JOURNAL OF COMPUTER AND SYSTEM SCIENCES **54**, 332–344 (1997)
ARTICLE NO. SS971473

Efficient Out-of-Core Algorithms for Linear Relaxation Using Blocking Covers

Charles E. Leiserson*

MIT Lab for Computer Science, 545 Technology Square, Cambridge, Massachusetts 02139

Satish Rao

NEC Research Institute, 4 Independence Way, Princeton, New Jersey 08540

and

Sivan Toledo*

MIT Lab for Computer Science, 545 Technology Square, Cambridge, Massachusetts 02139

Received January 20, 1994

When a numerical computation fails to fit in the primary memory of a serial or parallel computer, a so-called "out-of-core" algorithm, which moves data between primary and secondary memories, must be used. In this paper, we study out-of-core algorithms for sparse linear relaxation problems in which each iteration of the algorithm updates the state of every vertex in a graph with a linear combination of the states of its neighbors. We give a general method that can save substantially on the I/O traffic for many problems. For example, our technique allows a computer with M words of primary memory to perform $T = \Omega(M^{1/5})$ cycles of a multigrid algorithm for a two-dimensional elliptic solver over an n-point domain using only $\Theta(nT/M^{1/5})$ I/O transfers, as compared with the naive algorithm which requires $\Omega(nT)$ I/O's. Our method depends on the existence of a "blocking" cover of the graph that underlies the linear relaxation. A blocking cover has the property that the subgraphs forming the cover have large diameters once a small number of vertices have been removed. The key idea in our method is to introduce a variable for each removed vertex for each time step of the algorithm. We maintain linear dependences among the removed vertices, thereby allowing each subgraph to be iteratively relaxed without external communication. We give a general theorem relating blocking covers to I/O-efficient relaxation schemes. We also give an automatic method for finding blocking covers for certain classes of graphs, including planar graphs and d-dimensional simplicial graphs with constant aspect ratio (i.e., graphs that arise from dividing d-space into "well-shaped" polyhedra). As a result, we can perform T iterations of linear relaxation on any n-vertex planar graph using only $\Theta(n + nT\sqrt{\lg n}/M^{1/4})$ I/O's or on any n-node d-dimensional simplicial graph with constant aspect ratio using only $\Theta(n + nT \lg n/M^{\Omega(1/d)})$ I/O's.

* This research was supported in part by the Advanced Research Projects Agency under Grant N00014-91-J-1698.

1. INTRODUCTION

Many numerical problems can be solved by linear relaxation. A typical linear relaxation computation operates on a directed graph $G = (V, E)$ in which each vertex $v \in V$ contains a numerical state variable x_v which is iteratively updated. On step t of a linear relaxation computation, each state variable is update by a weighted linear combination of its neighbors:

$$x_v^{(t)} = \sum_{(u,v) \in E} A_{uv}^{(t)} x_u^{(t-1)}, \tag{1}$$

where $A_{uv}^{(t)}$ is a predetermined *relaxation weight* of the edge (u, v). We can view each iteration as a matrix–vector multiplication $x^{(t)} = A^{(t)} x^{(t-1)}$, where $x^{(t)} = \langle x_1^{(t)}, x_2^{(t)}, \ldots, x_{|V|}^{(t)} \rangle^T$ is the state vector for the tth step, and $A^{(t)} = (A_{uv}^{(t)})$ is the *relaxation matrix* for the tth step. We assume $A_{uv}^{(t)} = 0$ if $(u, v) \notin E$. The goal of the linear relaxation is to compute a final state vector $x^{(T)}$ given an initial vector $x^{(0)}$, a scheme for computing $A_{uv}^{(t)}$ on each step t, and a total number T of steps. Examples of linear relaxation computations include Jacobi relaxation, Gauss–Seidel relaxation, multigrid computations, and many variants of these methods [3]. (Iterative processes of the form $y^{(t)} = M^{(t)} y^{(t-1)} + b$ can be transformed to an iteration of the form $x^{(t)} = A^{(t)} x^{(t-1)}$ using a straightforward linear transformation.)

A computer with ample primary memory can perform a linear relaxation computation by simply updating the state

0022-0000/97 $25.00

vector according to Eq. (1). Since we are normally interested only in the final state vector $x^{(T)}$, we can reuse the state-vector storage. If we assume that the scheme for generating the nonzero entries of the relaxation matrices $A^{(t)}$ is not a significant cost (for example, all relaxation matrices may be identical), then the time to perform each iteration on an ordinary, serial computer is $O(E)$.[1] (It can be less, since if a row v of the relaxation matrix at some step t is 0 everywhere except for a 1 on the diagonal, then no computation is required to compute $x_v^{(t)}$.) Besides the space required for the relaxation weights, the total amount of storage required for the entire T-step computation is $\Theta(V)$ to store the state vector.

If the computation does not fit within the primary memory of the computer, however, a so-called "out-of-core" method must be used. An out-of-core algorithm tries to be computationally efficient, but in addition it attempts to move as little data as possible between the computer's primary memory and secondary memories, since the I/O bandwidth between the two memories is severely limited in most computer systems. The naive method repeatedly applies Eq. (1) to all vertices, computing the entire state vector for one time step before proceeding to the next. This strategy causes $\Omega(TV)$ words to be transferred (I/O's) for a T-step computation, if $|V| \geqslant 2M$, where M is the size of primary memory.

For some graphs, however, there are more clever strategies that use many fewer I/O's. Such strategies have been used at least since the 1960s; the earliest reference we have found is due to Pfeifer [11]. Hong and Kung [5] analyze a method for a T-step linear relaxation algorithm on a $\sqrt{n}$-by-$\sqrt{n}$ mesh that uses only $\Theta(Tn/\sqrt{M})$ I/O's, where the primary memory has size M. The idea is illustrated in Fig. 1. We load into primary memory the initial state of a k-by-k submesh S, where $k \leqslant \sqrt{M}$ is a value to be determined. With this information in primary memory, we can compute the state after one step of relaxation for all vertices in S except those on S's boundary ∂S. We can then compute the state after two steps of relaxation for vertices in $S - \partial S$, except for vertices in $\partial(S - \partial S)$. After τ steps, we have a $(k-2\tau)$-by-$(k-2\tau)$ submesh S' at the center of S such that every vertex $i \in S'$ has state $x_i^{(\tau)}$. We then write the state of S' out to secondary memory. By tiling the $\sqrt{n}$-by-$\sqrt{n}$ mesh with $(k-2\tau)$-by-$(k-2\tau)$ submeshes, we can compute τ steps of linear relaxation in time $\Theta(k^2\tau \cdot n/(k-2\tau)^2)$, since there are $n/(k-2\tau)^2$ submeshes in the tiling, each requiring $\Theta(k^2\tau)$ work. By choosing $k = \sqrt{M}$ and $\tau = \sqrt{M}/4$, the total time required for τ steps is $\Theta(4n\tau) = \Theta(n\tau)$. The number of I/O's for τ steps is $\Theta(k^2 \cdot n/(k-2\tau)^2) = \Theta(4n) = \Theta(n)$. By repeating this strategy, we can compute T steps with proportional efficiency, saving a factor of $\Theta(\sqrt{M})$ I/O's over the naive method and using only a small constant factor more work, which results from redundant calculations. Hong and Kung extend this result to save a factor of $\Theta(M^{1/d}/d)$ I/O's for d-dimensional meshes.

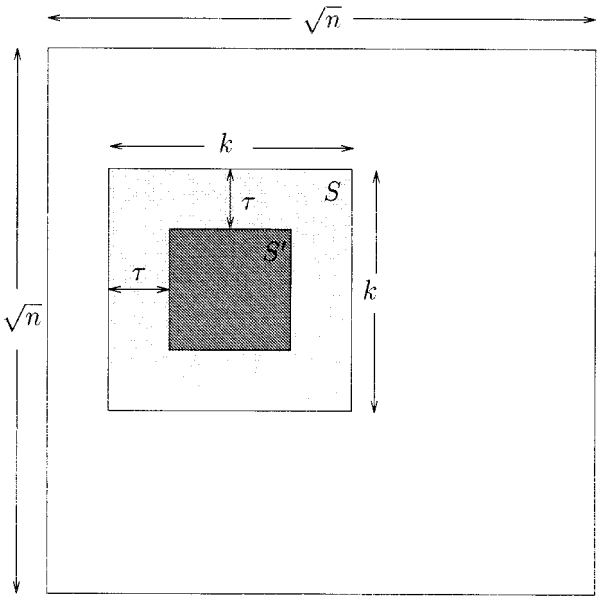

FIG. 1. Performing an out-of-core linear relaxation computation on a $\sqrt{n}$-by-$\sqrt{n}$ mesh. The k-by-k mesh S is loaded into primary memory, τ relaxation steps are performed, and the smaller mesh S' is stored to secondary memory.

Linear relaxation on a mesh naturally arises from the problem of solving sparse linear systems of equations arising from the discretization of partial differential equations on a mesh. For this class of problems, however, it has been found that more rapid convergence can often be obtained by performing a linear relaxation computation on a multigrid graph [3]. A multigrid graph is a hierarchy of progressively coarser meshes, as is shown in Fig. 2. The kth level is a $\sqrt{n/2^k}$-by-$\sqrt{n/2^k}$ mesh, for $k = 0, 1, ..., (\lg n)/2$, whose (i, j) vertex is connected to the $(2i, 2j)$ vertex on the $(k-1)$th mesh.

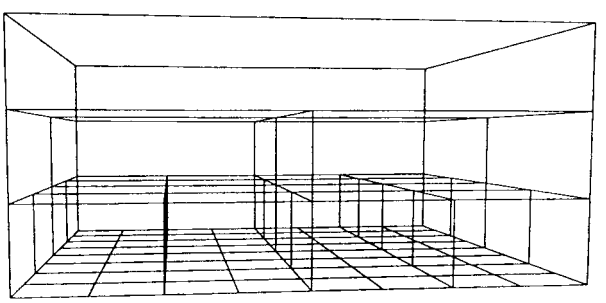

FIG. 2. A 9-by-9 multigrid graph. The graph has levels 0, the bottommost, through 3, the topmost.

[1] Inside asymptotic notation (such as O-notation or Θ-notation), the symbol V denotes $|V|$ and the symbol E denotes $|E|$.

A typical multigrid application is that of solving a parabolic partial differential equation. The computation consists of many repeated *cycles*, in which the relaxation proceeds level by level from the finest mesh to the coarsest and back down. A naive implementation of $T \geqslant \lg n$ cycles of the computation takes $\Theta(Tn)$ time, even though there are $\Theta(\lg n)$ levels, since the number of vertices on each level decreases geometrically as the grids become coarser. For a computer with M word of memory running T cycles of a $\sqrt{n}$-by-$\sqrt{n}$ multigrid algorithm, where $n \geqslant 2M$, the number of I/O's required for T cycles is $\Theta(Tn)$ as well.

Can the number of I/O's be reduced for this multigrid computation? We shall show in Section 5 that in the "red–blue pebble game" model for I/O proposed by Hong and Kung [5], the answer is no, even if redundant computations are allowed. The naive algorithm is optimal. The problem is essentially that information propagates quickly in the multigrid graph because of its small diameter.

Nevertheless, we shall see in Section 5 that we can actually save a factor of $M^{1/5}$ in I/O's. The key idea is to artificially restrict information from passing through some vertices by treating their state variables symbolically. Because the relaxations are linear, we can maintain dependences among the symbolic variables efficiently as a matrix. This technique is general and is particularly suited to graphs whose connections are locally dense and globally sparse.

The remainder of this paper is organized as follows. In Section 2 we formally introduce the notion of blocking covers and discuss the relation between state variables in a linear relaxation computation and a blocking cover. In Section 3 we present our method. The details of the method are presented in Section 4. The application of our basic result to multigrid relaxation is presented in Section 5. In Section 6 we describe algorithms for finding good blocking covers for planar and simplicial graphs, which yield I/O-efficient relaxation algorithms for these classes of graphs. Section 7 contains a discussion of the practical aspects of our method. We conclude the paper in Section 8 with a brief discussion of our results and of related out-of-core algorithms that we have developed.

2. BLOCKING COVERS

This section introduces the definition of a blocking cover, as well as several other definitions and notations that we shall use extensively in subsequent sections. We conclude the section with an important identity describing how state variables depend on one another.

We can abstract the method of Hong and Kung described in Section 1 using the notion of graph covers. Given a directed graph $G = (V, E)$, a vertex $v \in V$, and a constant $\tau \geqslant 0$, we first define $N^{(\tau)}(v)$ to be the set of vertices in V such that $u \in N^{(\tau)}(v)$ implies there is a path of length at most τ from u to v. A *τ-neighborhood-cover* [2] of G is a sequence of subgraphs $\mathscr{G} = \langle G_1 = (V_1, E_1), ..., G_k = (V_k, E_k) \rangle$, such that for all $v \in V$ there exists a $G_i \in \mathscr{G}$ for which $N^{(\tau)}(v) \subseteq V_i$. Hong and Kung's method can reduce the I/O requirements by a factor of τ over the naive method if the graph has a τ-neighborhood-cover with $O(E/M)$ subgraphs, each of which has $O(M)$ edges, where M is the size of primary memory. Although a vertex can belong to more than one subgraph in the cover, there is one subgraph that it considers to be its "home," in the sense that the subgraph contains all of its neighbors within distance τ. When performing a linear relaxation on G for τ time steps, therefore, the state of v depends only on other vertices in v's home. Thus, in a linear relaxation computation, we can successively bring each subgraph in the cover into primary memory and relax it for τ steps without worrying about the influence of any other subgraph for those τ steps.

The problem with Hong and Kung's method is that certain graphs, such as multigrid graphs and other low-diameter graphs, cannot be covered efficiently with small, high-diameter subgraphs. Our strategy to handle such a graph is to "remove" certain vertices so that the remaining graph has a good cover. Specifically, we select a subset $B \subseteq V$ of vertices to form a *blocking set*. We call the vertices in the blocking set *blocking vertices* or *blockers*. We define the τ-neighborhood of v with respect to a blocking set $B \subseteq V$ to be $N_B^{(\tau)}(v) = \{u \in V: \exists \text{ a path } u \to u_1 \to \cdots \to u_t \to v, \text{ where } u_i \in V - B \text{ for } i = 1, 2, ..., t < \tau\}$. Thus, the τ-neighborhood of v with respect to B consists of vertices that can be reached with paths of length at most τ whose internal vertices do not belong to B.

We can now define the notion of a blocking cover of a graph.

DEFINITION. Let $G = (V, E)$ be a directed graph. A *(τ, r, M)-blocking-cover* of G is a pair $(\mathscr{G}, \mathscr{B})$, where $\mathscr{G} = \langle G_1 = (V_1, E_1), ..., G_k = (V_k, E_k) \rangle$ is a sequence of subgraphs of G and $\mathscr{B} = \langle B_1, ..., B_k \rangle$ is a sequence of subsets of V such that

BC1. for all $i = 1, ..., k$, we have $M/2 \leqslant |E_i| \leqslant M$;

BC2. for all $i = 1, ..., k$, we have $|B_i| \leqslant r$;

BC3. $\sum_{i=1}^{k} |E_i| = O(E)$;

BC4. for all $v \in V$, there exists a $G_i \in \mathscr{G}$ such that $N_{B_i}^{(\tau)}(v) \subseteq V_i$.

For each $v \in V$, we define home(v) to be an arbitrary one of the G_i that satisfies BC4.

Our basic algorithm for linear relaxation on a graph $G = (V, E)$ depends on having a (τ, r, M)-blocking-cover of G such that $r^2\tau^2 \leqslant M$. In the description and analysis of the basic algorithm, we shall assume for simplicity that each

step of the computation uses the same relaxation matrix A. We shall call such a computation a *simple linear relaxation computation.* We shall relax this simplifying assumption in Section 5.

In a simple linear relaxation computation on a graph $G = (V, E)$ with a relaxation matrix A, the state vector $x^{(t)}$ at time t satisfies

$$x^{(t)} = A^t x^{(0)}.$$

That is, the computation amounts to powering the matrix.[2] We shall generally be interested in the effect that one state variable $x_u^{(s)}$ has on another $x_v^{(t)}$. Define the weight $w(p)$ of a length-r path $p = v_0 \to v_1 \to \cdots \to v_r$ in G to be

$$w(p) = \prod_{k=1}^{r} A_{v_{k-1}, v_k}. \tag{2}$$

For two vertices $u, v \in V$, we define

$$w(r; u, v) = \sum_{p \in \mathscr{P}(r)} w(p),$$

where $\mathscr{P}(r) = \{ p \in G: p \text{ is a length-}r \text{ path from } u \text{ to } v \}$. (We define $w(r; u, v) = 0$ if no length-r path exists between u and v.) Using this notation, we have

$$x_v^{(t)} = \sum_{u \in V} w(t; u, v)\, x_u^{(0)}. \tag{3}$$

If $B_i \subseteq V$ is a blocking set, then we define

$$w_{B_i}(r; u, v) = \sum_{p \in \mathscr{P}_{B_i}(r)} w(p),$$

where $\mathscr{P}_{B_i}(r) = \{ p \in G: p \text{ is a length-}r \text{ path from } u \text{ to } v \text{ whose intermediate vertices belong to } V - B_i \}$.

Consider a subgraph G_i in the cover and its corresponding blocking set B_i. The following lemma shows that in order to know the value of a state variable $x_v^{(t)}$ where $\text{home}(v) = G_i$, it suffices to know the intial values of all state variables for vertices in G_i at time 0 and also to know the values of the *blocker variables*—state variables for blockers—in B_i at all times less than t.

LEMMA 1. *Let $G = (V, E)$ be a directed graph, and let $(\mathscr{G}, \mathscr{B})$ be a (τ, r, M)-blocking-cover of G. Then, for any simple linear relaxation computation on G, we have for all $v \in V$ and for any $t \leqslant \tau$,*

$$x_v^{(t)} = \sum_{u \in V_i} w_{B_i}(t; u, v)\, x_u^{(0)} + \sum_{u \in B_i} \sum_{s=1}^{t-1} w_{B_i}(s; u, v)\, x_u^{(t-s)}, \tag{4}$$

where $G_i = \text{home}(v)$.

Proof. We have $\mathscr{P}(r) = \mathscr{P}_{B_i}(r) + \mathscr{P}_{\bar{B}_i}(r)$, where $\mathscr{P}_{\bar{B}_i}(r) = \{ p \in G: p \text{ is a length-}r \text{ path from } u \text{ to } v \text{ with at least one intermediate vertex which belongs to } B_i \}$. We also define

$$w_{\bar{B}_i}(r; u, v) = \sum_{p \in \mathscr{P}_{\bar{B}_i}(r)} w(p),$$

By Eq. (3) and the notation above we have

$$x_v^{(t)} = \sum_{u \in V} w(t; u, v)\, x_u^{(0)} = \sum_{u \in V} w_{B_i}(t; u, v)\, x_u^{(0)} + \sum_{u \in V} w_{\bar{B}_i}(t; u, v)\, x_u^{(0)}. \tag{5}$$

We now prove that the first sum in Eq. (5) equals the first sum in Eq. (4) and that the second sum in Eq. (5) equals the second sum in Eq. (4). That the first summations are equal follows from condition BC4 in the definition of blocking covers, which imply that if $\text{home}(v) = G_i$, $u \notin V_i$, and $t \leqslant \tau$, then $w_{B_i}(t; u, v) = 0$.

We use induction on t to prove that

$$\sum_{u \in B_i} \sum_{s=1}^{t-1} w_{B_i}(s; u, v)\, x_u^{(t-s)} = \sum_{z \in V} w_{\bar{B}_i}(t; z, v)\, x_z^{(0)}.$$

For $t = 0$ the equation holds since both summations are empty. Assume that the equation holds for all $s > 0$ and for all $v \in V$ such that $\text{home}(v) = G_i$. We split the blocking influence $w_{\bar{B}_i}(t; z, v)$ according to the last blocker on each path,

$$w_{\bar{B}_i}(t; z, v) = \sum_{u \in B_i} w_{\bar{B}_i, u}(t; z, v),$$

where $w_{\bar{B}_i, u}(t; z, v)$ is the sum of path weights over all length-t paths from z to v in which the last vertex in B_i is u. Splitting the paths from z to v at u we get

$$w_{\bar{B}_i, u}(t; z, u) = \sum_{s=1}^{t-1} w_{B_i}(s; u, v)\, w(t-s; z, u).$$

[2] Since we do not want to destroy the sparsity of A, and we wish our technique to generalize, we do not take advantage of techniques such as repeated squaring.

We now have

$$\begin{aligned}
\sum_{z \in V} w_{\bar{B}_i}(t; z, v)\, x_z^{(0)} &= \sum_{z \in V} \sum_{u \in B_i} w_{\bar{B}_i, u}(t; z, v)\, x_z^{(0)} \\
&= \sum_{u \in B_i} \sum_{z \in V} w_{\bar{B}_i, u}(t; z, v)\, x_z^{(0)} \\
&= \sum_{u \in B_i} \sum_{z \in V} \sum_{s=1}^{t-1} w_{B_i}(t-s; u, v) \\
&\quad \times w(s; z, u)\, x_z^{(0)} \\
&= \sum_{u \in B_i} \sum_{s=1}^{t-1} w_{B_i}(s; u, v) \\
&\quad \times \sum_{z \in V} w(t-s; z, u)\, x_z^{(0)} \\
&= \sum_{u \in B_i} \sum_{s=1}^{t-1} w_{B_i}(t-s; u, v)\, x_u^{(t-s)},
\end{aligned}$$

where the last equality follows by the inductive assumption. ∎

3. SIMPLE LINEAR SIMULATION

In this section, we present our I/O-efficient algorithm to perform a simple linear relaxation computation on any graph with a (τ, r, M)-blocking-cover $(\mathscr{G}, \mathscr{B})$. We call it a "simulation" algorithm, because it has the same effect as executing a simple linear relaxation algorithm, but it does not perform the computation in the same way. The simulation algorithm is not a new numerical algorithm. Rather, it is a new way to *implement* a numerical algorithm. Since we only present a new implementation strategy, convergence properties are maintained. Given a numerical algorithm, if both a conventional implementation and our implementation are executed on an ideal computer with no rounding errors, the output is exactly the same. In this section, we give an overview of the simulation algorithm and analyze its performance. In Section 7 we will discuss the issue of rounding errors.

The goal of the simulation algorithm is to compute the state vector $x^{(T)}$ in Eq. (3) given an initial state vector $x^{(0)}$ and a number T of steps. The algorithm has four phases, numbered 0 through 3. Phase 0 is executed once as a precomputation step. It computes the coefficients $w_{B_i}(s; u, v)$ in the second summation of Eq. (4) that express the influence of one blocker variable on another. Phases 1–3 advance the state vector by τ steps each time they are executed, and these steps are then iterated until the state vector has been advanced by a total of T steps. Phase 1 computes the first summation in Eq. (4) for the blocker variables which, in combination with the coefficients computed in Phase 0, yields a triangular linear system of equations on the blockers. Phase 2 solves these equations for the blockers using back substitution. Finally, Phase 3 extends the solution for the blocker variables to all the state variables. If $r^2\tau^2 \leqslant M$, each iteration of Phases 1–3 performs $O(\tau E)$ work, performs $O(E)$ I/O's, and advances the state vector by τ steps, as compared with the naive algorithm, which would perform $O(\tau E)$ I/O's for the same effect. Phase 0, the precomputation phase of the algorithm, requires $O(r\tau E)$ work and $O(E)$ I/O's.

We now describe each phase in more detail.

The goal of Phase 0 is to compute the coefficients $w_{B_i}(s; u, v)$ in the second summation of Eq. (4) for all $s = 1, 2, ..., \tau - 1$, for all $u \in B_i$, and for all $v \in B$ where $B = \bigcup_{B_i \in \mathscr{B}} B_i$ and $G_i = \text{home}(v)$. The coefficient $w_{B_i}(s; u, v)$ represents the influence that the value of u at time $\tau - s$ has on the value of v at time τ. The influence (coefficient) of blocker u on another in the same time step ($s = 0$) is 0, unless the other vertex is in fact u, in which case the influence is 1. Inductively, suppose that the state variable for each vertex v contains $w_{B_i}(s; u, v)$. To compute the $w_{B_i}(s+1; u, v)$, we set the blocker variables to 0 and run one step of linear relaxation on G_i. The value for $w_{B_i}(s+1; u, v)$ is produced in the state variable for v. To compute up to $s = \tau - 1$, this computation is therefore nothing more than a linear relaxation computation in which the blockers are zeroed at every step. Intuitively, this method works because any individual coefficient can be obtained from Eq. (4) by setting all the state variables in both summations to 0, except for that state variable in the second summation which is multiplied by the desired coefficient, which we set to 1. During Phase 0, any coefficient that represents an influence of a blocker on a blocker whose home is G_i is saved for use in Phase 2.

In Phase 1 we focus on the first summation in Eq. (4). Phase 1 computes the sums $\sum_{u \in V_i} w_{B_i}(t; u, v)\, x_u^{(0)}$ for all $t \leqslant \tau$ and for all $v \in B$, where $G_i = \text{home}(v)$. These values represent the contribution of the initial state on v without taking into account contributions of paths that come from or pass through blockers in v's home. For a given subgraph G_i in the blocking cover for G, the first summation is simply a linear relaxation on the subgraph of G_i induced by $V_i - B_i$. Thus, we can compute this summation for all blocker variables whose home is G_i by a linear relaxation on G_i as follows. We initialize the state variables according to $x^{(0)}$, and then for each subsequent step, we use the value 0 whenever the computation requires the value of a blocker variable.

Phase 2 solves for the blocker variables. If we inspect Eq. (4), we observe that since we have computed the value of the first summation and the coefficients of the variables in the second summation, the equations become a linear system in the blocker variables. Furthermore, we observe that the system is triangular, in that each $x_v^{(i)}$ depends only on various $x_u^{(j)}$ where $j < i$. Consequently, we can use the

back substitution method [4, Section 31.4] to determine the values for all the blocker variables.

Phase 3 computes the state variables for the nonblocker vertices by performing linear relaxations in each subgraph as follows. For a subgraph G_i, we set the initial state according to $x^{(0)}$ and perform τ steps of linear relaxation, where at step i blocker variable $x_u^{(i)}$ is set to the value computed for it in Phase 2. We can show that the state variables for each node whose home is in G_i assume the same values as if they were assigned according to a linear relaxation of G with the initial state $x^{(0)}$ by using induction and the fact that each blocker variable assumes the proper value.

In Section 4 we prove that given a graph $G=(V, E)$ with a (τ, r, M)-blocking-cover such that $r^2\tau^2 \leqslant M$, a computer with $O(M)$ words of primary memory can perform $T \geqslant \tau$ steps of a simple linear relaxation on G using at most $O(TE)$ work and $O(TE/\tau)$ I/O's. The precomputation phase (which does not depend on the initial state) requires $O(r\tau E)$ work and $O(E)$ I/O's.

4. THE ALGORITHM IN DETAIL

This section contains the details of our basic algorithm, which was outlined in Section 3. We begin by describing the data structures used by our algorithm and then present the details of each of the four phases of the algorithm. We give pseudocode for the phases and lemmas that imply their correctness.

Data Structures

The main data structure that the algorithm uses is a table S which during the algorithm contains information about vertices in one of the subgraphs G_i in the blocking cover of G with respect to the blocking set B_i. Each row $S[j]$ of S contains several fields of information about one vertex. The field $S[j].Name$ contains the vertex index in G, the boolean field $S[j].IsInB$ denotes whether the vertex belongs to $B=\bigcup_{B_i \in \mathscr{B}} B_i$, the boolean field $S[j].IsBlocker$ denotes whether the vertex belongs to B_i, and the boolean field $S[j].IsHome$ denotes whether the home of the vertex is G_i. The field $S[j].Adj$ is an adjacency list of the neighbors of the vertex in G_i (incoming edges only). Each entry in the adjacency list is the index of a neighbor in S, together with the relaxation weight of the edge that connects them. The two last fields are numeric fields $S[j].x$ and $S[j].y$. The field $S[j].x$ holds the initial state of the vertex, and after the algorithm terminates the field $S[j].y$ holds the state after time step τ if $S[j].IsHome$ is set.

A data structure S_i is stored in secondary memory for each subgraph G_i in the cover. In addition to S_i we store in secondary memory a table H_i for each subgraph G_i. For every vertex v whose home is G_i, the table lists all the subgraphs G_j in the blocking cover containing v and the index of v in S_j. These tables enable us to disperse the value of $x_v^{(\tau)}$ from the home of v to all the other G_j which contain v. The size of each S_i is $6|V_i|+2|E_i|$. The total amount of secondary storage required to store the blocking cover is $O(E)$.

We also store in secondary memory two 2-dimensional numeric tables WX and X of size τ-by-$|B|$, and one 3-dimensional numeric table W of size τ-by-r-by-$|B|$. The table W is used to store the coefficients $w_{B_i}(s; u, v)$ for all $s<\tau$, for all $u \in B_i$, and for all $v \in B$, where $G_i = \text{home}(v)$. The table WX is used to store the sum $\sum_{u \in V_i} w_{B_i}(t; u, v)\, x_u^{(0)}$ for all $t \leqslant \tau$, and all $v \in B$, where $G_i = \text{home}(v)$. The table X is used to store the values $x_v^{(t)}$ for all $t \leqslant \tau$ and all $v \in B$.

Phase 0

The pseudocode below describes Phase 0 of the algorithm. The influence of one blocker on all other blockers in a subgraph is computed by Procedure BLOCKERSINFLUENCE. Procedure PHASEZERO loads one subgraph at a time into primary memory, and then calls BLOCKERSINFLUENCE at most r times. Before each call exactly one vertex whose $S[j].IsBlocker$ field is set is chosen, its $S[j].x$ field is set to 1 and all the other x fields are set to 0. The index i of the blocker whose influence is computed is passed to BLOCKERSINFLUENCE.

```
PHASEZERO( )
1  for i ← 1 to k
2      do load S_i into primary memory
3         for b ← 1 to M
4             do if S[b].IsBlocker
5                   then for l ← 1 to M do S[l].x ← 0
6                        S[b].x ← 1
7                        BLOCKERSINFLUENCE(b)
```

```
BLOCKERSINFLUENCE(b)
1  for s ← 1 to τ − 1
2      do for j ← 1 to M
3             do S[j].y ← Σ_{(l,a)∈S[j].Adj} a · S[l].x
4         for j ← 1 to M
5             do if S[j].IsBlocker
6                   then S[j].x ← 0
7                   else S[j].x ← S[j].y
8                if S[j].IsInB and S[j].IsHome
7                   then write S[j].y to W[s, S[b].Name,
                         S[j].Name]
```

LEMMA 2. *After Phase 0 ends, for each $v \in B$, $u \in B_i$ and $s<\tau$, we have*

$$W[s, u, v] = w_{B_i}(s; u, v),$$

where $G_i = \text{home}(v)$.

Proof. We denote the state vectors in phase 0 by $e^{(t)}$ instead of $x^{(t)}$ to indicate that the initial state is a unit vector with 1 for one blocker and 0 for all the other vertices.

We prove by induction on s that lines 1–7 of BLOCKERSINFLUENCE perform linear relaxation on G with all outgoing edges from blockers in B_i removed, on all the vertices v for which $N_{B_i}^{(s)}(v) \subseteq V_i$ and on all the blockers in B_i. The claim is true for $s = 0$ because before the first iteration the state $S[j] \cdot x$ of every vertex is the initial state set by PHASEZERO. Assume that the claim is true for $s < \tau - 1$. In the next iteration $S[j].y$ is assigned the weighted linear combination of all of its neighbors in G_i. If the vertex v is a blocker, its state is zeroed in line 6 and the claim holds. If the vertex is not a blocker but $N_{B_i}^{(s+1)}(v) \subseteq V_i$, then all its neighbors are in G_i, and each neighbor u is either a blocker or $N_{B_i}^{(s)}(u) \subseteq V_i$. In either case, the y field is assigned the weighted linear combination of vertex states which are correct by induction, so its own state is correct.

The initial state is 0 for all vertices except for one blocker $u = S[b].Name$ whose initial state is $e_u^{(0)} = 1$. By Eq. (3) and condition BC4 in the definition of blocking-covers we have for all $s < \tau$ and $v \in V$ such that $G_i = \text{home}(v)$

$$\begin{aligned} e_v^{(s)} &= \sum_{z \in V} w_{B_i}(s; z, v)\, e_z^{(0)} \\ &= w_{B_i}(s; u, v). \end{aligned}$$

The value $e_v^{(s)} = w_{B_i}(s; u, v)$ is written to $W[s, u, v]$ in line 9 for all $v \in B$ such that $G_i = \text{home}(v)$. ∎

Let us analyze the amount of work and the number of I/O's required in Phase 0. BLOCKERSINFLUENCE is called at most rk times, where k is the number of subgraphs in the cover. In each call, the amount of work done is $O(\tau M)$ so the amount of work is $O(rkM\tau) = O(r\tau E)$. The total number of I/O's is $O(E)$ to load all the S_i into primary memory, and $|B|\, r\tau \leqslant kr^2\tau = O(E)$ to store the table W (since $W[s, *, v]$ is a sparse vector).

Phase 1

Phase 1 is simpler than Phase 0. Procedure INITIALSTATEINFLUENCE is similar to Procedure BLOCKERSINFLUENCE in Phase 0, but the table WX is written to secondary memory instead of the table W. Procedure PHASEONE loads one subgraph at a time and calls INITIALSTATEINFLUENCE once, with the initial state loaded from secondary memory.

```
PHASEONE( )
1  for i ← 1 to k
2      do load S_i into primary memory
3          INITIALSTATEINFLUENCE( )
```

```
INITIALSTATEINFLUENCE( )
1  for s ← 1 to τ
2      do for j ← 1 to M
3             do S[j].y ← Σ_{(l,a) ∈ S[j].Adj} a · S[l].x
4         for j ← 1 to M
5             do if S[j].IsBlocker
6                   then S[j].x ← 0
7                   else S[j].x ← S[j].y
8                if S[j].IsInB and S[j].IsHome
9                   then write S[j].y to WX[s, S[j].Name]
```

LEMMA 3. *After Phase* 1 *ends, for each* $v \in B$ *and* $s \leqslant \tau$, *we have*

$$WX[s, v] = \sum_{u \in V_i} w_{B_i}(s; u, v)\, x_u^{(0)},$$

where $G_i = \text{home}(v)$.

Proof. Lines 1–7 of INITIALSTATEINFLUENCE simulate linear relaxation on G with all outgoing edges from blockers in B_i removed. The proof of this claim is identical to the proof of Lemma 2, with the initial state being the given initial state $x^{(0)}$. Therefore we have for all $s \leqslant \tau$ and $v \in V$ such that $G_i = \text{home}(v)$,

$$x_v^{(s)} = \sum_{z \in V} w_{B_i}(s; z, v)\, x_z^{(0)}. \tag{6}$$

This value is written to $WX[s, v]$ in line 9 for all $v \in B$ such that $G_i = \text{home}(v)$. ∎

The total amount of work in Phase 1 is $O(k\tau M) = O(\tau E)$. The number of I/O's is $O(E)$ to load all the subgraphs and $|B|\, \tau$ to store the table WX.

Phase 2

Phase 2 solves the lower triangular system of linear equations defined by Lemma 1 for every $v \in B$ and all $t \leqslant \tau$. Entries in the tables W, WX, and X are written and read from secondary memory as needed.

```
PHASETWO( )
1  for t ← 1 to τ
2      do for v ← 1 to |B|
3             do Let G_i be the home of v
4                X[t, v] ← WX[t, v] + Σ_{1 ⩽ s < t, u ∈ B_i}
                     W[s, u, v] X[t − s, u]
```

LEMMA 4. *After Phase* 2 *ends we have for each* $v \in B$ *and* $t \leqslant \tau$,

$$X[t, v] = x_v^{(t)}.$$

Proof. The result follows immediately from Lemma 1 and the previous two lemmas. ▮

Since the number of terms in each of the $T|B|$ sums is at most rT, the total amount of work and I/O's is $O(r|B|\tau^2) = O(kr^2\tau^2) = O(E)$.

Phase 3

The structure of Phase 3 is similar to the structure of Phase 1. The main difference between the two phases is that in Phase 1 a zero was substituted for the state of a blocker during the simulation, whereas in Procedure RELAXG the correct value of the state of blockers is loaded from the table X in secondary memory. Procedure PHASETHREE loads each subgraph and its initial state to primary memory, calls RELAXG, and then stores back the subgraph with the correct state in the y field.

```
PHASETHREE( )
1  for i ← 1 to k
2      do load S_i into primary memory
3         RELAXG( )
4         store S_i back to secondary memory
```

```
RELAXG( )
 1  for s ← 1 to τ
 2      do for j ← 1 to M
 3             do S[j].y ← Σ_{(l,a) ∈ S[j].Adj} a · S[l].x
 4         for j ← 1 to M
 5             do if S[j].IsBlocker
 6                then read X[S[j].name, s] into S[j].x
 7                else S[j].x ← S[j].y
 8  for j ← 1 to M
 9      do if S[j].IsHome
10            then S[j].y ← S[j].x
```

LEMMA 5. *After Phase* 3 *ends, for every* $v \in V$ *whose home is* G_i, *the* y *field in the entry of* v *in* S_i *is* $x_v^{(\tau)}$.

Proof. We prove by induction on s that lines 1–7 of RELAXG simulate linear relaxation on G on all the vertices v for which $N_{B_i}^{(s)}(v) \subseteq V_i$ and on all the blockers in B_i. The claim is true for $s = 0$ because before the first iteration the state $S[j].x$ of every vertex is the initial state loaded from secondary memory. Assume that the claim is true for $s < \tau$. In the next iteration $S[j].y$ is assigned the weighted linear combination of all of its neighbors in G_i. If the vertex v is a blocker, its state is loaded from the table X in line 6 and the claim holds. If the vertex is not a blocker but $N_{B_i}^{(s+1)}(v) \subseteq V_i$, then all its neighbors are in G_i, and each neighbor u is either a blocker or $N_{B_i}^{(s)}(u) \subseteq V_i$. In either case, the y field is assigned the weigheted linear combination of vertex states which are correct by induction, so its own state is correct.

The lemma follows from the inductive claim, since if $G_i = \text{home}(v)$ then $N^{(\tau)}(v) \subseteq V_i$ and therefore its y field is assigned $x_v^{(\tau)}$. ▮

In Phase 3 each subgraph is loaded into primary memory and RELAXG is called. The total amount of work is $O(k\tau M) = O(\tau E)$ and the total number of I/O's is $O(E + V) = O(E)$.

Summary

The following theorem summarizes the performance of our algorithm.

THEOREM 6 (Simple linear simulation). *Given a graph* $G = (V, E)$ *with a* (τ, r, M)*-blocking-cover such that* $r^2\tau^2 \leqslant M$, *a computer with* $O(M)$ *words of primary memory can perform* $T \geqslant \tau$ *steps of a simple linear relaxation on* G *using at most* $O(TE)$ *work and* $O(TE/\tau)$ *I/O's. A precomputation phase* (*which does not depend on the initial state*) *requires* $O(r\tau E)$ *work and* $O(E)$ *I/O's.*

Proof. The correctness of the algorithm follows from Lemma 5. The bounds on work and I/O's follow from the performance analysis following the description of each phase. ▮

The simple linear simulation theorem applies directly in many situations, but in some special cases which are common in practice, we can improve the performance of our method. In Section 5 we will exploit two such improvements to obtain better I/O speedups.

5. MULTIGRID COMPUTATIONS

In this section we present the application of our method to multigrid relaxation algorithms. We show that a two-dimensional multigrid graph (shown previously in Fig. 2) has a $(\Theta(M^{1/6}), \Theta(M^{1/3}), M)$-blocking-cover, and hence, we can implement a relaxation on the multigrid graph using a factor of $\Theta(M^{1/6})$ fewer I/O's than the naive method. We improve this result to $\Theta(M^{1/5})$ for multigrid computations such as certain elliptic solvers that use only one level of the multigrid graph at a time and have a regular structure of relaxation weights. We conclude by discussing our results in the context of the "red-blue pebble game" I/O model of Hong and Kung [5].

LEMMA 7. *For any* $\tau \leqslant \sqrt{M}$, *a two-dimensional multigrid graph* G *has a* (τ, r, M)*-blocking-cover, where* $r = O(\tau^2)$.

Proof. Consider a cover of a multigrid graph in which every $G_i = (V_i, E_i)$ consists of a k-by-k submesh at the bottommost level together with all the vertices above it in the multigrid graph, and the blocking set $B_i \subset V_i$ consists of all the vertices in levels $l + 1$ and above. Let each subgraph G_i

be the home of all vertices in the inner $(k-\tau 2^{l+1})$-by-$(k-\tau 2^{l+1})$ bottommost submesh of G_i and all the vertices above them. The number of vertices in B_i is

$$\begin{aligned} r &= \sum_{i=l+1}^{(\lg n)/2} \left(\frac{k}{2^i}\right)^2 \\ &< \left(\frac{k}{2^{l+1}}\right)^2 \sum_{i=0}^{\infty} 4^{-i} \\ &= \frac{4}{3}\left(\frac{k}{2^{l+1}}\right)^2 \end{aligned}$$

and the number of edges in G_i is

$$\begin{aligned} |E_i| &= (2+\tfrac{1}{4})\,|V_i| \\ &< \tfrac{9}{4}\cdot\tfrac{4}{3}\,k^2 \\ &= 3k^2, \end{aligned}$$

since there are at most two edges for each vertex in a mesh, and in the multigrid, $\frac{1}{4}$ of the $O((4/3)\,k^2)$ vertices have edges connecting to a higher level mesh. Setting $k-\tau 2^{l+1}=k/2$, we obtain $k=4\tau 2^l$ and $r<(4/3)(2\tau)^2=(16/3)\,\tau^2$. Setting $l=\frac{1}{2}\lg(M/\tau^2)$, we obtain $|E_i|<48M$. ∎

Combining Theorem 6 and Lemma 7, we obtain the following result.

COROLLARY 8. *A computer with $\Theta(M)$ words of primary memory can perform $T=\Omega(M^{1/6})$ steps of a simple linear relaxation on a $\sqrt{n}$-by-$\sqrt{n}$ multigrid graph using $O(Tn)$ work and $O(Tn/M^{1/6})$ I/O's A precomputation phase requires $O(M^{1/2}n)$ work and $O(n)$ I/O's.*

Proof. Set $\tau=M^{1/6}$ in Lemma 7 and apply Theorem 6. ∎

As a practical matter, linear relaxation on a multigrid graph is not simple: it does not use the same relaxation matrix at each step. Moreover, for many applications, a given step of the relaxation is performed only on a single level of the multigrid or on two adjacent levels.

For example, one generic way to solve a discretized version of a parabolic two-dimensional heat equation in the square domain $[0,1]^2$, as well as a wide variety of other time-dependent systems of partial differential equations, such as the Navier–Stokes equations, is to use discrete time steps, and in each time step to solve an elliptic problem on the domain. In the heat equation example, for instance, the elliptic problem is

$$\frac{\partial^2 u(x,y,t_i)}{\partial x^2}+\frac{\partial^2 u(x,y,t_i)}{\partial y^2}=\frac{u(x,y,t_i)-u(x,y,t_{i-1})}{t_i-t_{i-1}}.$$

In a common implementation of this strategy, the elliptic solver is a multigrid algorithm, in which case the entire solver can be described as a linear relaxation algorithm on a multigrid graph [3]. The algorithm consists of a number of *cycles*, where each cycle consists of $\Theta(\lg n)$ steps in which the computation proceeds level-by-level up the multigrid and then back down. Since the size of any given level of the multigrid is a constant factor smaller than the level beneath it, the $\Theta(\lg n)$ steps in one cycle of the algorithm execute a total of $\Theta(n)$ work and update each state variable only a constant number of times. Thus, a naive implementation of T cycles of the elliptic solver requires $O(nT)$ work and $O(nT)$ I/O's.

We can use the basic idea behind the simple linear simulation algorithm to provide a more I/O-efficient algorithm. We present the algorithm in the context of a multigrid graph which is used to solve an equation with constant coefficient, but the same algorithm works in other special cases.

DEFINITION. We say that a multigrid graph has *regular edge weights* if for every level, the edge weights in the interior of the grid are all identical, the edge weights in the interior of every face (or edge) of the grid are all identical, and if the edge weights going from one level to another are all identical in the interior and all identical along each face.

THEOREM. *A computer with $\Theta(M)$ words of primary memory can perform $T=\Omega(M^{1/5})$ multigrid cycles on a $\sqrt{n}$-by-$\sqrt{n}$ multigrid graph with regular edge weights using $O(nT)$ work, and $O(nT/M^{1/5})$ I/O's. A precomputation step requires $O(M^{8/5})$ work and $O(M)$ I/O's.*

Proof. The algorithm generally follows the simple linear relaxation algorithm. We outline the differences.

The linear simulation algorithm uses a $(\tau, r.\, M)$-blocking-cover as described in the proof of Lemma 7, but we now choose $\tau=M^{1/5}$ and $r=\Theta(M^{2/5})$. Because the relaxation algorithm is not simple, the paths defined by Eq. (2) must respect the weights defined by the various relaxation matrices. In a single cycle, however, there are only a constant number of relevant state variables for a single vertex. Moreover, the phases can skip over steps corresponding to updates at level $(3/10)\lg M+1$ and above, since only blocker variables occupy these levels. Most of these changes are technical in nature and, whereas the bookkeeping is more complicated, we can simulate one cycle of the elliptic solver with asymptotically the same number of variables as the simple linear simulation algorithm uses to simulate one step of a simple linear relaxation problem on the lowest level of the multigrid.

The real improvement in the simulation, however, comes from exploiting the regular structure of the blocking cover of the multigrid graph. The cover has three types of subgraphs: interior ones, boundary ones, and corner ones. All subgraphs of the same type have isomorphic graph structure, the same relaxation weights on isomorphic edges, and an isomorphic set of blockers. Thus, in Phase 0 of

the simulation algorithm, we only need to compute the influence on blockers in one representative subgraph of each type. We store these coefficients in primary memory for the entire algorithm, and hence, in Phase 2, we need not perform any I/O's to load them in. Phase 1 and 3 are essentially the same as for simple linear simulation.

The change to Phase 2 is what allows us to weaken the constraint $r^2\tau^2 \leqslant M$ from Theorem 6 and replace it by $r^2\tau \leqslant M$, which arises since the total work $(r\tau)^2 E/M$ in Phase 2 must not exceed $E\tau$ if we wish to do the same work as the naive algorithm. Because all three types of subgraphs must fir into primary memory at the same time, the contraint $3r^2\tau \leqslant M$ also arises. Maximizing τ under the constaints of Lemma 7 yields the choice $\tau = M^{1/5}$. The work in Phase 2 is $O((r\tau)^2 E/M) = O(M^{1/5}n)$, rather than $O(n)$ as it would be without exploiting the regularity of subgraphs. The number of I/O's in Phase 2 is $O(r\tau)\, E/M) = O(n)$ in order to input the constants computed in Phase 1 corresponding to the first summation in Eq. (4). The work in Phases 1 and 3 is $O(\tau E) = O(M^{1/5}n)$, and the number of I/O's is $O(r\tau E/M) = O(n)$. The amount of work in Phase 0 becomes $O(3r\tau M) = O(M^{8/5})$, and the number of I/O's for this precomputation phase is $O(M)$. ▌

We mention two extensions of this theorem. The three-dimensional multigrid graph has a (τ, r, M)-blocking-cover, where $r = O(\tau^3)$, which yields an I/O savings of a factor of $\tau = \Theta(M^{1/7})$ over the naive algorithm when τ is maximized subject to the constraint $r^2\tau \leqslant M$. For the two-dimensional problem, one can exploit the similarity of the coefficients computed by Phase 0 to save a factor of as much as $\Theta(M^{1/3})$ in I/O's over the naive method, but at the expense of doing asymptotically more work.

We conclude this section with a discussion of Hong and Kung's "red–blue pebble game" [5]. This pebble game is a formal model for studying the I/O requirements of out-of-core algorithms. The model assumes that an algorithm is given as a directed acyclic graph (dag) in which nodes represent intermediate values in the computation. The only constaint in this model is that all predecessors of a node must reside in primary memory when the state of the node is computed. Other than this constraint, the red–blue pebble game allows arbitrary scheduling of the dag. In a linear relaxation computation, for example, each node in the dag corresponds to a state variable, and its predecessors are the state variables of its neighbors at the previous time step. The arbitrary scheduling allowed in the red–blue pebble game can be effective in reducing I/O as outlined in Section 1 for relaxation on multidimensional meshes. It has also been applied to various other problems (see [1] for examples).

Elliptic problems with constant coefficients are often solved using algorithm based on the fast Fourier transform rather than multigrid algorithms. Hong and Kung [5] showed that under the assumptions of the red–blue pebble game, the reduction in I/O for the FFT is limited to $\Theta(\lg M)$.

We now show that under the assumptions of the red–blue pebble game, the reduction in I/O for the multigrid computation is limited to $O(1)$. In other words, *no* asymptotic saving is possible in this model.

Multigrid algorithms belong to a large class of numerical methods, including Krylov subspace methods such as conjugate gradient, which have a common information flow structure. The methods iteratively update a state vector. The dag associated with the methods contains a set of nodes corresponding to the state vector variables for each iteration of the algorithm, plus some intermediate variables. The state of a variable at the end of iteration t is an immediate predecessor in the dag of the state of the variable at the end of iteration $t+1$. In addition, the states of *all* variables at the end of iteration t are indirect predecessors of every variable at the end of iteration $t+1$. In multigrid algorithms we associate the state vector with the state of the finest mesh, the intermediate variables with the state of all other meshes, and each iteration corresponds to a multigrid cycle in which information is transferred from the finest mesh to the coarsest and back to the finest.

THEOREM 10. *Let D be the dag corresponding to a T-step iterative computation with an n-node state vector $x^{(t)}$, in which the state $x_v^{(t+1)}$ of a node v after iteration $t+1$ depends directly on $x_v^{(t)}$ and indirectly on the $x_u^{(t)}$ for all state variables u. Then any algorithm that satisfies the assumptions of the red–blue pebble game requires at least $T(n-M)$ I/O's to simulate the dag D on a computer with M words of primary memory.*

Proof. The red–blue pebble game allows redundant computations, and therefore the state of a vertex v after time step t, $x_v^{(t)}$, may be computed more then once during the course of the execution of the algorithm. Let $\text{Time}_v^{(t)}$ be the first instant during the execution of the algorithm in which $x_v^{(t)}$ is computed. We denote by $\text{Time}^{(t)}$ the first instant in which the state of some vertex after time step t is computed,

$$\text{Time}^{(t)} = \min_{v \in V} \{\text{Time}_v^{(t)}\}.$$

The state of each vertex v after iteration $t+1$, $x_v^{(t+1)}$, depends on the state of all vertices after iteration t. Therefore we deduce that $\text{Time}^{(0)} < \text{Time}^{(1)} < \cdots < \text{Time}^{(T)}$ and that algorithm must compute the state of all vertices after iteration t between $\text{Time}^{(t)}$ and $\text{Time}^{(t+1)}$.

Let $C_u^{(t)}$ be the chain of vertices in the dag $x_u^{(0)} \to x_u^{(1)} \to \cdots \to x_u^{(t)}$. If $x_u^{(t)}$ is computed between two time points $\text{Time}^{(t)}$ and $\text{Time}^{(t+1)}$, we know either that a vertex in $C_u^{(t)}$ was in memory at $\text{Time}^{(t)}$ or one I/O was performed between $\text{Time}^{(t)}$ and $\text{Time}^{(t+1)}$ in order to bring some vertex in $C_u^{(t)}$ into primary memory.

The vertex sets $C_u^{(t)}$ and $C_w^{(t)}$ are disjoint for $u \neq w$. Since primary memory at time Time$^{(t)}$ can contain at most M vertices, one vertex from a least $n-M$ chains $C_u^{(t)}$ must be brought from secondary memory between Time$^{(t)}$ and Time$^{(t+1)}$. Summing over all iterations, we conclude that the algorithm must perform at least $T(n-M)$ I/O's. ▌

Since the total work in each multigrid cycle is $\Theta(n)$, the performance of the naive out-of-core algorithm matches the asymptotic performance of any algorithm which satisfies the red-blue pebble game assumptions, if $M < 2n$. Corollary 8 and Theorem 9 show that by simulating the dag in an unorthodox fashion, this lower bound does not apply. In particular, our method does not always compute the state of a node from the previous state of its neighbors.

6. FINDING BLOCKING COVERS

In this section, we describe how to find blocking covers for graphs that arise naturally in finite-element computations for physical space. Consequenctly, I/O efficient linear relaxation schemes exist for these classes of graphs. Specially, we focus on planar graphs to model computations in two dimensional and d-dimensional simplicial graphs of bounded aspect ratio to model computations in higher dimensions. Planar graphs are those that can be drawn in the plane so that no edges cross. Simplicial graphs arise from dividing d-dimensional space into polyhedra whose aspect ratio is bounded and where the sizes of polyhedra are *locally similar*: the volume of a polyhedron is no larger than twice (or some constant times) the volume of any neighboring polyhedron. Linear relaxation algorithms on such graphs can be used to solve differential equations on various d-dimensional structures [9, 10].

We begin by defining simplicial graphs formally using definitions from [9].

DEFINITION. A *k-dimensional simplex*, or *k-simplex*, is the convex hull of $k+1$ affinely independent points in $\mathfrak{R}^d$. A simplicial complex is a collection of simplices closed under subsimplex and intersection. A *k-complex* K is a simplicial complex such that for every k'-simplex in K, we have $k' \leqslant k$.

A 3-complex is a collection of cells (3-simplexes), faces (2-simplices), edges (1-simplices), and vertices (0-simplices). A *d-dimensional simplicial graph* is the collection of edges (1-simplices) and vertices (0-simplices) in a k-complex in d-dimensions. The *diameter* of a k-complex is the maximum distance between any pair of points in the complex, and the *aspect ratio* is the ratio of the diameter to the kth root of the volume. A simplicial graph of aspect ratio α is a simplicial graph that comes from a k-complex with every k-simplex having aspect ratio at most α.

We now state the main theorems of this section.

THEOREM 11. *A computer with $\Theta(M)$ words of primary memory can perform $T \geqslant \tau$ steps of simple linear relaxation on any n-vertex planar graph using $O(nT)$ work and $O(nT/\tau)$ I/O's, where $\tau = O(M^{1/4}/\sqrt{\lg n})$. A precomputation phase requires $O(\tau^2 n \lg n)$ work and $O(n)$ I/O's. Computing the blocking cover requires $O(n \lg n)$ work and I/O's.*

THEOREM 12. *A computer with $\Theta(M)$ words of primary memory can perform $T \geqslant \tau$ steps of simple linear relaxation on any n-vertex d-dimensional simplicial graph of constant aspect ratio using $O(nT)$ work and $O(nT/\tau)$ I/O's, where $\tau \leqslant M^{\Omega(/d)}/\lg n$. A precomputation step requires $(\tau^{\Omega(d)} n \lg^{\Omega(d)} n)$ work and $O(n)$ I/O's. Computing the blocking cover requires $(n^2/\tau + nM)$ work and I/'s.*

These theorems follow from the fact that good blocking covers can be found for planar and simplicial graphs by extending the techniques of [6, 12]. We proceed by stating the definition of a cut cover from [6], and then we relate cut covers to blocking covers. We describe recent results from [6, 12] that describe how to find good cut covers, and thus, how to find good blocking covers for planar and simplicial graphs.

Given a subgraph $G_i = (V_i, E_i)$ of a graph $G = (V, E)$ with vertex and edge weights w: $V \cup E \to \{0, 1\}$, we define the weight of G_i as $w(G_i) = \sum_{v \in V_i} w(v) + \sum_{e \in E_i} w(e)$. The following definitions are slight modifications of definitions in [6].

DEFINITION. A *balanced (τ, r, ε)-cut-cover* of a graph $G = (V, E)$ with vertex and edge weights w: $V \cup E \to \{0, 1\}$ is a triplet (C, G_1, G_2), where $C \subseteq V$ is called a *cut set* and $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are subgraphs of G, such that

CC1. $|C| \leqslant r$;

CC2. $w(G_1) + w(G_2) \leqslant (1+\varepsilon)\, w(G)$;

CC3. $\max(w(G_1), w(G_2)) \leqslant 2w(G)/3$;

CC4. $\forall v \in V$, either $N_C^{(\tau)}(v) \subseteq V_1$ or $N_C^{(\tau)}(v) \subseteq V_2$.

DEFINITION. A *two-color (τ, r, ε)-cut-cover* for a graph $G = (V, E)$ with two weight functions w_1, w_2: $V \cup E \to \{0, 1\}$ is a triplet (C, G_1, G_2) which constitutes a balanced (τ, r, ε)-cut-cover for G both weight functions.

The following theorem relates cut covers to blocking covers.

THEOREM 13. *If every subgraph of a graph, $G = (V, E)$, has a two-color $(\tau, r, O(1/\lg E))$-cut-cover for any two weight functions, then the graph has a $(\tau, 3r, M)$-blocking-cover.*

Proof. We find a blocking cover by recursively taking two-color cut-covers of subgraphs of G with respect to two weight functions. One weight function w_E assigns weight 1 to each edge in the graph and weight 0 to each vertex. The

second weight function w_B assigns 1 to any node that was designated to be a blocker at a higher recursive level and assigns 0 to any other node or edge. That is, we find a two-color $(\tau, r, \varepsilon = O(1/\lg E))$-cut-cover (B, G_1, G_2) on the current subgraph, G, and then repeat on each of G_1 and G_2, where w_B and w_E for G_i is inherited from G, except that the new w_B assigns 1 to any element of B in G_i.

We now argue that the set of subgraphs generated at level $\log_{3/2} |E|/M$ of the recursive decomposition is a $(\tau, 3r, M)$-blocking cover of G. The set of subgraphs forms a τ-cover since a (τ, r, ε) cut-cover is a τ-cover and successively taking τ-covers yields a τ-cover of the first graph. The number of blockers in any subgraph can be bounded by $3r$ as follows. Assume that at some recursive level, the current subgraph G contains $3r$ blockers from higher recursive levels. Then the number of blockers that G_1 or G_2 contains is less than $\frac{2}{3}(3r) + |B| \leqslant 3r$ by the definition of two-color cut-cover. After $\log_{3/2} |E|/M$ recursive levels, the largest subgraph has at most M edges, since the number of edges in a subgraph is reduced by at least $\frac{2}{3}$ at each recursive level. Finally, the total number of edges in the set of subgraphs is at most $(1+\varepsilon)^{\log_{3/2}(|E|/M)} |E| \leqslant e^{\varepsilon \log_{3/2}(|E|/M)}|E| = O(E)$, since the total number of edges does not increase by more than $(1+\varepsilon)$ at each recursive level. ▌

Kaklamanis, Krizanc, and Rao [6] have shown that for every integer l, every n-vertex planar graph has a two-color $(\tau, O(l), \tau/l)$-cut-cover which can be found in $O(n)$ time. Moreover, Plotkin, Rao, and Smith [12] have recently shown that for every l, every n-vertex d-dimensional simplicial graph of constant aspect ratio has a two-color $O(\tau, O(l^{O(d)} \lg n), \tau/l)$-cut-cover that can be found in $O(n^2/l)$ time.[3] These results can be combined with Theorem 13 to yield the following corollaries.

COROLLARY 14. *For every $\tau > 0$, every n-vertex planar graph has a (τ, r, M)-blocking cover, where $\tau = O(r/\lg n)$.*

COROLLARY 15. *For every $r > 0$, every n-vertex d-dimensional simplicial graph with constant aspect ratio has a (τ, r, M)-blocking cover, where $\tau = O(r^{\Theta(1/d)}/\lg^{1+\Theta(1/d)} n)$.*

Corollary 14 and Corollary 15 can be combined with Theorem 6 to prove Theorem 11 and Theorem 12.

7. PRACTICAL CONSIDERATIONS

In this section we discuss two implementation issues, the constants involved with our multigrid algorithm and the numerical stability of our linear relaxation algorithm.

[3] In fact, they can find cut-covers in any graph that exludes a *shallow* minor. That is, they consider graphs that do not contain K_h as a minor, where the set of vertices in the minor that correspond to a vertex of K_h has small diameter. Our results also hold for this class of graphs.

Estimated Performance of the Multigrid Algorithm

We describe below the results of a careful analysis of the constants involved with the implementation of a two-dimensional multigrid algorithms using our method. We have made the following assumptions:

- The amount of data reuse should be around 10 primary memory accesses to every word fetched from secondary memory. This level of data reuse hides most of the low bandwidth of the I/O channel of a modern workstation.
- Information propagates distance of about four grid points at every of a multigrid computation. The information propagation is due to two or three relaxation steps on each level, plus the transfer of information from one level to another.

To achieve data reuse ratio of 10, the value of τ in our algorithm should be around 30, to compensate for the loading of the graph in both Phases 1 and 3, for the amount of overlap in the blocking cover, and for the I/O in Phase 2. We determined that, with a certain choice of parameters, our scheme requires that the size M in words of primary memory must satisfy $M > 5{,}570\tau^5$, or $M > 180 \times 10^9$, for $\tau = 32$. The constant is much larger in three dimensions.

We feel that these constants are too large for the method to be attractive for practical use on workstations—the amount of memory required is larger than the address space of a 32-bit microprocessor and certainly much larger than the actual memory size of any existing workstation.

The method may still be useful for simulation of other graphs, such as planar graphs, and for situations where the amount of memory available is much larger, such as on supercomputers. Since the amount of memeory required is proportional to τ^5, it is very sensitive to the value of τ, and therefore, applying the algorithm with a smaller τ requires much less memory. On the other hand, the algorithm performs between 2–3 times more work than the naive algorithm, so it is not particularly useful for hiding small differences between the bandwidths of primary and secondary memory. We believe that our ideas do have practical value and that situations in which blockers and blocking covers are advantageous will arise in the future.

Numerical Stability

We now briefly discuss the numerical stability of our out-of-core linear relaxation algorithm. In exact arithmetic our algorithm yields the same output as the naive algorithm. Therefore, we need only to be concerned with rounding errors in floating-point arithmetic and not with convergence properties. There are two possible sources of instability in our algorithm. First, the linear relaxation which is used in Phase 0 and 1 is done on the graph with some edges

removed. This process may be unstable, leading to an inaccurate values for WX or for W. In addition, the matrix W may be ill-conditioned, which may lead to instability in Phase 2.

8. CONCLUSIONS

We conclude the paper with a short discussion of our results and with a reference to two important extensions of our work, which will be discussed in detail elsewhere.

Our out-of-core linear relaxation algorithm is the first algorithmic design method for out-of-core algorithms which is not based on scheduling and redundant computations. By exploiting problem-specific properties such as linearity, our method achieves speedups which are not possible by merely scheduling and computing redundantly. Although a careful analysis of the constants in our analysis leads us to believe that the multigrid algorithm would not be very useful in practice, we have found that the idea of blockers can lead to efficient out-of-core algorithms.

Baruch Awerbuch of MIT has pointed out that our approach of using blocking covers can be applied to reduce the latency of performing linear relaxation algorithms in parallel systems. For example, we can perform $O(N^{2/14})$ steps of a linear relaxation on a multigrid graph in a two-dimensional mesh-connected computer in $O(\sqrt{N})$ time, yielding an average latency of only $O(N^{5/14})$. Any *general computation* on a multigrid graph, however, requires an average latency of at least $\Omega(N^{1/2}/\lg N)$ [7].

Using ideas similar to the those developed in this paper, we have developed two other out-of-core numerical methods: one for Krylov space algorithms and the other for implicit linear relaxation on a line, a problem that arises from discretization of one-dimensional parabolic partial differential equations. One can show that the lower bound we proved in Theorem 10 holds for these methods as well, and therefore, only through the use of specific problem properties can one get an efficient out-of-core implementation. These extensions were discussed in a preliminary version of this paper [8], and we plan to describe them in detail in future papers.

ACKNOWLEDGMENTS

We thank one of the referees for his helpful comments.

REFERENCES

1. A. Aggarwal and J. S. Vitter, The input/output complexity of sorting and related problems, *Comm. ACM* **31**, No. 9 (1988), 1116–1127.
2. B. Awerbuch and D. Peleg, Sparse partitions, *in* "31st Symposium on Foundations of Computer Science, 1990," pp. 503–513.
3. W. L. Briggs, "A Multigrid Tutorial," SIAM, Philadelphia, 1987.
4. T. H. Cormen, C. E. Leiserson, and R. L. Rivest, "Introduction to Algorithms," MIT Press, McGraw–Hill, New York, 1990.
5. J.-W. Hong and H. T. Kung, I/O complexity: The red–blue pebble game, *in* "Proceedings, 13th Annual ACM Symposium on Theory of Computing, 1981," pp. 326–333.
6. C. Kaklamanis, D. Krizanc, and S. Rao, New graph decompositions and fast emulations in hypercubes and butterflies, *in* "Proceedings, 5th Annual ACM Symposium on Parallel Algorithms and Architectures, June 1993," pp. 325–334.
7. R. Koch, T. Leighton, B. Maggs, S. Rao, and A. Rosenberg, Work-preserving emulations of fixed-connection networks, *in* "Proceedings, 21st Annual ACM Symposium on Theory of Computing, May 1989," pp. 227–240.
8. C. E. Leiserson, S. Rao, and S. Toledo, Efficient out-of-core algorithms for linear relaxation using blocking covers, *in* "34rd Annual Symposium on Foundations of Computer Science, November 1993," pp. 704–713.
9. G. Miller and W. Thurston, Separators in two and three dimensions, *in* "Proceedings, 22nd Annual ACM Symposium on Theory of Computing, May 1990," pp. 300–309.
10. G. L. Miller, S.-H. Teng, and S. A. Vavasis, A unified geometric approach to graph separators, *in* "Proceedings, 32nd Annual Symposium on Foundations of Computer Science, 1991," pp. 538–547.
11. C. J. Pfeifer, Data flow and storage allocation for the PDQ-5 program on the Philco-2000, *Comm. ACM* **6**, No. 7 (1963), 365–366.
12. S. Plotkin, S. Rao, and W. Smith, Shallow excluded minors and improved graph decomposition, *in* "Proceedings, 5th Annual ACM-SIAM Symposium on Discrete Algorithms, 1994," pp. 462–470.