An important issue in applying algebraic Schwarz methods is the question of how to determine the overlapping subsets. In terms of the graph $\mathcal{G} = \mathcal{G}(A)$ of the matrix A we have to find overlapping subgraphs ("blocks") of $\mathcal{G}$. The OBGP (Overlapping

Blocks by Growing a Partition) algorithm to be proposed here is a new approach to compute overlapping subgraphs. In [3], a simple strategy was proposed in which each subgraph of an existing nonoverlapping partitioning is enlarged by adding all the nodes adjacent to it. This is the default strategy in the software PETSc [1]. Our proposed variant works by taking an existing nonoverlapping partitioning and growing each block to include the most appropriate vertices from other blocks, not necessarily including all adjacent nodes, but often including some others. Algebraic Schwarz methods based on the thus-obtained groups of overlapping subsets of variables can be used as preconditioners for modern iterative solvers. Since the preconditioner may use direct methods, this approach can be considered a hybrid method; see, e.g., [11] for another example of such hybrid methods. We mention that hybrid methods are increasingly popular as they adapt well to the memory hierarchies of modern computers.

After a short overview of the required concepts from graph theory in section 2, we describe algebraic Schwarz methods in some detail in section 3. We then elaborate on our OBGP algorithm in section 4 and discuss its efficient implementation in section 4.2. In section 5 we perform a detailed analysis of the time complexity of OBGP. We finally give results of numerical experiments in section 6 and present our conclusions in section 7.

2. Notions from graph theory. In this section we review some concepts from graph theory needed in this paper.

A directed graph, or digraph, is a pair $\mathcal{G} = (V, E)$ of vertices V and edges E, where $E \subseteq \{(i, j) : i, j \in V, i \neq j\}$. Given a subset V' of the vertices V, the induced subgraph $\mathcal{G}|_{V'}$ has vertices V' and edges $E' = \{(i, j) : i, j \in V', (i, j) \in E\}$. We define the degree of $v \in V$ as the number of edges incident to v, i.e., $\deg(v) := |\{(v, w) \in E\}|$.

We say that vertex $i \in V$ is *incident* with edge $e = (k, \ell)$ if $i = k$ or $i = \ell$. Note that this terminology makes no distinction between the starting point k and the end point ℓ of the edge e. We also say that the edge $e = (k, \ell)$ is incident to either of its vertices k or ℓ. The two vertices k and ℓ incident to $e = (k, \ell)$ are called adjacent. Given a set of vertices $S \subseteq V$, the adjacency set $\mathrm{adj}(S)$ contains all vertices which are not in S but which are adjacent to a vertex in S; i.e.,

$$\mathrm{adj}(S) = \big\{ j \in V : j \notin S \text{ and } j \text{ is adjacent to some } i \in S \big\}.$$

This definition is somewhat nonstandard, because a vertex ℓ belongs to $\mathrm{adj}(S)$ irrespective of whether it is the starting or the end vertex of a (directed) edge e with its other incident vertex from S.

Adjacency sets $\mathrm{adj}(S)$ can be generalized to higher levels according to the following definition.

DEFINITION 2.1. *Let $\mathcal{G} = (V, E)$ be a graph and $S \subset V$. The kth level set $L_k(S)$ with respect to S is defined as*

$$L_k(S) := \begin{cases} S & \text{if } k = 0, \\ \mathrm{adj}(S) & \text{if } k = 1, \\ \mathrm{adj}\big(L_{k-1}(S)\big) \setminus L_{k-2}(S) & \text{if } k > 1. \end{cases}$$

Figure 2.1 shows for some graph $G = (V, E)$ and some node set S the level sets $L_0(S)$ through $L_4(S)$.

We need some additional notation to describe the edges which are incident with sets of vertices.
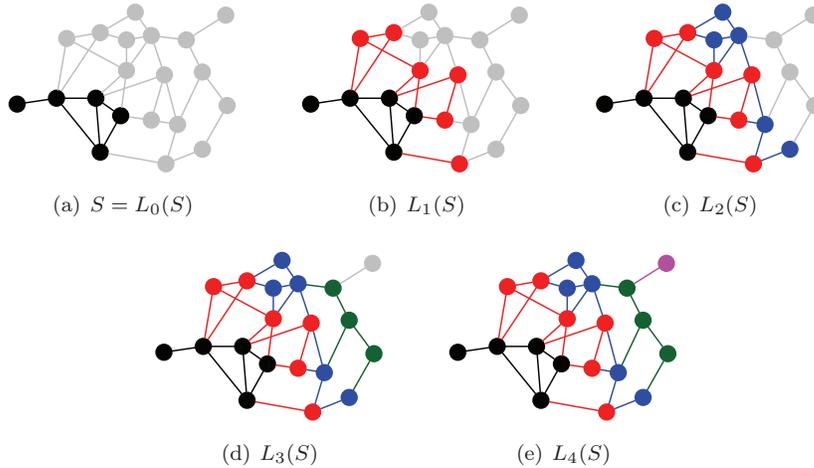
(a) $S = L_0(S)$ (b) $L_1(S)$ (c) $L_2(S)$

(d) $L_3(S)$ (e) $L_4(S)$

FIG. 2.1. *Level sets $L_0(S)$ through $L_4(S)$ with respect to the node set S, and their incident edges.*

DEFINITION 2.2. *Let $\mathcal{G} = (V, E)$ be a graph. The notation $\mathrm{inc}(v)$ is used to describe the set of edges incident to $v \in V$. For a set of nodes $S \subset V$ we define $\mathrm{inc}(S)$ to be the set of edges in E which are incident to some node in S, i.e.,*

$$\mathrm{inc}(S) = \bigcup_{v \in S} \{e \in E : e \text{ incident to } v\} = \text{edges in } \mathcal{G}|_{S \cup \mathrm{adj}(S)}.$$

The set $\mathrm{inc}(S)$ is called the set of edges incident to S.
    For two sets $S, T \subset V$ define

$$\mathrm{inc}(S, T) := \mathrm{inc}(S) \cap \mathrm{inc}(T)$$

to be the set of edges incident to both S and T.
    Any square matrix $A \in \mathbb{R}^{n \times n}$ possesses an associated digraph $\mathcal{G}(A) = (V, E)$ given by

$$V = \{1, \dots, n\}, \quad E = \{(i, j) : a_{ij} \neq 0, i \neq j\}.$$

3. Algebraic Schwarz preconditioners. The original Schwarz alternating method is a theoretical tool for proving the existence of solutions of elliptic partial differential equations for nonstandard domains [25]. The idea is to subdivide the domain Ω into several, possibly overlapping domains of simpler form, and to solve the equation alternatingly on the subdomains. In each local solve, the Dirichlet boundary conditions on the artificial interface inside Ω are given by the latest available approximation of the global solution. This new local solution is then used to update the approximation to the global solution; see, e.g., [27] or [28] for details.

The algebraic multiplicative Schwarz method is an abstract version of the original alternating method applied to the solution of an arbitrary linear system. The basic idea is still the same: We alternatingly use local solves to update the approximation of the global solution. The partitioning into subdomains now corresponds to a partitioning of the set of variables, and the local problems to solve are linear systems given by the diagonal blocks of the matrix corresponding to the respective set of variables. We now introduce the notation needed to fully describe the algebraic Schwarz approach.

Let $A \in \mathbb{R}^{n \times n}$ be the matrix of the linear system (1.1) and $V = \{1, \ldots, n\}$ the set of vertices in the directed graph $\mathcal{G}(A)$ associated with A.

DEFINITION 3.1. *A family $\mathcal{V} = \{V_i\}_{i=1,\ldots,q}$ of nonempty subsets $V_i \subset V$ is called a decomposition of V if $\bigcup_{i=1}^{q} V_i = V$. A decomposition is called a partitioning if the sets V_i are pairwise disjoint, i.e., if $V_i \cap V_j = \emptyset$ for $i \neq j$. Let $n_i = |V_i|$. We denote the $n_i = |V_i|$ elements of V_i by $v_1^{(i)}, \ldots, v_{n_i}^{(i)}$.*

A partitioning induces a permutation of the elements of V.

DEFINITION 3.2. *Let $\mathcal{V} = \{V_1, \ldots, V_q\}$ be a partitioning of $V = \{1, \ldots, n\}$ with $n_i = |V_i|$. The permutation $\pi : V \to V$ with respect to the partitioning $\mathcal{V}$ is defined as $\pi(1) = v_1^{(1)}, \ldots, \pi(n_1) = v_{n_1}^{(1)}, \pi(n_1 + 1) = v_1^{(2)}, \ldots, \pi(n) = v_{n_q}^{(q)}$.*

We now associate with each node $i \in V$ the unit vector $e_i \in \mathbb{R}^n$, and with a whole node set $S \subset V$ the linear subspace spanned by the unit vectors associated with the nodes from S. Thus, V is associated with $\mathbb{R}^n$ and a subset $S = \{s_1, \ldots, s_m\} \subset V$ is associated with the linear subspace $\text{span}\{e_{s_1}, \ldots, e_{s_m}\} \subset \mathbb{R}^n$.

DEFINITION 3.3. *Let $S = \{s_1, \ldots, s_m\} \subset V$ be a subset of the node set $V = \{1, \ldots, n\}$. The restriction operator $R_S \in \mathbb{R}^{m \times n}$ onto the subspace associated with S is defined by*

$$(R_S)_{ij} := \delta_{s_i, j} = \begin{cases} 1 & \text{if } s_i = j, \\ 0 & \text{otherwise.} \end{cases}$$

The transpose R_S^T of a restriction operator R_S is the corresponding prolongation op-erator.

Note that row k of R_S is just row s_k of the identity I_n.

In most cases we consider restrictions belonging to an element V_i of a decomposi-tion or partitioning $\mathcal{V}$ of V. To simplify our notation we use the following abbreviation:

$$R_i := R_{V_i} \in \mathbb{R}^{n_i \times n}.$$

We use this subscript notation also to indicate submatrices and subvectors.

DEFINITION 3.4. *Let $\mathcal{V} = \{V_i\}_{i=1,\ldots,q}$ be a decomposition of the node set $V = \{1, \ldots, n\}$. The submatrix A_{ij} of a matrix $A \in \mathbb{R}^{n \times n}$ is defined by*

$$A_{ij} := R_i A R_j^T \in \mathbb{R}^{n_i \times n_j}.$$

Let $S = \{s_1, \ldots, s_m\} \subset V$ be a subset of V. The submatrix A_S is defined by

$$A_S := R_S A R_S^T \in \mathbb{R}^{m \times m}.$$

For simplicity, we will write A_i for $A_{ii} = A_{V_i}$. Finally, the subvector z_i *of a vector $z \in \mathbb{R}^n$ is defined by*

$$z_i := R_i z \in \mathbb{R}^{n_i}.$$

For a given vector v, the algebraic multiplicative Schwarz preconditioner cycles through the blocks to update the current approximation z to $A^{-1}v$ as follows: It first computes the current residual $v - Az$, restricts it to the block V_i (resulting in $R_i(v - Az)$), and then corrects the current approximation z by adding the prolongated solution of the local error equation

$$A_i \underline{e}_i = R_i(v - Az).$$

Algorithm 1 Multiplicative Schwarz preconditioner.

1: **input:** vector v
2: **output:** approximation $z = M^{-1}v$ to $A^{-1}v$ $\{z = (I - Q)A^{-1}v\}$
3:    $z = 0$
4: **for** $i = 1, \ldots, q$ **do**
5:      $z := z + R_i^T A_i^{-1} R_i(v - Az)$ $\{z = (I - Q_i)A^{-1}v\}$
6: **end for**

This is formulated as Algorithm 1, where we include comments which indicate why the computed vector z can be represented as

$$z = (I - Q)A^{-1}v,$$

where

$$(3.1) \qquad Q = Q_q, \; Q_i = (I - R_i^T A_i^{-1} R_i A)Q_{i-1}, \quad i = 1, \ldots, q \text{ with } Q_0 = I;$$

see, e.g., [27], [28].

The multiplicative Schwarz preconditioning step can be viewed as the first iteration step of the stationary iteration

$$(3.2) \qquad x^{(k+1)} = Qx^{(k)} + (I - Q)A^{-1}v, \quad k = 0, 1, \ldots, \; x^{(0)} = 0,$$

to solve $Ax = v$. We refer the reader to, e.g., [2], [17], [27] for convergence results for this iteration in the case where A is symmetric positive definite or an M-matrix. Performing one or more steps of a convergent iteration for the system $Ax = v$ can be viewed as computing an approximation to $A^{-1}v$. Therefore such an iteration can be used as a preconditioner.

The block Gauss–Seidel method has the same basic idea as the multiplicative Schwarz method: For each block the current residual is restricted to the block (one variable in the point method) to compute the local error which is prolongated to a correction of the current approximation. In fact, the multiplicative Schwarz method with a nonoverlapping decomposition is identical to the block Gauss–Seidel method.

Note that an implementation of Algorithm 1 would not need to compute the full vector $v - Az$ in the expression $z := z + R_i^T A_i^{-1} R_i(v - Az)$, because the restriction R_i can be applied to A before the multiplication with z. If this is done carefully and if the blocks are nonoverlapping, then in total only one matrix-vector multiplication with A is needed. Furthermore, the preconditioned matrix-vector multiplication can be further optimized using a technique similar to Eisenstat's trick [12]; see [13], [14].

We denote the multiplicative Schwarz method based on the decomposition $\mathcal{V} = \{V_i\}_{i=1,\ldots,q}$ by $\mathrm{MS}(\{V_i\}) = \mathrm{MS}(V_1, \ldots, V_q)$. If we apply the multiplicative Schwarz method for two different decompositions, we usually end up with two different iterations. Let MS_1 and MS_2 be multiplicative Schwarz iterations based on two different decompositions. We call MS_1 and MS_2 equivalent if their iteration matrices Q_{MS_1} and Q_{MS_2} are the same; here Q_{MS_1} and Q_{MS_2} refer to Q in (3.1). This definition of equivalence implies that $x_{\mathrm{MS}_1}^{(k+1)}$ and $x_{\mathrm{MS}_2}^{(k+1)}$ computed by (3.2) are equal for all choices of $x^{(k)}$ and b.

If a decomposition contains a set of vertices that is not connected, then we can create a new decomposition by splitting this set into its connected components. As the following theorem shows, the multiplicative Schwarz methods based on these two decompositions are equivalent if we keep the ordering of the sets.

THEOREM 3.5. *Let $\mathcal{G}(A) = (V, E)$ be the digraph of a matrix $A \in \mathbb{R}^{n \times n}$. Let $\{V_i\}_{i=1,\ldots,q}$ be a decomposition of V. Let $j \in \{1, \ldots, q\}$ be such that $\mathcal{G}|_{V_j}$ is not connected. Let $C_1, \ldots, C_m$ be the connected components of $\mathcal{G}|_{V_j}$. Then*

$$\mathrm{MS}(V_1, \ldots, V_q) = \mathrm{MS}(V_1, \ldots, V_{j-1}, C_1, \ldots, C_m, V_{j+1}, \ldots, V_q).$$

For a proof of Theorem 3.5 see [13]. An interpretation of this result is that we should aim at maintaining connectivity of the node sets when we grow them into an overlapping decomposition.

Since we also show experiments for additive and restricted additive Schwarz preconditioners, we briefly describe them here. Additive Schwarz is equivalent to the classical Jacobi iterative method when there is no overlap, and instead of the process (3.1), we simply have

$$(3.3) \qquad z = \left(I - \sum_{i=1}^{q} R_i^T A_i^{-1} R_i\right) v;$$

see, e.g., [27], [28] for traditional descriptions, as well as [2], [15] for algebraic analyses. Restricted additive Schwarz (RAS) consists of replacing in (3.3) the prolongation operator R_i^T, which has the variables with overlap, with a prolongation operator *restricted* to variables without the overlap; see [4], [16] for further details.

4. The OBGP algorithm. The general idea of the OBGP (overlapping blocks by growing a partition) algorithm is to take an existing nonoverlapping partitioning, e.g., a partitioning computed by Metis [19] or PABLO [5], [14], [20], and extend the blocks so that they have some overlap. In this way, OBGP can be seen as an enhancement of existing high-quality graph partitioners.

The extension of a block is done separately for each block in the partitioning and is completely independent of the extensions of the other blocks. Our new graph-based algorithm is based on the graph connectivity, and it yields good results in many cases where simpler strategies to create an overlap are not useful. Examples where simple strategies do not work well are easy to construct; see, e.g., [13].

4.1. Growing a block. Let $V_1, \ldots, V_q$ be an existing (nonoverlapping) partitioning of V, i.e., $\bigcup_{i=1}^{q} V_i = V$ and $V_i \cap V_j = \emptyset$, $i \neq j$. The task of OBGP is to grow this partitioning into an overlapping decomposition $W_i, \ldots, W_q$ with $W_i \supsetneq V_i$. Let $B = V_i$ for some i be the node set of a block we want to grow. For the remainder of this section, we assume that the index i is fixed.

In order to contain the computational effort, we want to restrict the amount of nodes considered for inclusion into B. Since Theorem 3.5 shows that we should aim at maintaining connected sets when we add the overlap, we want to grow existing blocks and not just combine new blocks (or connected components). Therefore, in OBGP only nodes directly adjacent to the current block, i.e., nodes in $\mathrm{adj}(B)$, will be considered. We call these nodes *candidate nodes*. Thus, in the basic version of OBGP we have $W_i \subset V_i \cup \mathrm{adj}(V_i)$. This restriction reduces the computational effort and is sufficient to guarantee that we do not add (part of) a new connected component.

Note that we could potentially reduce the number of connected components; i.e., it can happen that OBGP "connects" two or more of the connected components of $\mathcal{G}|_B$ by growing the block. If we use, e.g., XPABLO [14] to find the nonoverlapping partitioning, it is not uncommon for a block to have more than one connected component.

Algorithm 2 Outline of the OBGP(ℓ) algorithm.

1: **input:** a digraph $\mathcal{G}(A) = (V, E)$
         and a (nonoverlapping) partitioning $\{V_1, \ldots, V_q\}$
2: **output:** an (overlapping) decomposition $\{W_1, \ldots, W_q\}$ of V
3: **for** $i = 1, \ldots, q$ **do**   {go through each block}
4:         $B := V_i$
5:         **for** $k = 1, \ldots, \ell$ **do**   {ℓ rounds}
6:                determine $L := \mathrm{adj}(B)$ {set of candidate nodes}
7:                select nodes $N \subset L$ for inclusion
8:                $B := B \cup N$
9:         **end for**
10:        $W_i := B$
11: **end for**

To allow more overlap and more distant nodes to become candidate nodes, a block is usually extended several times. In OBGP(ℓ) we do ℓ *rounds* of extending the block. Algorithm 2 shows the outline of OBGP(ℓ). We denote the original block B by $B^{(0)}$ and the grown block after k rounds by $B^{(k)}$; i.e., $B^{(\ell)}$ is the grown block computed by OBGP(ℓ). In round k, $1 \leq k \leq \ell$, only nodes in $\mathrm{adj}(B^{(k-1)})$ are candidate nodes. Therefore in the first round ($k = 1$) only nodes in $\mathrm{adj}(B)$ are candidate nodes. However, the candidate nodes in round $k > 1$ need not to be in $\mathrm{adj}(B)$; rather they are from $L_k(B)$.

We mention here that including all the nodes in $\mathrm{adj}(B^{(k-1)})$, i.e., to consider $\ell = 1$, is what was proposed in [3], and it is also how the default overlap is computed in the domain decomposition programs of the software PETSc [1]. As we shall see, our proposed method makes a better selection of nodes. Not all nodes in $\mathrm{adj}(B^{(k-1)})$ are added to the block, but some other nodes are.

Adding all candidate nodes in round k, $1 \leq k \leq \ell$, is equivalent to adding all nodes in $\mathrm{adj}(B^{(k-1)})$ to the block. This can grow the blocks and hence the computational cost excessively. An example for such a behavior is the `RAEFSKY2` matrix from the University of Florida Sparse Matrix Collection [7]. Although the iteration count reduces substantially by adding all candidate nodes as overlap in each round (see Figure 4.1 for the convergence histories), the total solution time actually increases; i.e., the reduced iteration count does not compensate for the additional time needed in each iteration and in the setup of the preconditioner. In Table 4.1 we show the total solution times and iteration counts for solving `RAEFSKY2` using the different amounts of overlap.

In order to limit the number of nodes added to a block, in OBGP(ℓ) we introduce two user-supplied bounding functions μ and ν. In any round k, $k = 1, \ldots, \ell$, at most $\mu\bigl(|B^{(k-1)}|\bigr)$ of the candidate nodes can be selected for the extension, where $\mu : \mathbb{N} \to \mathbb{N}$ is a function chosen a priori.

The total growth bound $\nu(|B|) \geq 0$ limits the size of $B^{(k)}$ for all $k = 1, \ldots, \ell$; i.e., we require $|B^{(k)}| \leq |B^{(0)}| + \nu(|B^{(0)}|)$, $k = 1, \ldots, \ell$. Both μ and ν can be set to ∞ so as not to enforce the bound. In most applications we will have $\mu(|B^{(k)}|) < \bigl|\mathrm{adj}(B^{(k)})\bigr|$; i.e., in general we expect to have more candidate nodes than the number of nodes we are allowed to add.

OBGP now needs to decide which candidate nodes are actually added to the block. This selection process is at the heart of OBGP. It works as follows: Each candidate
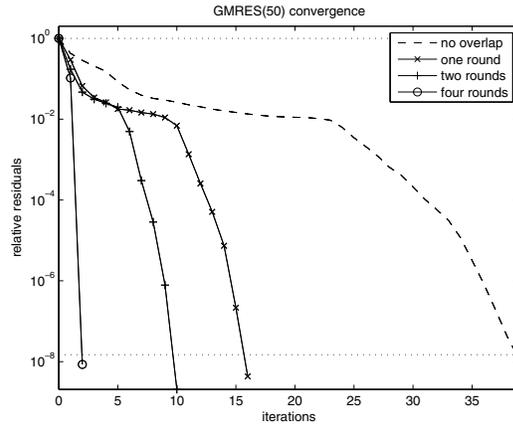
FIG. 4.1. *Adding overlap in multiplicative Schwarz preconditioning. The curves show the (full) GMRES convergence for solving a linear system (matrix* RAEFSKY2*) using the block Gauss–Seidel preconditioner (dashed curve) and several multiplicative Schwarz preconditioners (solid curves).*

TABLE 4.1
Results for adding all candidate nodes as overlap.

Rounds of adding overlap	Time	Iter
zero rounds (no overlap)	0.665	39
one round	0.846	16
two rounds	1.388	10
three rounds	2.230	2

node is given a node weight, and only nodes with largest weight are added to the block. In round k up to $\mu(|B^{(k-1)}|)$ nodes with largest weights from the candidate node set $\mathrm{adj}(B^{(k-1)})$ are used to grow the block. We add all candidate nodes if the number of candidate nodes $\mathrm{adj}(B^{(k-1)})$ is less than the limit $\mu(|B^{(k-1)}|)$ and less than the number of nodes that we can still add according to the total growth bound ν; i.e., we add all candidate nodes if

$$\left|\mathrm{adj}(B^{(k-1)})\right| \le \min\Big\{ \underbrace{\mu(|B^{(k-1)}|)}_{\text{number of nodes we can add in round } k}, \overbrace{\nu(|B^{(0)}|)}^{\text{total growth bound}} - \underbrace{\big(|B^{(k-1)}| - |B^{(0)}|\big)}_{\text{number of nodes added in first } k-1 \text{ rounds}} \Big\}.$$

The weight attributed to a node j should reflect the strength of its connectivity with the other nodes from the block B. Our choice is to basically just take the 1-norm of the new row and column in the matrix block $A_{B\cup\{j\}}$ that we would obtain by adding the node as our measure of the strength of connectivity. The precise definition is as follows.

DEFINITION 4.1. *Let* $\mathcal{G} = (V, E)$ *be a directed graph with edge weights* $w_E(e)$, $e \in E$. *Let* $B \subset V$ *be a set of nodes and* $j \in V$. *The weight* $w(j, B)$ *of* j *with respect to* B *is defined as*

$$w(j, B) := \sum_{e \in \mathrm{inc}(\{j\}, B)} |w_E(e)|.$$

As discussed before, the number of nodes to be added in round k is bounded by

$\mu = \mu(|B^{(k-1)}|)$. Typically we will set this bound to

$$(4.1) \qquad \mu\big(|B^{(k-1)}|\big) = \left\lceil \alpha \cdot \sqrt{|B^{(k-1)}|} \right\rceil ,$$

where α is a user-supplied positive constant. Typical values for α are $\alpha = 1$ or $\alpha = 2$, which were found in our experiments to give good results. This definition of μ is motivated by the size of the boundary in a graph stemming from a discretization of a two-dimensional partial differential equation over a square domain. We will later see that this bound μ also has nice consequences for the time complexity of OBGP; see Corollary 5.5.

Let $L^{(k)}$ be the set of all candidate nodes, and let $N^{(k)}$ be the set of candidate nodes selected for extending $B^{(k-1)}$ in round k, $k = 1, \ldots, \ell$, i.e., $N^{(k)} \subset L^{(k)} = L_1(B^{(k-1)})$ and $B^{(k)} = B^{(k-1)} \cup N^{(k)}$. The sets $N^{(k)}$ and $L^{(k)}$ can be described in terms of level sets by

$$N^{(1)} \subset L^{(1)} = L_1(B),$$
$$N^{(2)} \subset L^{(2)} = L_1(B \cup N^{(1)}),$$
$$\vdots$$
$$N^{(k)} \subset L^{(k)} = L_1\Big(B \cup \bigcup_{j=1}^{k-1} N^{(j)}\Big).$$

In terms of level sets with respect to B, the relations

$$N^{(1)} \subset L^{(1)} = L_1(B),$$
$$N^{(2)} \subset L^{(2)} \subset L_1(B) \cup L_2(B),$$
$$\vdots$$
$$N^{(k)} \subset L^{(k)} \subset \bigcup_{j=1}^{k} L_j(B)$$

hold. So we have the following result.

PROPOSITION 4.2. *A node added to a block in the kth round comes from one of the first k level sets with respect to the original block from the nonoverlapping partitioning.*

4.2. Implementation. In this section we discuss in detail all ingredients needed to implement OBGP(ℓ) in an efficient way. Let $V_i^{(0)} = V_i$ be the ith block of a (nonoverlapping) partitioning $\{V_1, \ldots, V_q\}$ of V. Let $V_i^{(k)}$ be the block after the kth round, and let W_i be the block after all ℓ rounds, i.e., $W_i = V_i^{(\ell)}$. Similar to the notation in the previous section, let $N_i^{(k)}$ and $L_i^{(k)}$ be the sets N and L computed in the kth round for block i. Since each block is considered separately, the index i will be omitted in many cases where the statements hold for all $i = 1, \ldots, q$.

In the case of multiple rounds, i.e., if $\ell > 1$, we can update the set L of candidate nodes going from one round to the next instead of determining it from scratch in each round. We will see in section 5 that it is in fact much more efficient to update L. In the kth round the new set of candidate nodes $L^{(k)}$ consists of the previous candidate

Algorithm 3 OBGP(ℓ).

1: **input:** number of rounds ℓ, a digraph $\mathcal{G}(A) = (V, E)$ with edge weights w_E
     and a (nonoverlapping) partitioning $\{V_1, \ldots, V_q\}$ of V
2: **output:** an (overlapping) decomposition $\{W_1, \ldots, W_q\}$ of V
3: set $w(j) := 0$, $j = 1, \ldots, n$
4: **for** $i = 1, \ldots, q$ **do**   {go through each block}
5:     $V_i^{(0)} := V_i$
6:     $\nu_0 := \nu(|V_i^{(0)}|)$
7:     $N^{(0)} := V_i$ {newly added nodes}
8:     $L := \emptyset$ {level set}
9:     **for** $k = 1, \ldots, \ell$ **do**
10:       **for all** $v \in N^{(k-1)}$ **do**
11:         **for all** $e \in E$ incident to v **do**
12:           **if** e connects v to some node $v' \notin V_i^{(k-1)}$ **then**
13:             $w(v') := w(v') + |w_E(e)|$
14:             **if** $v' \notin L$ **then**
15:               $L := L \cup \{v'\}$
16:             **end if**
17:           **end if**
18:         **end for**
19:       **end for**
20:       select nodes $N^{(k)} \subset L$ with largest weights
          $\{|N^{(k)}| = \min\{|L|, \mu(V_i^{(k-1)}), \nu_{k-1}\}\}$
21:       $L := L \setminus N^{(k)}$
22:       $V_i^{(k)} := V_i^{(k-1)} \cup N^{(k)}$
23:       $w(j) := 0$, $j \in N^{(k)}$
24:       $\nu_k := \nu_{k-1} - |N^{(k)}|$ {$\nu_k \geq 0$ since $|N^{(k)}| \leq \nu_{k-1}$}
25:     **end for**
26:     $w(j) := 0$, $j \in L$ {now we have $w \equiv 0$}
27:     $W_i := V_i^{(\ell)}$
28: **end for**

nodes from $L^{(k-1)}$ not added to the block in round $k-1$, i.e., not in $N^{(k-1)}$, and nodes adjacent to $N^{(k-1)}$, which are not in the previous block $V^{(k-1)}$, i.e.,

$$L^{(k)} = \left(L^{(k-1)} \setminus N^{(k-1)}\right) \cup \left(L_1(N^{(k-1)}) \setminus V^{(k-1)}\right).$$

Algorithm 3 describes OBGP(ℓ) using updates of the candidate set. This algorithm also shows how the node weights $w(v) = w(v, V_i^{(k-1)})$, $v \in L$, are computed and updated. Figure 4.2 shows how OBGP grows a given block.

For a time and space efficient implementation of OBGP(ℓ), the sets $V_i^{(k)}$, $k = 0, \ldots, \ell$, and the set N are not stored separately. For a fixed block number i, the invariants

$$V^{(k-1)} \subset V^{(k)}, \qquad V^{(k-1)} \cap N^{(k)} = \emptyset, \qquad \text{and} \qquad V^{(k-1)} \cup N^{(k)} = V^{(k)}$$

hold for $k = 1, \ldots, \ell$. We can therefore store the node numbers of the nodes in these sets in an integer vector of size n as shown in Figure 4.3. Furthermore, this integer
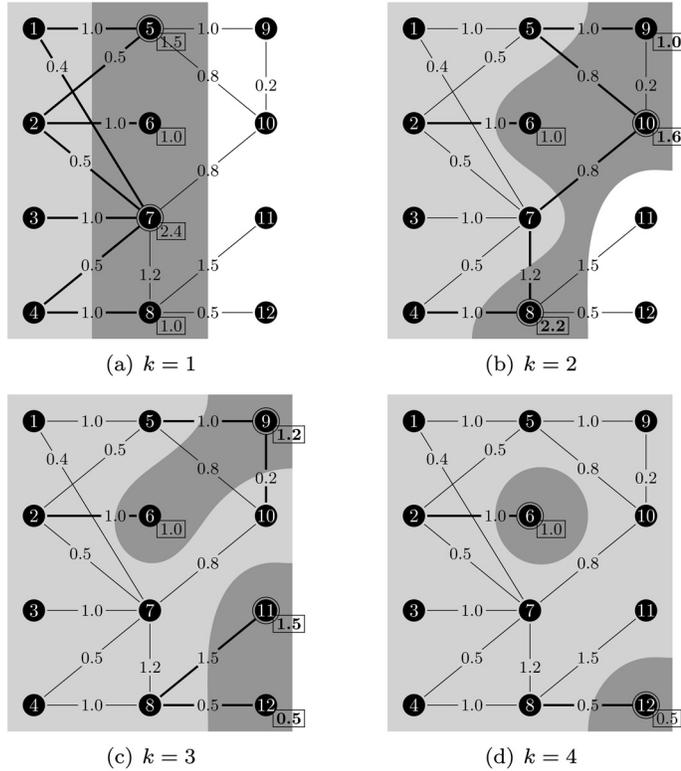
(a) $k = 1$

(b) $k = 2$

(c) $k = 3$

(d) $k = 4$

FIG. 4.2. *OBGP block growth showing edge weights. In each picture the current block $B^{(k-1)}$ is shown with a light gray background and the set L of candidate nodes with a slightly darker gray background. The index k counts the current round; cf. Algorithms 2 and 3. The edges used to compute the node weights are printed with thick lines. The computed node weights are printed in a box right next to the nodes. Node weights printed in bold have been changed from the previous round. The nodes selected for inclusion into the block are marked by an extra circle around the node.*

vector can be reused for the different blocks as each block is grown independently of the others.

An important choice is the data structure for storing the set L. Several operations are done with L:

- Adding a node to L.
- Selecting and removing some nodes with largest weights.
- Iterating over the nodes in L. Note that it is not necessary to traverse the nodes in a specific order.
- Changing the weight associated with a node in L.

There is no single "best" data structure allowing all listed operations with optimal time complexity. If we use, e.g., a linked list with an external index to store L, adding nodes, iterating over the nodes, and changing node weight all have time complexity $\mathcal{O}(1)$, but selecting the node with largest weight is $\mathcal{O}(|L|)$. Overall, we mainly want all operations to have a reasonable time complexity. The best compromise we know of is to use a (maximum) heap data structure. In section 5 and in our numerical results in section 6, we use a binary heap. We refer the reader, e.g., to the textbook [6] for a detailed description and analysis of binary heaps. If a binary heap is used,
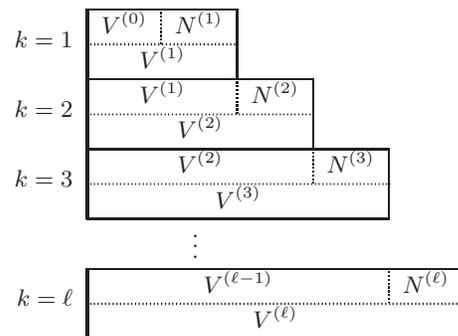
FIG. 4.3. *Storage of node sets $V^{(k)}$ and $N^{(k)}$. The figure shows how to store $V^{(k)}$ and $N^{(k)}$ in OBGP(ℓ) for a fixed block i. Only one integer vector of size n is needed.*

adding a node to L or removing the largest node has time complexity $\mathcal{O}(\log |L|)$. In the following, "heap" will always mean "binary heap," although the results hold also for other kinds of heaps like binomial heaps or Fibonacci heaps. We again refer the reader to [6] for details about binomial heaps and Fibonacci heaps.

Remark 4.3. It can be readily observed that in Algorithm 3 the main loop (lines 4–28), can be executed completely in parallel. Each block is grown without any information on the way the other blocks are grown. In other words the work on each block can be performed simultaneously on different processors. In terms of information for each processor, all that is needed are the ℓ level sets for the corresponding block. We mention, however, that, as the analysis in the next section shows, the complexity of OBGP is so low that a sequential implementation is very efficient.

5. Complexity analysis of OBGP. In this section we assume that a heap data structure is used for storing L. Let nnz(A) denote the number of nonzeros of the matrix A.

THEOREM 5.1. *Let L be stored in a heap. Then algorithm OBGP(ℓ) can be implemented in such a way that the time complexity is $\mathcal{O}(q \cdot (\text{nnz}(A) + n \log n))$.*

Proof. We will show that the time complexity for the computation of one particular set W_i is $\mathcal{O}(\text{nnz}(A) + n \log n)$; see lines 5–27 in Algorithm 3 for the necessary steps.

Updating w is done for edges incident to nodes in $N^{(k)}$, $k = 0, \ldots, \ell-1$. Note that $N^{(k)} \cap N^{(k')} = \emptyset$ for $k \neq k'$; i.e., for each node in V the incident edges are considered at most once and hence any edge in E is considered at most twice. Therefore, the time complexity for updating w (line 13) is $\mathcal{O}(\text{nnz}(A))$.

A node selected and removed from L (lines 20 and 21) is added to the block and can therefore not be added to L again; see the condition in line 12. Therefore, each node of V is added to L at most once and thus also selected and removed from L at most once. An upper bound for $|L|$ is n. This gives a time complexity of $\mathcal{O}(n \log n)$ for adding nodes to L and for selecting and removing nodes from L.

It is easy to see that the work outside the k-loop has time complexity $\mathcal{O}(n)$. Adding all this together, we have that the time complexity for the computation of one W_i block is $\mathcal{O}(\text{nnz}(A) + n \log n)$. For q blocks the total time complexity is then $\mathcal{O}(q \cdot (\text{nnz}(A) + n \log n))$.   □

Remark 5.2. If OBGP is implemented in parallel, as discussed in Remark 4.3, then the factor q can be dropped from the result of Theorem 5.1 and the parallel

complexity (using q processors) is $\mathcal{O}(\mathrm{nnz}(A) + n\log n)$.

For many practical problem cases a tighter time complexity than that of Theorem 5.1 can be shown. This will be stated in Theorem 5.6. To prepare the theorem, we need some auxiliary results for special choices of the bound function ν.

LEMMA 5.3. *Let $\mathcal{G} = (V, E)$ be a directed graph and let d be the maximum degree, i.e., $d = \max_{v \in V} \deg(v)$. If the total growth of block B is bounded by $|B| + \nu(|B|) \leq c_\nu \cdot |B|$, where c_ν is a constant independent of $|B|$, then*

  1. $|W_i| = \mathcal{O}(|V_i|)$ *and* $\sum_{i=1}^{q} |W_i| = \mathcal{O}(n)$;
  2. $|\mathrm{inc}(W_i)| \leq d|W_i| \leq dc_\nu|V_i|$ *and hence* $\sum_{i=1}^{q} |\mathrm{inc}(W_i)| = \mathcal{O}(dn)$.

Proof.

  1. Part 1 follows immediately from $\nu(|B|) \leq c_\nu \cdot |B|$ and $\sum_{i=1}^{q} \mathcal{O}(|V_i|) = \mathcal{O}(n)$.
  2. The set $\mathrm{inc}(W_i)$ contains the edges considered by OBGP while computing W_i. Since d is the maximum number of edges incident to a node, the number of edges in $\mathrm{inc}(W_i)$ is bounded by $|\mathrm{inc}(W_i)| \leq d|W_i|$. Thus,

$$\sum_{i=1}^{q} |\mathrm{inc}(W_i)| \leq d\sum_{i=1}^{q} |W_i| \leq dc_\nu \sum_{i=1}^{q} |V_i| = dc_\nu n = \mathcal{O}(dn). \quad \Box$$

We note that the bound $\mathcal{O}(dn)$ in part 2 of the lemma is also a bound on the number of edges in the graph, since d is the maximal degree. We will show that with our suggested bound function μ and a mild restriction on the number of rounds, ℓ, we automatically fulfill the assumptions of Lemma 5.3. For this it is necessary to know by how much a single block can grow in ℓ rounds. A simple induction over ℓ—omitted here for brevity—shows the following; see [13].

THEOREM 5.4. *Let $\{V_1, \ldots, V_q\}$ be a nonoverlapping partitioning of V and let $\{W_1, \ldots, W_q\}$ be the decomposition computed by OBGP(ℓ). Let the number of nodes to be added to $V_i^{(k)}$, $i = 1, \ldots, q$, $k = 0, \ldots, \ell$, be bounded by*

$$(5.1) \qquad \mu\big(|V_i^{(k)}|\big) = \alpha \cdot \sqrt{|V_i^{(k)}|}.$$

Then the size of W_i, $i = 1, \ldots, q$, is bounded by

$$(5.2) \qquad |W_i| \leq |V_i| + \ell \cdot \mu\big(|V_i|\big) + \frac{\ell(\ell-1)\alpha^2}{4}.$$

COROLLARY 5.5. *Let V_i be a node set of a partitioning $\{V_i\}_{i=1,\ldots,q}$ of V. If $\mu(|V_i|) = \alpha\sqrt{|V_i|}$ and $\ell \leq \sqrt{|V_i|}$, then the total growth of block B is bounded by $c_\nu|B|$, where c_ν is a constant independent of $|B|$; i.e., with this choice of μ and ℓ the assumptions for Lemma 5.3 are satisfied.*

Proof.

$$|W_i| \leq |V_i| + \ell \cdot \mu\big(|V_i|\big) + \frac{\ell(\ell-1)\alpha^2}{4} \leq |V_i| + \alpha|V_i| + \frac{\ell^2\alpha^2}{4}$$

$$\leq |V_i| + \alpha|V_i| + \frac{\alpha^2}{4}|V_i| = \underbrace{(1 + \alpha + \alpha^2/4)}_{=:\,c_\nu}|V_i|. \quad \Box$$

THEOREM 5.6. *Let L be stored in a heap. Let $d = \max_{v \in V} \deg(v)$ be the maximum degree and let $s = \max_{i=1}^{q} |V_i|$ be the maximum block size of the nonoverlapping partitioning. Let the total block growth be bounded by $|W_i| \leq c_\nu|V_i|$ for all $i = 1, \ldots, q$,*

where c_ν is a constant independent of the block size. Then the algorithm OBGP(ℓ) has time complexity $\mathcal{O}(dn + n\log s)$.

Proof. From Lemma 5.3 it follows that we consider $\mathcal{O}(dn)$ edges while growing all blocks. The time for updating the node weights w is linear in the number of edges considered; i.e., the total time for updating w is $\mathcal{O}(dn)$.

In total, at most $c_\nu n$ nodes are added and removed from L, since $\sum_{i=1}^{q}|W_i| \leq c_\nu n$. The maximum size of L is at most $\max_{i=1}^{q}|\operatorname{inc}(W_i)| \leq \max_{i=1}^{q} d|W_i| \leq dc_\nu s$. Thus, the total time for adding nodes to L and removing the largest nodes from L is bounded by

$$c_\nu n\mathcal{O}\big(\log(dc_\nu s)\big) = c_\nu n\mathcal{O}(\log d + \log c_\nu + \log s) = \mathcal{O}\big(n(\log d + \log s)\big).$$

So far, we have ignored the phenomenon that nodes in the heap may change their weight and hence may have to be moved to a different position within the heap. In Algorithm 3 we can see that the weight of a node can only grow; i.e., nodes only move upwards within the heap because of a changed weight. Over the time a particular node stays in the heap, it could move from the bottom of the heap up to the top. This is also the worst-case scenario as the node cannot move downwards again. Nodes newly added to a heap are first put at the bottom and then moved up until they reach their correct position. In the worst case, a newly added node has to be moved up through the whole heap because the node belongs at the top. Therefore, the time needed for moving a node up to the top due to weight changes is already included in our worst-case bound for the time for adding the node to the heap.

The total time complexity is then

$$\mathcal{O}(dn) + \mathcal{O}\big(n(\log d + \log s)\big) = \mathcal{O}(dn + n\log s). \qquad \square$$

Note that Corollary 5.5 shows that with our default choice of μ we attain the total block growth bound for Theorem 5.6 as long as $\ell \leq \sqrt{|V_i|}$ for all $i = 1, \ldots, q$; i.e., with the default μ and ℓ not too large we do not have to bound the total block growth explicitly by setting ν.

5.1. Dealing with nodes of high degree. Theorem 5.6 shows that the execution time of OBGP can be sensitive to the maximum degree d. If the original system has, e.g., dense or nearly dense rows or columns, then d is $\mathcal{O}(n)$ and the worst-case time complexity becomes $\mathcal{O}(n^2)$. If this ever becomes a problem in practice, the situation can be resolved by "removing" the nodes with high degree from the set of nodes considered for overlap; i.e., we let OBGP(ℓ) work on $\mathcal{G}|_{V\setminus V_H}$, where V_H is the set of nodes with high degree. As the nodes in V_H are especially well connected to nodes outside their block, it may be helpful to add them as another block, i.e., to use them as a kind of coarse-grid variables.

Since we did not observe severe problems with nodes of high degree in our test cases, we allow OBGP(ℓ) always work on the whole graph.

6. Numerical results. In this section we present numerical experiments using OBGP as a tool for setting up Schwarz preconditioners. We compare these Schwarz preconditioners with those obtained using the whole first level set as in PETSc following [3], with ILUTP preconditioners, and with direct solvers.

Our test programs are written in a combination of MATLAB, C, C++, and Fortran and were compiled using the GNU Compiler Collection (GCC version 4.3). Thus, while we used MATLAB as a frame for our programs, most of our codes were compiled rather than interpreted. Identical optimization flags were used in all cases.

TABLE 6.1
Summary information on the test matrices.

	Matrix	n	nnz	Condest
1	PARA-8	155 924	2 094 873	–
2	OHNE2	181 343	6 869 939	–
3	3D_51448_3D	51 448	537 038	–
4	IBM_MATRIX_2	51 448	537 038	–
5	MATRIX_9	103 430	1 205 518	–
6	MATRIX-NEW_3	125 329	893 984	–
7	LSQ_2d_200000	200 000	4 010 198	$2.08 \cdot 10^{+8}$
8	LSQ_2d_400000	400 000	8 144 136	–
9	MPS_2d_200000	200 000	1 186 000	$1.13 \cdot 10^{+9}$
10	MPS_2d_400000	400 000	2 380 500	$3.01 \cdot 10^{+9}$
11	LSQ_3d_100000	100 000	4 138 471	–
12	LSQ_3d_200000	200 000	8 691 582	–
13	MPS_3d_100000	100 000	830 863	–
14	MPS_3d_200000	200 000	1 729 316	–

In this manner, our numerical experiments allow us to also compare the execution times and not only the iteration counts, keeping in mind that the latter will always depend to a certain extent on the hardware and software environment used. All experiments reported here were run on a MacBook Pro with a 2.4 GHz Intel Core 2 Duo processor and 4 GB of memory running Linux (Ubuntu 10.04) and MATLAB version R2011a.

For our experiments we use a set of test problems. Table 6.1 shows some basic properties of the problems. For each matrix we show the dimension (n), the number of nonzeros (nnz), and the 1-norm condition estimate (Condest), where available. The given condition estimate is the result of MATLAB's `condest` function. If no condition estimate is given, the computation failed for some reason; usually the available memory was not sufficient.

The test matrices 1 to 6 are semiconductor device simulation problems from the University of Florida Sparse Matrix Collection [7]. More information on solving systems from semiconductor device simulation can be found, e.g., in [24]. The matrices 7 to 14 were kindly provided by Benjamin Seibold. They are two- and three-dimensional meshfree discretizations of Poisson's equation with mixed boundary conditions; see [26].

The choice of the right-hand sides b was as follows: If b is provided with the matrix in [7], we use it as the right-hand side for (1.1). In all other cases we use $b = A\underline{e}$, where $\underline{e} = (1, \dots, 1)^T$ is the vector of all ones. The initial vector is taken as the zero vector.

For the direct solve results we use the UMFPACK [8] sparse (direct) solve package via MATLAB's backslash operator in the expression `x = A \ b`. All results for iterative methods use GMRES(r), i.e., restarted GMRES with a restart after every r iterations. GMRES(r) was preconditioned with an incomplete LU factorization or with a Schwarz preconditioner based on an overlapping partition computed with OBGP. We discuss the value of r used in our experiments later in this section.

We note that for the size of the blocks in these experiments, the use of UMFPACK is state of the art for sequential direct solvers. More sophisticated direct solvers, especially for high performance computers, are available, such as those developed by the PARDISO solver project; see, e.g., [23].

For the incomplete LU preconditioning we use MATLAB's `ilu` function to compute an incomplete LU factorization with threshold τ and pivoting using an ILUTP approach; see [21], [22]. For comparison we tested the thresholds 10^{-2}, 10^{-3}, and 10^{-4}. For each problem we report the results for the threshold that gave the lowest total solution time.

Before we start, we preprocess the matrices with the subroutine MC64 from the HSL library so that the matrices are permuted and scaled in such a way that they are diagonal zero-free, and, in fact, the diagonal entries are one in absolute value and all the off-diagonal entries are less than or equal to one in absolute value; see [9], [10] for its description. This preprocessing in particular makes the comparison with ILU a fair one.

In our experiments with OBGP we use both Metis and XPABLO to produce the initial nonoverlapping partitioning. For XPABLO we use the default parameters except that we use $minbs = 500$ and $maxbs = 5000$; see [14] for a detailed description of XPABLO. From the Metis package we use the *PartGraphKway* approach and set the number of blocks to $\lceil n/5000 \rceil$; i.e., on average each block has a size of less than 5000; see [18], [19] for descriptions of Metis.

The stopping criterion in all iterative methods was

$$\|r_k\|/\|b\| < \epsilon,$$

where r_k is the preconditioned residual at the kth iteration, b is the preconditioned right-hand side, and ϵ is a prescribed tolerance. GMRES(r) is stopped if convergence is not reached after 1000 total iterations. We performed two rounds of numerical experiments, with different parameters for GMRES(r). For our first round, we set $r = 100$ and $\epsilon = 10^{-10}$. For our second round, we set $r = 50$ and $\epsilon = \sqrt{\epsilon_M}$, where ϵ_M is the machine precision, i.e., $\sqrt{\epsilon_M} \approx 10^{-8}$ since we use IEEE 754 double precision arithmetic.

In our experiments we tested the preconditioner without overlap, with $\ell = 1$ (corresponding to what it is used in PETSc), $\ell = 5$, $\ell = 10$, and $\ell = 20$ (and in some cases some other values of ℓ), and varying values of the parameter α which limits the number of nodes taken from each level set, as defined in (4.1). From these experiments, we found that a recommended set of parameters, which we can use as "default parameters," are $\ell = 10$ and $\alpha = 2$. With these parameters, the timings obtained were either the best or close to it.

In Table 6.2 we present the results for multiplicative Schwarz preconditioning, using these default parameters, and they are compared with the solution using a direct method and ILU preconditioning. All times are in seconds and measure the total solution time, including the setup phases. Best times among these three cases are marked in bold. Whenever the method ran out of memory we label it in the table as "*oom*." For the iterative approaches we give the number of total iterations (It). For the incomplete LU preconditioners we report the logarithm of the used threshold τ ($\log_{10} \tau$)—the one which gave the lowest execution time. For the Schwarz preconditioners we indicate the method selected: M, M5, M10, etc. (X, X5, X10, etc.) means that we use a partitioning computed by Metis (XPABLO) without overlap or with overlap computed by OBGP doing 5, 10, etc., rounds, respectively. Note that here the ILU factorization preconditioning is successful for all matrices in our test bed; cf. [24], where this is not always the case. In any case, OBGP-based Schwarz preconditionings appear to provide a very good alternative when ILU preconditioning is not as successful.

TABLE 6.2

Comparison of different solvers and preconditioners for our test problems. For multiplicative Schwarz with OBGP, the default parameters are used. Here $r = 100$, $\epsilon = 10^{-10}$.

		Direct	Incomplete LU			Multiplicative Schwarz		
	Matrix	Time	$\log_{10} \tau$	Time	It	Method	Time	It
1	PARA-8	**oom**	−3	51.96	76	M10	**26.04**	27
2	OHNE2	**oom**	−3	195.80	190	X10	**155.40**	106
3	3D_51448_3D	19.9	−2	7.36	72	X10	**3.92**	19
4	IBM_MATRIX_2	20.8	−2	7.01	68	X10	**4.21**	20
5	MATRIX_9	**oom**	−2	20.26	56	X10	**17.24**	50
6	MATRIX-NEW_3	25.1	−2	23.80	88	X10	**17.50**	53
7	LSQ_2D_200000	**13.1**	−3	63.87	74	M10	33.01	33
8	LSQ_2D_400000	**37.3**	−3	201.85	104	M10	101.53	45
9	MPS_2D_200000	**4.31**	−4	25.84	33	M10	19.09	35
10	MPS_2D_400000	**11.03**	−4	68.76	45	M10	75.11	48
11	LSQ_3D_100000	**oom**	−2	40.88	36	X10	**30.10**	19
12	LSQ_3D_200000	**oom**	−2	**109.57**	45	X10	**oom**	
13	MPS_3D_100000	32.9	−2	**8.56**	39	M10	10.29	18
14	MPS_3D_200000	**oom**	−2	28.85	46	M10	**27.78**	23

TABLE 6.3

Best times for each Schwarz preconditioner. Here $r = 50$, $\epsilon = \sqrt{\epsilon_M}$.

	Additive				Restricted additive				Multiplicative			
	Md	α	Time	It	Md	α	Time	It	Md	α	Time	It
1	M10	2	28.14	39	M10	3	23.7	22	M10	2	18.44	15
2	X10	3	387.1	375	X10	3	172.4	163	X10	3	82.97	48
3	X5	1	4.95	47	X2	.5	4.33	44	X10	.5	2.89	19
4	X4	1	5.78	56	X5	4	4.95	28	X3	2	3.15	21
5	M2	1	33.28	92	M5	1	26.43	60	X15	1	14.35	47
6	X		24.24	126	X		25.7	126	X20	1	15.5	40
7	M3	∞	46.6	70	M3	∞	38.0	48	M3	∞	24.8	26
8	M20	1	194.6	117	M2	∞	157.9	80	M20	1	89.6	40
9	M20	4	27.5	41	M20	4	20.1	27	M10	4	13.3	21
10	M20	5	95.5	53	M20	5	73.3	36	M20	5	45.4	21
11	M		24.7	39	M1	.5	25.2	38	X		19.8	27
12	X		75.5	57	X		80.0	57	X		55.0	32
13	M10	.5	12.9	43	M1	∞	10.8	27	M3	1	8.3	23
14	M3	.5	35.2	63	M1	∞	32.6	32	M4	.5	21.9	29

Observe that for the two-dimensional meshfree discretizations (problems 8–11) a direct solver is superior to both ILU and multiplicative Schwarz. For the three-dimensional discretizations (problems 12–15) the situation is fundamentally different; indeed, the direct solver gave poor results and was not able to solve several problems because the available memory was not sufficient. The results in Table 6.2 are in this sense representative of the behavior of the preconditioner with other values of α and ℓ. As we show in Table 6.3, for these problems the best results were attained using Metis- or XPABLO-based multiplicative Schwarz preconditioners with only little or no overlap.

In Table 6.3 we present the best possible times (for the different values of α and ℓ that we tried) for the three Schwarz preconditioners. ("Md" stands for method). It can be appreciated that, as is well known in the domain decomposition community, RAS is faster than additive Schwarz, and multiplicative Schwarz is the fastest. Note
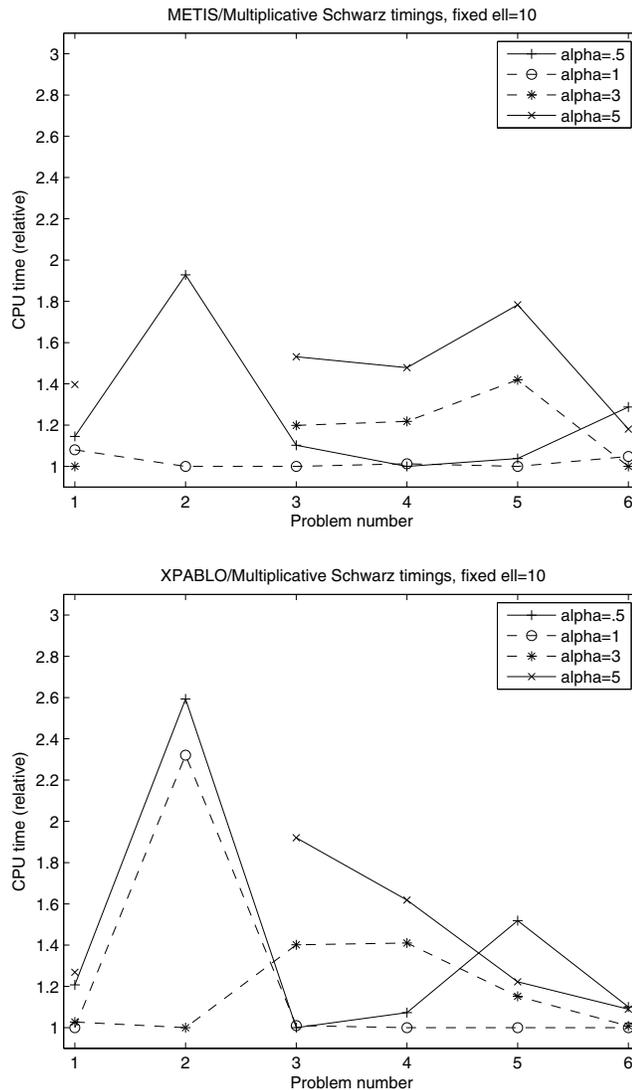
FIG. 6.1. *Graphical comparison of different solvers for our test problems* 1–6 *for* $\ell = 10$ *and varying* α. *The times are scaled such that the fastest method for a particular test problem has a relative time of* 1. *Ordering obtained by Metis (top) or XPABLO (bottom).*

that only in one case $\alpha = \infty$ (the PETSc default value as proposed in [3]) provides the best time among the experiments tried. We note that in our experiments, the best times are close to those obtained with the default values.

The robustness of the OBGP approach to produce the multiplicative Schwarz preconditioning, when varying the parameters α and ℓ, can be observed in Figures 6.1 and 6.2. We show complete running times for the matrices taken from the Matrix Market (matrices 1–6 in Table 6.1). For each problem the timings were scaled so that the fastest solution method has time 1; i.e., a relative time of two would indicate that the method was slower than the fastest method for this particular problem by a factor
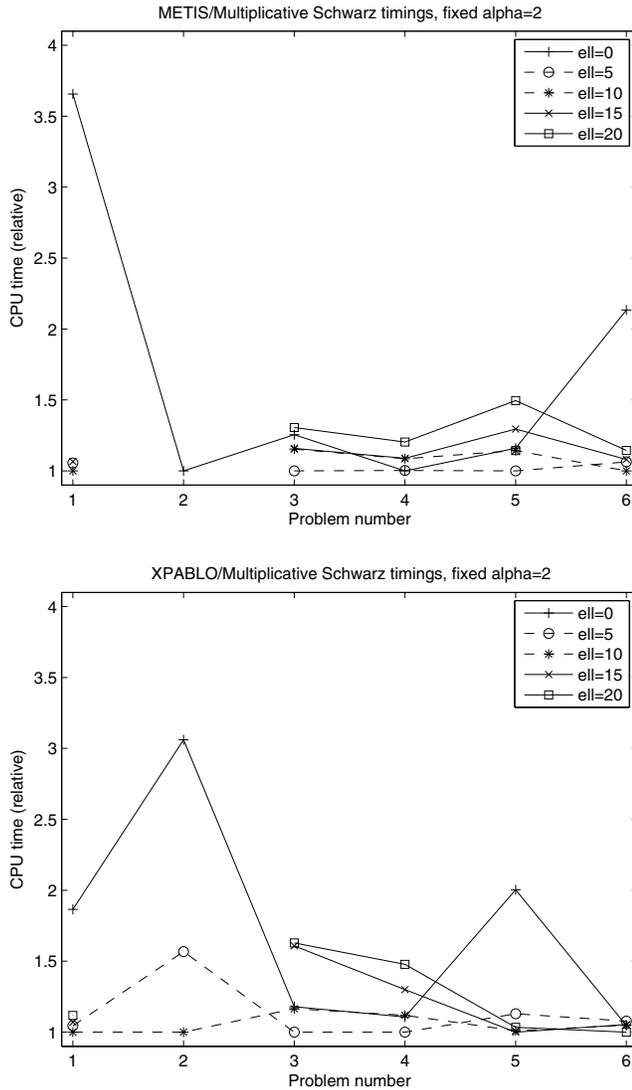
FIG. 6.2. *Graphical comparison of different solvers for our test problems* 1–6 *for* $\alpha = 2$ *and varying* ℓ. *The times are scaled such that the fastest method for a particular test problem has a relative time of* 1. *Ordering obtained by Metis (top) or XPABLO (bottom).*

of two. No marker and no connecting lines are printed if the method did not yield a result, i.e., if the method ran out of memory.

Since, heuristically, we expect the multiplicative Schwarz preconditioner to represent a better approximation of the inverse of A if we add overlap or increase the overlap, it is not surprising that we can in general see a decrease in the iteration count. The lower iteration count achieved by adding overlap is counterbalanced by a higher computational cost needed to set up the preconditioner and to apply it in each iteration step. The increased setup cost and increased cost per iteration can be observed for several test problems, e.g., for the `LSQ_3d_`$*$ and the `MPS_3d_`$*$ matrices.

The numerical results show that adding overlap can substantially improve the performance of block-based iterative solvers. For the semiconductor device simulation problems the iteration count improves tremendously and, what is important for practical applications, also the total time needed to solve the linear system improves by adding overlap.

In general we can conclude that the amount of overlap needed to minimize the total solution time is problem specific, though our default parameters provide a good starting place for any exploration.

7. Conclusions. We presented a linear time algorithm to obtain a set of overlapping blocks suitable for multiplicative Schwarz preconditioning which can be combined with existing high-quality graph partitioners. The idea is to grow a given partitioning of the underlying graph by incorporating certain adjacent nodes in the underlying graph. Nodes are selected by a criterion based on the size of the respective matrix entries. We implemented our method carefully so that the measured timings should give a clear indication of its performance. We performed a series of numerical experiments for problems coming from semiconductor device simulation and mesh-free discretizations of Poisson's equation. These experiments show that a substantial amount of overlap usually produces the best method, in most cases superior to ILU preconditioning and/or direct solvers, as well as to the approach from [3], which is the default in the PETSc software [1].

REFERENCES

[1] S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith, *PETSc* 2.0 *User's Manual*, Technical Report ANL-95/11, Revision 2.0.22, Argonne National Laboratory, 1998; available online from http://www.mcs.anl.gov/petsc.

[2] M. Benzi, A. Frommer, R. Nabben, and D. B. Szyld, *Algebraic theory of multiplicative Schwarz methods*, Numer. Math., 89 (2001), pp. 605–639.

[3] X.-C. Cai and Y. Saad, *Overlapping domain decomposition algorithms for general sparse matrices*, Numer. Linear Algebra Appl., 3 (1996), pp. 221–237.

[4] X.-C. Cai and M. Sarkis, *A restricted additive Schwarz preconditioner for general sparse linear systems*, SIAM J. Sci. Comput., 21 (1999) pp. 792–797.

[5] H. Choi and D. B. Szyld, *Application of threshold partitioning of sparse matrices to Markov chains*, in IEEE International Computer Performance and Dependability Symposium IPDS'96, Urbana-Champaign, IL, IEEE Computer Society Press, Los Alamitos, CA, 1996, pp. 158–165.

[6] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd ed., MIT Press, Cambridge, MA, 2009.

[7] T. A. Davis, *University of Florida Sparse Matrix Collection*, available online from http://www.cise.ufl.edu/research/sparse/matrices/.

[8] T. A. Davis, *Algorithm 8xx: UMFPACK: An Unsymmetric-Pattern Multifrontal Method*, ACM Trans. Math. Software, 30 (2004); available online from http://www.cise.ufl.edu/research/sparse/umfpack/.

[9] I. S. Duff and J. Koster, *The design and use of algorithms for permuting large entries to the diagonal of sparse matrices*, SIAM J. Matrix Anal. Appl., 20 (1999), pp. 889–901.

[10] I. S. Duff and J. Koster, *On algorithms for permuting large entries to the diagonal of a sparse matrix*, SIAM J. Matrix Anal. Appl., 22 (2001), pp. 973–996.

[11] I. Duff, S. Gratton, X. Pinel, and X. Vasseur, *Multigrid based preconditioners for the numerical solution of two-dimensional heterogeneous problems in geophysics*, Int. J. Comput. Math., 84 (2007), pp. 1167–1181.

[12] S. C. Eisenstat, *Efficient implementation of a class of preconditioned conjugate gradient methods*, SIAM J. Sci. Statist. Comput., 2 (1981), pp. 1–4.

[13] D. Fritzsche, *Overlapping and Nonoverlapping Orderings for Preconditioning*, Ph.D. thesis, Department of Mathematics, Temple University, Philadelphia, 2010.

[14] D. Fritzsche, A. Frommer, and D. B. Szyld, *Extensions of certain graph-based algorithms for preconditioning*, SIAM J. Sci. Comput., 29 (2007), pp. 2144–2161.

[15] A. Frommer and D. B. Szyld, *Weighted max norms, splittings, and overlapping additive Schwarz iterations*, Numer. Math., 83 (1999), pp. 259–278.

[16] A. Frommer and D. B. Szyld, *An algebraic convergence theory for restricted additive Schwarz methods using weighted max norms*, SIAM J. Numer. Anal., 39 (2001), pp. 463–479.

[17] W. Hackbusch, *Iterative Solution of Large Sparse Systems of Equations*, Springer, New York, Berlin, Heidelberg, 1994.

[18] G. Karypis and V. Kumar, *MeTis: Unstructured Graph Partitioning and Sparse Matrix Ordering System, Version* 4.0, user manual, Department of Computer Science, University of Minnesota, 1998; available online from http://www.cs.umn.edu/˜karypis/metis/metis/files/manual.ps.

[19] G. Karypis and V. Kumar, *A fast and high quality multilevel scheme for partitioning irregular graphs*, SIAM J. Sci. Comput., 20 (1998), pp. 359–392.

[20] J. O'Neil and D. B. Szyld, *A block ordering method for sparse matrices*, SIAM J. Sci. Statist. Comput., 11 (1990), pp. 811–823.

[21] Y. Saad, *ILUT: A dual threshold incomplete LU factorization*, Numer. Linear Algebra Appl., 1 (1994), pp. 387–402.

[22] Y. Saad, *Iterative Methods for Sparse Linear Systems*, 2nd ed., SIAM, Philadelphia, PA, 2003.

[23] O. Schenk and K. Gärtner, *Solving unsymmetric sparse systems of linear equations with PARDISO*, J. Future Generation Comput. Syst., 20 (2004), pp. 475–487.

[24] O. Schenk, S. Röllin, and A. Gupta, *The effects of unsymmetric matrix permutations and scalings in semiconductor device and circuit simulation*, IEEE Trans. CAD Integrated Circuits Syst., 23 (2004), pp. 400–411.

[25] H. A. Schwarz, *Über einen Grenzübergang durch alternierendes Verfahren*, Vierteljahrsschrift der Naturforschenden Gesellschaft in Zürich, 15 (1870), pp. 272–286.

[26] B. Seibold, *Minimal positive stencils in meshfree finite difference methods for the Poisson equation*, Comput. Methods Appl. Mech. Engrg., 198 (2008), pp. 592–601.

[27] B. F. Smith, P. E. Bjørstad, and W. Gropp, *Domain Decomposition: Parallel Multilevel Methods for Elliptic Partial Differential Equations*, Cambridge University Press, New York, 1996.

[28] A. Toselli and O. Widlund, *Domain Decomposition Methods: Algorithms and Theory*, Springer Ser. Comput. Math. 34, Springer, Berlin, Heidelberg, 2005.