1

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

(Extended Abstract)

Charles E. Leiserson
MIT Lab for Computer Science
545 Technology Square
Cambridge, MA 02139

Satish Rao
NEC Research Institute
4 Independence Way
Princeton, NJ 08540

Sivan Toledo
MIT Lab for Computer Science
545 Technology Square
Cambridge, MA 02139

Abstract

When a numerical computation fails to fit in the primary memory of a serial or parallel computer, a so-called "out-of-core" algorithm must be used which moves data between primary and secondary memories. 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 from the graph. 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 graph, 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.

We use similar methods to reduce the I/O in other numerical algorithms, including conjugate gradient and implicit relaxation.

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 updated by a weighted linear combination of its neighbors:

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

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 $x^{(t)} = \left\langle x_1^{(t)}, x_2^{(t)}, \ldots, x_{|V|}^{(t)}\right\rangle^{\mathrm{T}}$ is the state vector for the tth step, and $A^{(t)} = \left(A_{uv}^{(t)}\right)$ is the **relaxation matrix** for the tth step, where 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].

A computer with ample primary memory can perform a linear relaxation computation by simply updating the state vector according to Equation (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

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

an ordinary, serial computer is $O(E)$. (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 $x_v^{(t)}$.) Ignoring 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. The goal of an out-of-core algorithm is 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 Equation (1) to all vertices, computing the entire state vector for one time step before proceeding to the next. This strategy causes $\Theta(TV)$ I/O transfers (I/O's) to be made for a T-step computation, if $|V| \geq 2M$.

For some graphs, however, there are more clever strategies that use many fewer I/O's. For instance, Hong and Kung [5] allude to 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 Figure 1. We load into primary memory the initial state of a k-by-k submesh S. 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 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.

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 Figure 2. The kth level is a $\sqrt{n}/2^k$-by-$\sqrt{n}/2^k$ mesh, for $k = 0, 1, \ldots, (\lg n)/2$, whose (i,j) vertex is connected to the $(2i, 2j)$ vertex on the $(k-1)$st mesh.

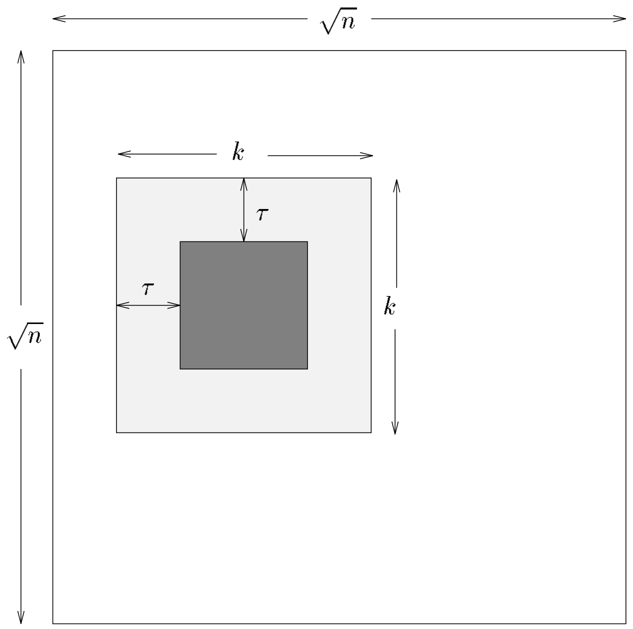

Figure 1: Performing an out-of-core linear relaxation computation on a mesh.

A typical multigrid application is 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 $T \geq \lg n$ 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 memory running T cycles of a $\sqrt{n}$-by-$\sqrt{n}$ multigrid algorithm, where $n \geq 2M$, 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 4 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 because of its small diameter. Consequently, it is impossible to tile the graph with overlapping subgraphs so that each subgraph can execute for a long time without requiring data from a neighboring subgraph.

Nevertheless, we shall see in Section 4 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 application of our basic result to multigrid relaxation is presented in Section 4. In Section 5 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 6 contains an overview of further results. In particular, we present efficient out-of-core implementations of a conjugate gradient algorithm and of an implicit solver for one-dimensional parabolic differential equations. We conclude the paper in Section 7 with a brief discussion of some other issues.

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 $G = (V, E)$, 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 $\boldsymbol{\tau}$**-neighborhood-cover** [2] of G is a set of subgraphs $\mathcal{G} = \langle G_1 = (V_1, E_1), \ldots, G_k = (V_k, E_k)\rangle$ such that for all $v \in V$, there exists a $G_i \in \mathcal{G}$ 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 $N_B^{(\tau)}(v) = \{u \in V : \exists$ a path $u \to u_1 \to \cdots \to u_t \to v$, where $u_i \in V - B$ for $i = 1, 2, \ldots, t < \tau\}$. 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 $(\mathcal{G}, \mathcal{B})$, where $\mathcal{G} = \langle G_1 = (V_1, E_1), \ldots, G_k = (V_k, E_k)\rangle$ is a sequence of subgraphs of G and $\mathcal{B} = \langle B_1, \ldots, B_k\rangle$ is a sequence of subsets of V such that

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

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

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

BC4. for all $v \in V$, there exists a $G_i \in \mathcal{G}$ 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 $r^2\tau^2 \leq M$. 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**.

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.[1] 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

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

where $\mathcal{P} = \{p \in G : p \text{ is a length-}r \text{ path from } u \text{ to } v\}$. (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

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

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

Lemma 1 *Let $G = (V, E)$ be a directed graph, and let $(\mathcal{G}, \mathcal{B})$ 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 $t \le \tau$*

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

where $G_i = home(v)$.

Proof: Induction and superposition (linearity). □

To understand Lemma 1, consider a subgraph G_i in the cover and its corresponding blocking set B_i. The lemma says that in order to know the value of a state variable $x_v^{(t)}$, it suffices to know the initial 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.

[1] 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.

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 $(\mathcal{G}, \mathcal{B})$. 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* numerical algorithms. 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 roundoff errors, the output is exactly the same. In this section, we describe the simulation algorithm and analyze its performance.

The goal of the simulation algorithm is to compute the state vector $x^{(T)}$ in Equation (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 Equation (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 Equation (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 $r^2\tau^2 \le M$, 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 Equation (4) for all $s = 0, 1, \ldots, \tau - 1$, for all $u \in B_i$, and for all v where $v \in B$ 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 Equation (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 Equation (4). Phase 1 computes the sums $\sum_{u\in V_i} w_{B_i}(t;u,v)x_u^{(0)}$ for all $t\leq\tau$ 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 Equation (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 non-blocker vertices by performing linear relaxations in each subgraph as follows. For a subgraph G_i, we set the initial state set 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.

The following theorem states the performance characteristics of the algorithm.

Theorem 2 (Simple Linear Simulation) *Given a graph $G=(V,E)$ with a (τ,r,M)-blocking-cover such that $r^2\tau^2\leq M$, a computer with $O(M)$ words of primary memory can perform $T\geq\tau$ 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: We need only determine the implementation costs of the simple linear simulation algorithm.

Phase 0 uses a τ-step linear relaxation on a subgraph G_i to compute the influence of a blocker u on all of the blockers whose home is G_i. That is, for a given blocker u in a subgraph G_i, Phase 0 uses $\tau|E_i|$ work to compute all the coefficients of the form $w_{B_i}(i;u,v)$, where v is a blocker and i belongs to the set $\{0,\ldots,\tau-1\}$. Since there are at most r blockers in any G_i, all the coefficients for a particular G_i can be computed with $r\tau|E_i|$ work. The number of I/O's, however, is only $O(E_i+r^2\tau)$, since G_i needs to be loaded into memory only once and there are only $r^2\tau$ coefficients that need to be stored in secondary memory for later use. Summing over all the subgraphs, we see that Phase 0 can be implemented using $O(r\tau|E|)$ work and $O(E+r^2\tau E/M)=O(E)$ I/O's.

Phase 1 uses a τ-step linear relaxation on each subgraph G_i to compute the value of the first summation in Equation (4) for each blocker variable. The total work is bounded by $O(\tau\sum_i|E_i|)=O(\tau E)$. The one-time transfer from secondary memory to primary memory of each subgraph of the cover requires $O(E)$ I/O's, because the size of the cover is $O(E)$. The storing of the values computed by Phase 1 requires $O(r\tau E/M)=O(E)$ I/O's, because the total number of outputs is bounded by the total number of blocker variables. Thus, the total number of I/O's for Phase 1 is $O(E)$.

Phase 2 uses back substitution to solve a triangular linear system on $O(r\tau E/M)$ equations. Each equation has at most $O(r\tau)$ nonzero coefficients, because indices in the double summation in Equation (4) range over the r elements of B_i and τ values for s. Consequently, the back substitution can be performed with $O((r\tau)^2E/M)=O(E)$ total work. The number of I/O's is also $O(E)$, since the number of I/O's is never more than the total work.

Finally, Phase 3 computes $x^{(\tau)}$ using a τ-step linear relaxation on each subgraph of the cover. The work involved is $O(\tau|E|)$, as in Phase 1. The number of

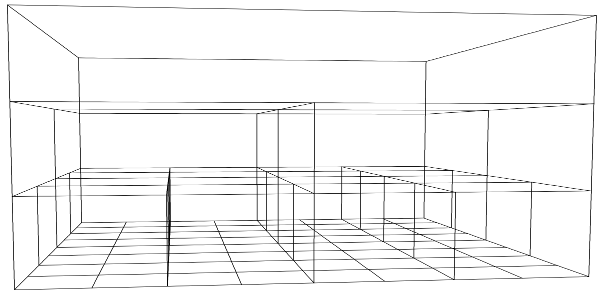

Figure 2: A 9-by-9 multigrid graph. The graph has levels 0, the bottommost, through 3, the topmost.

I/O's is $O(E)+O(r\tau E/M)+O(V)=O(E)$, reflecting the total size of the cover, the total number of blocker variables that need to be input, and the total size of the output. $\square$

In the proof of Theorem 2, the constraint $r^2\tau^2 \le M$ is tight only in Phase 2, where the $O(E)$ work is much less than the $O(\tau E)$ work in the other phases. In Section 4, we shall exploit this fact.

4 Multigrid Computations

In this section we present the application of our method to multigrid relaxation algorithms. We show that a two-dimensional multigrid graph 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].

The $\sqrt{n}$-by-$\sqrt{n}$ multigrid graph (Figure 2) is a hierarchy of progressively coarser meshes, each mesh connected to the one above and below it.

Lemma 3 *For any $\tau \le \sqrt{M}$, a 2-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 $\ell+1$ and above. Let each subgraph G_i be the home of all vertices in the inner $(k-\tau 2^{\ell+1})$-by-$(k-\tau 2^{\ell+1})$ bottommost submesh of G_i and all the vertices above them. The number of vertices in B_i is

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

and the number of edges in G_i is

$$|E_i| = (2+1/4)|V_i| < \frac{9}{4}\cdot\frac{4}{3}k^2 = 3k^2 ,$$

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

Combining Theorem 2 and Lemma 3, we obtain the following result.

Corollary 4 *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 3, and apply Theorem 2. $\square$

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 2-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]. This 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.

Theorem 5 *A computer with $\Theta(M)$ words of primary memory can perform $T = \Omega(M^{1/5})$ cycles of an elliptic solver on a $\sqrt{n}$-by-$\sqrt{n}$ multigrid graph 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 (τ, r, M)-blocking-cover as described in the proof of Lemma 3, 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 Equation (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 $\frac{3}{10}\lg M + 1$ 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. Phases 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 $r^2\tau^2 \leq M$ from Theorem 2 and replace it by $r^2\tau \leq M$, 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 3 types of subgraphs must fit into primary memory at the same time, the constraint $3r^2\tau \leq M$ also arises. Maximizing τ under the constaints of Lemma 3 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 Equation (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 $O(M)$. $\square$

We mention two extensions of this theorem. The 3-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 $r^2\tau \leq M$. For the 2-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 brief 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 main constraint 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 for a number of other problems (see [1] for examples.)

Under the assumptions of the red-blue pebble game, however, it can be shown that 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. (We omit the details from this extended abstract.) Corollary 4 and Theorem 5 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.

5 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, and thus I/O efficient linear relaxation schemes exist for these classes of graphs. Specifically, we focus on planar graphs to model computations in two dimensions 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 [8, 9].

We begin by defining simplicial graphs formally using definitions from [8].

Definition A **k-dimensional simplex**, or **k-simplex**, is the convex hull of $k+1$ affinely independent points in $\Re^d$ space. 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 $k' \leq k$.

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

Theorem 7 *A computer with $\Theta(M)$ words of primary memory can perform $T \geq \tau$ 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 $\tau \leq M^{\Omega(1/d)}/\lg n$. A precomputation step requires $O(n^2/\tau + nM)$ work I/O'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] and [10]. 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] and [10] that describe how to find good cut covers, and thus, 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 $w : V \cup E \rightarrow \{0, 1\}$, 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 $w : V \cup E \rightarrow \{0, 1\}$ 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. $|C| \leq r$;

CC2. $w(G_1) + w(G_2) \leq (1 + \epsilon) w(G)$;

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

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 $w_1, w_2 : V \cup E \rightarrow \{0, 1\}$ is triplet (C, G_1, G_2) which constitutes a balanced (τ, r, ϵ)-cut-cover for G for both weight functions.

The following theorem relates cut covers to blocking covers.

Theorem 8 *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 one weight function, w_E, that assigns weight 1 to each edge in the graph and weight 0 to each vertex, and a second weight function, w_B, that 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 $(\tau, r, \epsilon = O(1/\lg E))$-cut-cover (B, G_1, G_2) on the current subgraph, G_c, and then recurse on each of G_1 and G_2 where w_B and w_E for G_i is inherited from G_c except that the new w_B assigns 1 to any element of B in G_i.

We 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 as follows. The set of subgraphs forms a τ-cover by the definition of cut-cover. The number of blockers in any subgraph is at most $3r$ by induction. That is, assume at some recursive level the current subgraph G_c contained $3r$ blockers from higher recursive levels, the number of blockers that G_1 or G_2 contains is less than $(2/3)(3r)+|B| \leq 3r$ by the definition of two-color cut-cover. After $\log_{3/2}|E|/M$ recursive levels, the largest subgraph has at most M edges. Finally, the total number of edges in the set of subgraphs is at most $(1+\epsilon)^{\log_{3/2}(|E|/M)}|E| \leq e^{\epsilon \log_{3/2}(|E|/M)}|E| = O(E)$. □

Kaklamanis, Krizanc, and Rao [6] have shown that for every integer ℓ, every n-vertex planar graph has a two-color $(\tau, O(\ell), \tau/\ell)$-cut-cover which can be found in $O(n)$ time. Moreover, Plotkin, Rao, and Smith [10] have recently shown that for every ℓ, every n-vertex d-dimensional simplicial graph of constant aspect ratio has a two-color $O(\tau, O(\ell^{O(d)} \lg n), \tau/\ell)$-cut-cover that can be found in $O(n^2/\ell)$ time.[2] These results can be combined with Theorem 8 to yield the following corollaries.

Corollary 9 *For every $r > 0$, every n-vertex planar graph has a (τ, r, M)-blocking cover, where $\tau = O(r/\lg n)$.* □

Corollary 10 *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 9 and Corollary 10 can be combined with Theorem 2 to prove Theorem 6 and Theorem 7.

6 Further Results

In this section we discuss two additional out-of-core algorithms, one for conjugate gradient computations and the other for implicit linear relaxation on a line, a problem that arises from discretization of one-dimensional parabolic partial differential equations. Both algorithms provide savings in I/O bandwidth, but the conjugate gradient algorithm does so at the expense of performing more work. Such a scheme is useful when the performance of an algorithm is dominated by I/O and the processor is idle most of the time. In such a situation any reduction in the amount of I/O required by the algorithm would improve performance. Since the processor is idle most of the time, the penalty in terms of work does not lead to increased execution time.

6.1 Conjugate Gradient

Conjugate gradient is an iterative method for solving a sparse system of linear equations $Ax = b$. This method proceeds as does a linear relaxation algorithm on the graph underlying the set of equations, except that the relaxation matrix at a time t is "scaled" by a parameter that depends on an inner product of the current state. Thus, the value of a vertex at time t is nonlinearly dependent on the value of every vertex at time $t-1$. This precludes the direct application of the linear simulation algorithm of Section 3.

We can, however, compute the value of this scale factor for each time step t as follows, using just the values of $\langle A^i b \cdot A^j b \rangle$ and the value of the scale factors at earlier time steps. We maintain the current state as a set of symbolic polynomials in y where y^i represents $A^i b$. Thus, the scale factor at time t is the inner product of two symbolic polynomials in y. This scale factor turns out to be a polynomial in which each term has the form $a_i b_j \langle y^i \cdot y^j \rangle = a_i b_j \langle A^i b \cdot A^j b \rangle$ for $i, j < t$, where a_i and b_j are coefficients of the symbolic polynomials and can be computed from the scale factors at previous time steps. Consequently, the scale factor for time t can be numerically evaluated given the values for $\langle A^i b \cdot A^j b \rangle$ and the values of the scale factor at previous time steps. The coefficients of the symbolic polynomials for time t can, in turn, be computed and used to numerically compute the scale factor for time $t+1$, and so on.

Once the symbolic polynomials and the scale factors have been computed up to some time τ, we can evaluate them by computing (but not necessarily storing) $A^t b$ for each $t < \tau$ to find the state at time τ. We can use the techniques of Section 3 to do this in an I/O efficient manner. Also, the initial computation of $\langle A^i b \cdot A^j b \rangle$ for $i, j \leq \tau$ can be performed using techniques from Section 3. This initial computation, however, requires a factor of τ more inner-product computations than would be required in the naive implementation of the conjugate gradient method.

In the full version of the paper, we will present the technique sketched above in detail. Here, we simply

[2] In fact, they can find cut-covers in any graph that excludes 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.

state our results.

Theorem 11 *Given a graph $G = (V, E)$ with a (τ, r, m)-blocking-cover such that $\tau m + r^2\tau^2 \leq M$, a computer with $O(M)$ words of primary memory can perform $T \geq \tau$ iterations of a conjugate gradient relaxation on G using $O(TE + \tau^3 + \tau TV)$ work and $O(TE/\tau + \tau TE/m)$ I/O's. A precomputation phase requires $O(r\tau E)$ work and $O(E)$ I/O's.* □

The following corollary gives the performance of our conjugate gradient algorithm on a 2-dimensional mesh.

Corollary 12 *A computer with $\Theta(M)$ words of primary memory can perform $T \geq \tau$ iterations of a conjugate gradient relaxation on a $\sqrt{n}$-by-$\sqrt{n}$ grid using $O(\tau nT)$ work and $O(nT/\tau)$ I/O's, where $\tau \leq M^{1/3}$. No precomputation is required.* □

6.2 Implicit Relaxation

Discretization schemes for one-dimensional parabolic partial differential equations often give rise to iterative solutions of the form

$$Ax^{(t)} = x^{(t-1)} + b^{(t)} , \tag{5}$$

where A is a tridiagonal matrix and $b^{(t)}$, which represents the boundary condition, has only two nonzero entries. This equation is implicit, since A multiplies $x^{(t)}$ rather than $x^{(t-1)}$, and hence, one must solve a set of linear equations to find the state $x^{(t)}$ from the state $x^{(t-1)}$. Thus, any given point in $x^{(t)}$ may be influenced by all the points in $x^{(t-1)}$. By partitioning the domain (which is a line) into intervals and using blockers between intervals, we obtain the following result, whose proof is omitted .

Theorem 13 *A computer with $\Theta(M)$ words of primary memory can perform $T \geq \tau$ steps of Equation (5) using $O(TV)$ work and $O(TV/\tau)$ I/O's, where $\tau \leq M^{1/3}$. A precomputation phase requires $O(\tau^3 M)$ work and $O(\tau M)$ I/O's.* □

7 Discussion

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 2-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].

An important issue that we do not fully address in this extended abstract is the issue of data clustering. In most computer systems, a single word cannot be transferred between the primary to secondary memories, but rather a block of words, such as a disk page or cache line, must be transferred. In the full version of the paper we plan to describe clustering schemes that ensure that the clustering of data items does not lead to degraded performances of any of our algorithms.

We also plan to study the effect of roundoff errors on the numerical stability of our algorithms, as well as implement them.

References

[1] A. Aggarwal and J. S. Vitter. The input/ouput complexity of sorting and related problems. *Communications of the ACM*, 31(9), 1988.

[2] B. Awerbuch and D. Peleg. Sparse partitions. In *31st Symposium on Foundations of Computer Science*, pages 503–513, 1990.

[3] W.L. Briggs. *A Multigrid Tutorial.* SIAM, Philadelphia,PA, 1987.

[4] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* MIT Press and McGraw-Hill, New York, 1990.

[5] J.-W. Hong and H.T. Kung. I/O complexity: the red-blue pebble game. In *Proceedings of the 13th Annual ACM Symposium on Theory of Computing*, pages 326–333, 1981.

[6] C. Kaklamanis, D. Krizanc, and S. Rao. New graph decompositions and fast emulations in hypercubes and butterflies. In *Proceedings of the 5th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 325–334, June 1993.

[7] R. Koch, T. Leighton, B. Maggs, S. Rao, and A. Rosenberg. Work-preserving emulations of fixed-connection networks. In *Proceedings of the 21st Annual ACM Symposium on Theory of Computing*, pages 227–240, May 1989.

[8] G. Miller and W. Thurston. Separators in two and three dimensions. In *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing*, pages 300–309, May 1990.

[9] G.L. Miller, S.-H. Teng, and S.A. Vavasis. A unified geometric approach to graph separators. In *Proceedings of the 32nd Annual Symposium on Foundations of Computer Science*, pages 538–547, 1991.

[10] S. Plotkin, S. Rao, and W. Smith. Improved decompositions for graphs with excluded minors., 1993. Unpublished manuscript.