UCRL-ID-153078

# Efficiency Issues in Parallel Coarsening Schemes

*K. A. Gallivan and U. M. Yang*

U.S. Department of Energy

Lawrence Livermore National Laboratory

**May 2, 2003**

# Efficiency Issues in Parallel Coarsening Schemes [*]

Kyle Gallivan [†] Ulrike Meier Yang [‡]

May 2, 2003

**Abstract**

Various options for sequential, shared memory and distributed memory implementations for the CLJP algorithm, a parallel coarsening scheme within algebraic multigrid, are discussed. The use of different data structures as well as different approaches of implementating the actual algorithm are investigated, and experimental results illustrating the results are presented.

## 1 Introduction

With the advent of large high performance computers with large number of processors, it has become necessary to design parallel algorithms of all sorts. Particular emphasis has been placed on the development of scalable algorithms, such as multigrid methods. With this in mind, the parallelization of algebraic multigrid, a method that can be applied to a linear system, $Ax = b$, without additional knowledge, such as the underlying finite elements or a grid, has become very important. AMG proceeds by determining a subset of the original degree of freedoms through a coarsening algorithm, an interpolation operator that transfers from the coarse space to the fine space, a restriction operator that transfers from the fine space to the coarse space. Among those procedures, the classical coarsening scheme is highly sequential, it was therefore necessary to develop a parallel scheme.

In this paper, we examine the implementation of the CLJP algorithm. Various options for sequential, shared memory, and distributed memory implementations are discussed. The major characteristic determining the form of the algorithm is the form of the data structure assumed. We consider three different versions, one that uses row oriented storage, denoted 'CSR', one with column oriented storage, denoted 'CSC', and a modified version, denoted 'mod', that assumes that both forms are available. The 'CSC' scheme is equivalent to using the transpose of the strength matrix. There are some interesting differences in implementing this algorithm depending on which

[*]This work was performed under the auspices of the U.S. Department of Energy University of California Lawrence Livermore National Laboratory under contract No. W-7405-Eng-48.

[†]School of Computational Science and Information Technology, Florida State University, 476 Dirac Science Library, Tallahassee, FL 32306-4120

[‡]Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, Box 808, L-560, Livermore, CA 94551

matrices are available. We investigate how the different storage schemes influence the implementation and performance of the algorithm and consider the tradebacks. The high variations in performance for the different implementations show the importance of making the right decision on what storage scheme to choose.

## 2 The CLJP Algorithm

The Cleary-Luby-Jones-Plassman (CLJP) parallel coarsening algorithm was proposed by Cleary [1, 2], and is based on parallel graph partitioning algorithms introduced by Luby [4] and developed by Jones and Plassman [3].

The goal of the algorithm is to partition the degree of freedoms of the nonsymmetric matrix of the linear system $Ax = b$ to be solved into coarse ($C$) and fine ($F$) points. To select the $C$-points, we seek those unknowns $x_i$ that can be used to represent the values of nearby unknowns $x_j$. This gives rise to the concepts of *dependence* and *influence*. A point $i$ depends on the point $j$ if the value of the unknown $x_j$ is important in determining the value of $x_i$ from the $i$-th equation, i.e. $j$ influences $i$.

We define $S$, the auxiliary *influence matrix*:

$$S_{kl} = \begin{cases} 1 & \text{if } l \in S_k, \\ 0 & \text{otherwise.} \end{cases} \tag{1}$$

That is, $S_{kl} = 1$ only if $k$ depends on $l$. The $k$th row of $S$ gives $S_k$, the set of dependencies of $k$, while the $k$th column of $S$ gives $S_k^T$, the set of influences of $k$. We can then form the directed graph of $S$, and observe that a directed edge from vertex $k$ to vertex $l$, $(k, l)$, exists only if $S_{kl} \neq 0$. The $k$-th row $S_k$ contains the indices of the nodes that are destinations of outgoing edges from node $k$. The $k$-th column $S_k^T$ contains the indices of the nodes at the sources of incoming edges to node $k$.

To each point $k$ we define a measure $\mu(k) = |S_k^T| + \sigma(k)$, the number of incoming edges of point $k$ or the count of nonzeros in the corresponding column of $S$ plus a random number in $(0, 1)$. The random number is used as a mechanism for breaking ties between points with the same number of influences, thus allowing the definition of local maxima in the graph. We then select the set $C$ of local maxima, i.e. the set that consists of the points $k$, for which $\mu(k) > \mu(l)$ for all $l \in S_k \cap S_k^T$ (all points that either influence or depend on $k$). By construction, this set will be independent.

An example of the weight definition and coarse point selection is given in Figure 1.

Once the independent set $C$ is chosen, we modify the graph according to the following pair of heuristics, which are designed to ensure the quality of the coarse-grid while controlling its size.

**H1:** Values at $C$-points are not interpolated; hence, neighbors that influence a $C$-point are less valuable as potential $C$-points themselves.

**H2:** If $k$ and $l$ both depend on $j$, a given $C$-point, and $l$ influences $k$, then $l$ is less valuable as a potential $C$-point, since $k$ can be interpolated from $j$.

The algorithm now proceeds by defining a sequence of sets of coarse nodes and updating the graph in response to each of these sets. After each update, each node of the original graph is in one of three states: coarse, fine, or undetermined. The key

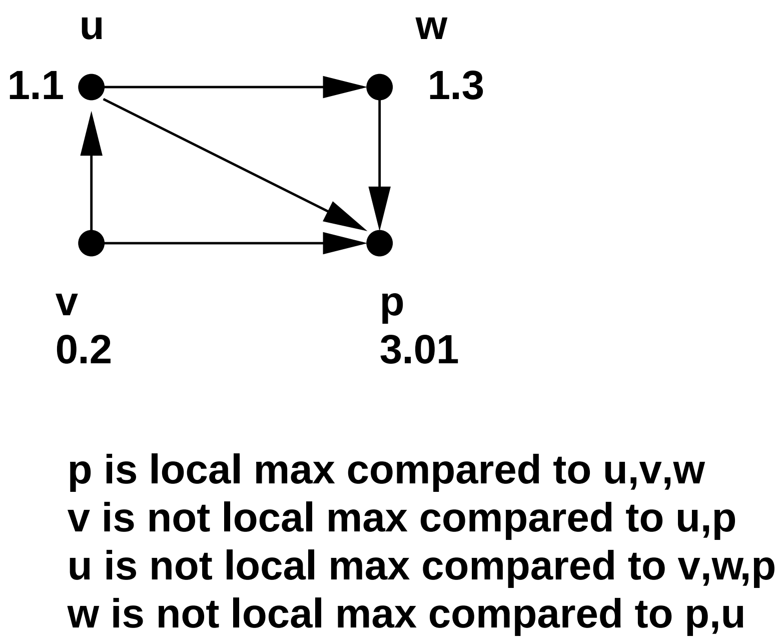
u
w
1.1
1.3
v
0.2
p
3.01
p is local max compared to u,v,w
v is not local max compared to u,p
u is not local max compared to v,w,p
w is not local max compared to p,u

Figure 1: Weights and Coarse Point Selection

components of the algorithm are: the selection of the next set of coarse nodes from the current set of undetermined nodes; the update of the edges associated with the current set of undetermined nodes to the set associated with the nodes that remain undetermined after removal of the next set of coarse nodes; and the selection of the fine nodes after the updates have been made.

The graph, $G_i$, is defined by the undetermined nodes, $V_i$, and remaining edges $E_i$ after the removal of edges and coarse nodes in sets $C_j$, $j < i$ and the induced fine nodes $F_j$, $j < i$. For example, $G_1$ is the original graph, $C_1 \subset V_1$, is the initial coarse set, $V_2 = V_1 - C_1 - F_1$, and $E_2 \subset E_1$ are the sets of nodes and edges respectively that follow from the application of the update rules to $G_1$ given $C_1$.

The heuristics **H1** and **H2** are implemented as follows. The edges in $E_i$ are updated based on the selection of $C_i$ by removing all edges, $e = (p, v)$, that satisfy one of the following:

- $p \in C_i$ (the edge's source is a coarse point)
- $v \in C_i$ (the edge's destination is a coarse point)
- $\exists (p, u) \in E_i, (v, u) \in E_i$ such that $u \in C_i$ (the edge connects two nodes that are the sources for edges with a common coarse point destination)

As the edges are removed, the weights associated with the destination node of each edge must be decremented in preparation for the determination of $C_{i+1}$.

An example of the edges removed from a graph is given in Figure 2. The edges that are removed are slashed. Coarse points are labeled with a $C$. Edges $(w, p)$ and $(p, x)$ are the only ones retained.

After removal of the edges and the update of the weights $F_i$ is taken to be all nodes with an updated weight less than 1. This implies that all incoming edges to the node have been removed. Note that this does not imply that all outgoing edges have been removed and care must be taken to have any destination nodes of such edges processed

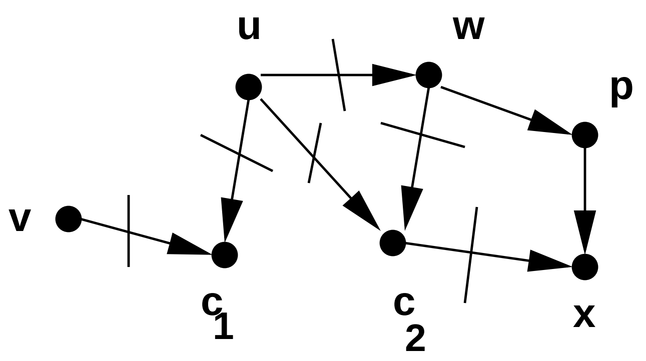

Figure 2: Edge updates

correctly and eventually become coarse nodes. The approach to guarantee this depends on the particular implementation and the data structures used.

## 3 Implementation

There are basically three ways of implementing the algorithm, The first one assumes a row-oriented storage scheme for the influence matrix and is denoted 'CSR'. The second one, denoted 'CSC', assumes a column-oriented storage scheme and is equivalent to using the $S^T$ or the dependence matrix with a row-oriented storage scheme. The third implementation, which we call modified implementation or 'mod', assumes that both $S$ and $S^T$ are available and can be used in the coarse grid selection process.

The choice of the storage scheme has a significant influence on the implementation. The algorithm consists basically of two passes, one of which is fairly similar in each of the three implementations. The second pass is however significantly different. In the following paragraph, the first pass is described in generic set-like pseudocode, followed by descriptions of the second pass for each implementation in the subsequent subsections. Implementation details are ignored until later sections, which consider sequential, distributed memory and shared memory implementations.

### 3.1 Pass 1 (Removal of $C$- and $F$-Points and Determination of Local Maxima

The first pass through $G_i$ in this version of the algorithm is used to remove the coarse nodes $C_i$ from $G_i$. (Of course, since $S_n$ is not needed again once a node is determined to be coarse it is not really necessary to remove the edges in $S_n$, i.e., the whole row is removed from consideration when $n$ is removed from $G_i$.) It is also used to accomplish two other important tasks: the identification of $F_i$ and the examination of the weights to determine new local maxima and therefore $C_{i+1}$. The weights have been completely updated at the end of the first pass but in order to use a simple row-oriented test to identify the local maxima some care must be taken in the implementation to avoid the need for an extra pass through the vector that is used to store the status of the node, i.e., fine, coarse, undetermined. This is discussed in Section 4. Note that the code assumes that a node currently in $G_i$ is going to be coarse in $G_{i+1}$ and the tests below mark it (possibly multiple times) as noncoarse. As a result, it is only at the end of this pass that any node is known to be in $C_{i+1}$ or not. The operations can be organized to consider each node and elements of its row once.

The only potential complication is the handling of a degenerate case of a fine node. Since a node becomes fine as soon as its weight becomes less than 1, i.e., its column count is 0, it is possible to have a fine point that still has a nonzero row count indicating the presence of outgoing edges from the node. The difficulty arises not with the fine node but with the nodes to which it points via the remaining edges in the row. This destination node must become coarse. If however the fine node is removed and the weight of the destination decremented, the destination may become a fine node. As a result, for the CSR version, a node is made fine, when its weight drops below 1 and all of its row edges have been removed, which can easily be verified by checking $S_f$. For the column-oriented version and possibly the modified implementation, a two-step approach is taken, where an $F$-point is marked as 'potentially fine' and kept in $G_i$, and if its status hasn't been altered to nondetermined after the next sweep through the algorithm, it is marked 'fine' and removed from $G_{i+1}$. The modified version has both the row- and the column-oriented structure and it might therefore be possible to find whether there are still outgoing edges from a potentially fine point by investigating $S_f$. It is however fairly expensive to keep both $S$ and $S^T$ up to date with regard to removed edges and $S^T$ is more likely to be the 'working' structure (see Section 3.4). Therefore it is recommended to use the same approach as used for the CSC implementation.

Nodes that have a weight below 1 with a nonzero row count are referred to as delayed fine nodes. To see that the nodes in $S_f$ where $f$ is a delayed fine node must eventually become coarse and $f$ converted to fine status, consider the following cases.

First, if, as expected, all of the remaining nodes in $S_f$ are marked coarse, then the row count of $f$ must reach 0 and its weight, $\mu(f)$, is still below 1 indicating that it will be marked fine and removed from the graph. Second, a node $u \in S_f$ cannot be marked fine since the edge $(f, u)$ remains in the graph until $f$ is removed and therefore $\mu(u) > 1$ until then as well. Third, if $u \in S_f$ is not made coarse by becoming a local maxima during the normal processing of nodes it must reach a state where the only nodes left defining its column are delayed fine nodes, i.e., $u \in S_v \rightarrow v$ is a delayed fine node. As a result, $\mu(u) > 1$ makes it a local maxima, and it is marked coarse.

The code for Pass 1 is as follows:

```
for each node $n \in G_i$
        if $n \in C_i$ then (the node is coarse)
            (1) remove all edges in $S_n$
            remove $n$ from $G_i$
        else if $\mu(n) < 1$ and $\|S_n\| = 0$ then (the node is fine)
            add node to $F_i$ and remove it from $G_i$
        else (node stays for $G_{i+1}$)
            (2) remove all marked edges from $S_n$
            (3) for $v \in S_n$ (all unmarked edges)
                if $\mu(n) > \mu(v)$ then $v \notin C_{i+1}$
                if $\mu(n) < \mu(v)$ then $n \notin C_{i+1}$
            (3) end for
        end if
end for
```

Note that for the CSC and the modified implementation $S_n$ in (1), (2) and (3) need to be replaced by $S_n^T$.

## 3.2 CSR implementation

In this section, the second pass of the algorithm is described in the CSR implementation. The row-oriented storage is exploited in that each node, $v$, has an associated structure $S_v$ containing the nodes that define edges with source $v$, i.e., $p \in S_v \rightarrow (v, p) \in E_i$.

Assuming that the weights have been assigned to all nodes in $V_1$ and $C_1$ has been constructed based on local maxima, the algorithm consists of a loop over $G_i$ that marks edges for removal based on their relationships to points in $C_i$. Each iteration of loop requires the weights and status information associated with each node in $G_i$.

```
for each $n \in G_i$
        if $n \in C_i$ (the node is coarse)
             (4) for each $v \in S_n$
                    mark edge $(n, v)$ for removal
                    decrement $\mu(v)$ by 1
             (4) end for
        else (the node is undetermined)
             (5) for each $v \in S_n \cap C_i$
                    mark edge $(n, v)$ for removal
                    decrement $\mu(v)$ by 1
             (5) end for
             (6) for each $v \in S_n$ such that
                    $(n, v)$ is not marked for removal (noncoarse)
                    if $(S_n \cap C_i) \cap (S_v \cap C_i) \neq \emptyset$ then
                        mark edge $(n, v)$ for removal
                        decrement $\mu(v)$ by 1
                    end if
             (6) end for
        end if
end for
```

The coarse nodes in $C_i$ are processed by loop (4). Edges whose source is the coarse node, e.g., edges like $(c, x)$ in Figure 2, are removed. Since the edge affects the column count of the destination node, the weight of $v$, $\mu(v)$, must be decremented.

The nodes in $G_i$ that are not coarse are processed by loops (5) and (6). The processing can be organized in several ways and is complicated by the fact that edges connecting a noncoarse node to a particular coarse node via paths of lengths 1 and 2 must be identified. In this case (5) identifies the paths of length 1 from a noncoarse node to a coarse node, effectively removing edges to nodes in the intersection of $S_n$ and $C_i$. Loop (6) can exploit the fact that this intersection has already been identified in loop (5). The first edge in a path of length 2 from $n$ to a coarse point $c$ for which $(n, c)$ also exists is identified in loop (6). The edge $(u, w)$ in Figure 2 is an example of this type. Only one node need be found in the intersection of the coarse points connected to $n$ and the coarse points connected to $v$ by outgoing edges in order to remove edge

$(n, v)$. Note that only noncoarse nodes in $S_n$ are considered since marked nodes at this point correspond to edges that are connected to coarse nodes, i.e., marked in loop (5).

This portion of the code is the source of the requirement of using the operation "mark edge for removal" rather than removing an edge when it is deemed necessary to do so. Consider the nodes $u$, $w$ and the coarse node $c_2$, to which they are both connected in Figure 2. Loop (5) could remove the edge $(w, c_2)$ from $G_i$. However, if this were done, i.e., node $w$ was processed in (5) before node $u$, the information required to determine that edge $(u, w)$ must be removed would not be available when $S_w$ was examined during the processing of $S_u$. As a result, edges (or more precisely their destination nodes stored in the row data structure) are marked for removal by a second pass through $G_i$. On the second pass, described below, the state of the node/edge is changed to "removed". The removed state is needed in order to avoid considering the edge repeatedly and for no purpose in later iterations. The maintenance of the three states and the isolation of the nodes/edges that have been removed from those that are still undetermined influences the implementation and the effectiveness of particular data structure choices. Note a coarse node and its row can be removed immediately after it is identified, since it does not affect the situation above. In the version described here, the removal of coarse nodes and their rows are delayed until the second pass handling $G_i$. Exactly when in a particular algorithm this is possible depends on the method of identifying $C_i$. In the algorithm presented here, $C_i$ is not known until the end of the second pass that processes $G_{i-1}$. So the earliest $C_i$ can be removed is during the first pass handling $G_i$.

## 3.3 CSC implementation

CLJP coarsening using column-oriented structure is in some respects much simpler than the row-oriented implementation, and it has more promise for an efficient parallel implementation. An initial sequential version is presented here. A modified version and comments on the use of some row information to improve efficiency are discussed in Section 3.4.

Once again assume that the algorithm will update $G_i$ given $C_i$ by passing through all active nodes and edges in $G_i$. Each node may be coarse or not, and this distinction is once again used to define the algorithm.

Consider first the case where the node is not coarse, e.g., node $x$ in Figure 2. The edges $(c_2, x)$ and $(p, x)$ are indicated by the node data in $S_x^T$. Clearly, the edge $(c_2, x)$ can be removed. The decision whether edge $(p, x)$ should be removed requires the more complicated processing of determining whether or not $p$ and $x$ are connected to a common coarse point. In this version of the algorithm this decision is made when visiting the common coarse point. So for noncoarse nodes, this version of the algorithm only removes incoming edges, whose sources are coarse nodes. Marking for later removal is not required, since the edge will play no role in processing other nodes in $G_i$.

Coarse nodes perform the bulk of the edge removal in this form of the algorithm. The goal is to remove edges pointing to the coarse node, e.g., $(u, c_2)$ in Figure 2 by removing the coarse node and its column from $G_i$ and edges between nodes that each have edges to the coarse node, e.g., $(u, w)$ in Figure 2. This can be done quite simply in terms of the column structures, and the edges can be marked for removal in the noncoarse node columns and then removed upon completion of the processing of the individual coarse node and not in a second pass through $G_i$.

Essentially, each noncoarse node, $v$, that is contained in $S_n^T$ for a coarse node $n$ must be considered. Edges such as $(u, w)$ in Figure 2 are identified and used to update, not $S_n^T$, but the $S_v^T$ where $v \in S_n^T$. In the case of Figure 2, when processing $S_{c_2}^T$, the information in $S_w^T$ must be examined and updated since $w \in S_{c_2}^T$. The edge $(u, w)$ must be removed from $G_i$ and therefore $u$ must be removed from $S_w^T$. Since $u \in S_{c_2}^T$, the node is easily identified as a member of $S_{c_2}^T \cap S_w^T$ and all nodes in that set must be removed from $S_w^T$ (in this case only $u$).

```
for each $n \in G_i$
        if $n \notin C_i$ then (the node is not coarse)
            for each $v \in S_n^T$
                if $v \in C_i$ then
                    remove edge $(v, n)$ and decrement $\mu(n)$
                end if
            end for
        else (node is coarse)
            for each $v \in S_n^T$
                Compute $S = S_n^T \cap S_v^T$
                Remove edges $(p, v)$ from $S_v^T$ where $p \in S$
                Decrement $\mu(v)$ by $\|S\|$
            end for
            (7) (*optional:* Remove $n$ and $S_n^T$ from $G_i$)
        end if
end for
```

Note that it is possible to remove coarse nodes from $G_i$ in this pass, as indicated by statement (7). If this option is chosen, $C$-nodes do not need to be eliminated from $G_i$ in Pass 1. Numerical experiments indicate that this option does not improve performance. Note that in this version of the algorithm, coarse and noncoarse processing is intermingled. As a result, when processing a coarse node $c$, $S_c^T$ may contain a mix of noncoarse nodes that have already been processed and those that have not been processed. This does not affect the correctness of the algorithm. It only effects when edges from a coarse node to a noncoarse node may be removed, if the coarse node and noncoarse node have a common coarse neighbor. This can be seen in Figure 3. The edge $(c_1, v)$ may be removed when processing the noncoarse node $v$ or the coarse node $c_2$.

## 3.4 Modified Implementation

The previous algorithm using column storage has as its main processing step the update of undetermined nodes that are within the column of a coarse node. In this step, a fan-out approach is taken, i.e., for each coarse node all of the columns of the undetermined nodes are updated based on the column of the current coarse node. While this form is appealing due to its simplicity, it has some potential problems with shared and distributed memory parallelism due to synchronization and communication.

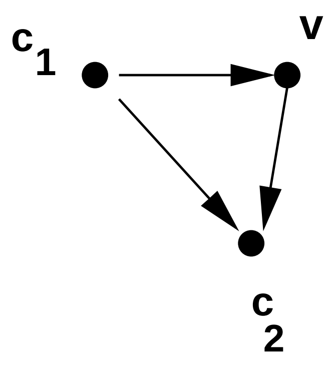

Figure 3: Column form nondeterministic update

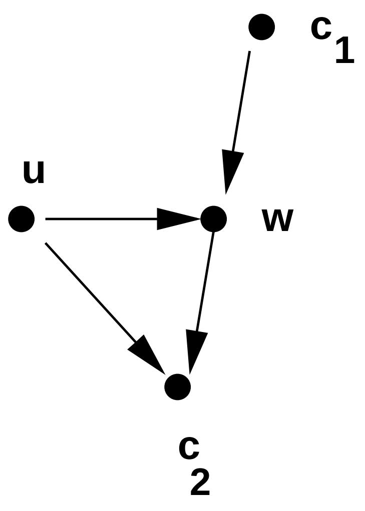

Figure 4: Modified form updates

The algorithm can be reorganized to concentrate the processing on undetermined nodes by merging the first loop where incoming edges with coarse sources are removed with an altered form of the second loop that updates each undetermined node based on several coarse nodes.

In order to see the reorganization, consider node $w$ in Figure 4. The edge from $c_1$ to $w$ is contained in $S_w^T$ and all such edges are removed by $S_w^T - S_w^T \cap C_i$. The value of $\mu(w)$ should be updated accordingly. The edges from $w$ to $c_2$ and $u$ to $c_2$ are removed when the coarse node $c_2$ is removed from the graph. This does not affect $\mu(w)$. These two actions remove all edges involving $w$ and a coarse node.

The edge $(u, w)$ is representative of those that are in a path of length two from $u$ to a coarse node also reachable from $u$ by a single edge. A characterization identifying coarse nodes such as $c_2$ and undetermined nodes $u$ is needed to complete the modified algorithm.

The set of coarse nodes like $c_2$ is easily characterized as $T_w = \left\{c \in C_i | w \in S_c^T\right\}$. However, we need to relate this directly to a data structure element of $w$. Clearly we have the following $T_w = S_w \cap C_i$. If the row-oriented data structure is not available, $S_w$ must be constructed for each $w$ by a simple process of scanning all nodes in the columns associated with each node in $C_i$ and creating a list of lists. The list associated with each node $w$ is the desired set of coarse points. Nodes like $u$ are contained in $S_w^T \cap S_c^T$ where $c \in T_w$.

If $S_w$is available, then its updates are trivially done at the same time as those of $S_w^T$. Keeping the column and row data structures consistent is only important relative to the coarse nodes that are selected on each step. The extra overhead in computing the intersection of the $S_w$ and $C_i$ on each step given that some unnecessary edges to undetermined or coarse nodes may still be in $S_w$ should be insignificant compared to the cost of keeping all row and column data structures consistent. Also note this may have implications as to how the coarse nodes are represented. It may be useful to not only be able to distinguish coarse nodes from noncoarse nodes but also to distinguish coarse nodes in $C_i$ from coarse nodes in $C_j$ where $i \neq j$.

```
for each node $n \in G_i - C_i$
        $T_n = S_n \cap C_i$
        $R_n = S_n^T \cap C_i$
        $S_n \leftarrow S_n - T_n$
        $S_n^T \leftarrow S_n^T - R_n$
        Decrement $\mu(n)$ by $\|R_n\|$
        for each coarse node $c \in T_n$
             $S_n^T \leftarrow S_n^T - S_n^T \cap S_c^T$
             Decrement $\mu(n)$ by $\|S_n^T \cap S_c^T\|$
        end for
end for
```

## 4 Implementation Details

There are several ways to implement each of the versions above based on decisions concerning supporting data structures, parallelism, and architectural issues. In this section, some basic approaches to these support issues are discussed. We assume that the edge information represented in row and/or column data structures cannot be destroyed. So we must use a procedure that allows an index to have "removed" status. This also implies that at some point in the algorithm removed nodes must have their row and/or column data structures restored, i.e., the indices must be replaced so as to indicate presence in the graph.

### 4.1 Handling Vertices and Edges

For all of the versions above, membership of a node in $G_i$ is maintained. This membership must also allow easy association of the nodes in $G_i$ with a loop schedule that may or may not be parallel. The easiest way to accomplish this is by using an indirection vector, *graph_array*$(1 : N)$, that is initialized to $1 : N$. A loop over all nodes in $G_i$ is then simply a loop from 1 to some position that contains the last index of a node still in $G_i$. Removal from $G_i$ corresponds to moving an index of a node from the region of the vector that contains nodes in $G_i$ to one that contains those that have been marked coarse or fine. As long as no ordering is assumed on the nodes in $G_i$ removal can be accomplished in $O(1)$ time by swapping the index to be removed with the last index in $G_i$ thereby extending the removed region of the vector by 1 and decrementing the position of the last retained node by 1. The structure of the vector can be generalized to keep $C_i$ and $F_i$ nodes together as well. This saving of coarse and fine indices assumes that some information about the coarse or fine nodes must be kept beyond whether or not they are coarse or fine. In most cases, this will not be needed, since a second vector $cf(1:N)$ that contains at least binary information concerning coarse and fine status is also assumed. How to use $cf$ is discussed in more detail below.

Membership in $G_i$ also implies manipulation of the status of edges contained in the strength matrix. In the case of the algorithms discussed above, two or three states may be needed for an edge: removed, active, and possibly removed and connected to a C-point.

The information that two nodes are connected needs to be kept, even if the edge has been removed, in order to recognize that a node is connected to a point that will at a later time become a C-point. Consequently, an edge should never be actually removed, but only marked as removed. This can be accomplished by setting active edge indices positive and removed edges negative.

Since in part of the algorithm only edges in active states are processed, it is more efficent to keep the row information so as to access these nodes only. This is easily done by maintaining a separation among the two main states of active and removed. A way that uses only an extra integer per point would be to swap the elements within the data structures. Different groupings are necessary here for the three different implementations mentioned in the previous section.

For the CSR implementation, three types of edges need to be distingushed, active, removed and removed connections to C-points. The edges should be grouped in each row in the following way:

| | | |
|---|---|---|
| removed | active | removed connections to C-points |

This implementation requires two extra integers per row, which act as pointers to where each of the states starts, and which are changed, whenever points are swapped out of one group into another. This allows for much more efficient computation, since loops can be shortened in part of the algorithms, and edges that have been removed do not have to be touched. It also clearly marks where to find removed connections to C-points.

For the CSC implementation, in which edges of non-coarse neighbors with a common C-point are removed from view of the C-point, only two states need to be distingushed, removed and active.

| | |
|---|---|
| removed | active |

Now only one integer pointer per row of $S^T$ is needed.

Finally, in the modified verion, where both $S$ and $S^T$ are available, we need a combination of the above approach. For $S^T$, the column structure, three types of edges should be distinguished, active, removed and removed connections to C-points; for $S$, the row structure, only two states are necessary, active and removed, leading to three additional integers per point.

The use of swapping elements can lead to significant savings in time, as our experiments in Section 4.4 show.

If the active nodes are to remain ordered within the column or row associated with each, then extra care must be taken with the removal of marked edges. The easiest way to do this is to compress as shown in the suggested implementation but rather than swapping marked edges to the beginning of the removed edge portion immediately, copy the indices to a temporary vector for later insertion. This allows compression of the kept edges to be done simply while scanning the nodes. More complicated but less storage intensive ways are also possible that involve maintaining multiple pointers during the scan. Note, however, that if the input data structure that is given by the user is not changeable then duplication or scans must be used to find active nodes in each row or column's data structure portion.

## 4.2 Status Information

All of the algorithms discussed above require some sort of status vector, here referred to as the $cf$ vector, to store global information about coarse and fine status as well as temporary information that aids in the computation of certain set intersections. For example, it could be defined that during the processing of $G_j$, $cf(i)$ is 0 if node $i$ is fine, $j \geq k > 0$ if node $i$ is a member of $C_k$, $k+1$ if node $i$ is undetermined, and temporarily negative during the computation of intersections.

While this definition is more complicated than a typical 0 if fine, 1 if coarse, negative for intersections, and positive for undetermined, it has the benefit of avoiding an extra pass through the undetermined nodes of $G_i$ in order to reinitialize the values of $cf$ that are updated during the determination of the next set of coarse points. It also provides information in $cf$ about the set of coarse nodes of which a particular node is a member.

The following algorithm demonstrates that two passes per $G_i$ is sufficient. Initially, the nodes in $G_1$ all have their positions in $cf$ set to 1, implying that they are all viewed as members of $C_1$. The checks on the weights are performed and each node, $j$, that is in fact not coarse is identified and its value $cf(j)$ is changed. If $cf(j)$ was set to 0 a reinitialization pass would be needed to reset $cf(j)$ to 1 for $i \in G_i - C_i$. However, if the value of $cf(j)$ is incremented, i.e., to 2 then the values indicate a noncoarse node and are initialized properly for the identification of $C_2$. Nodes that are in $C_2$ will have $cf(i) = 2$ and those that are not will have $cf(i)$ incremented to 3. Therefore the reinitialization pass is avoided, and $cf(j) = i$ when $j \in C_i$.

## 4.3 Intersections

The main issue of performance for the sequential versions of the algorithms is the implementation of the necessary set intersections in the most efficient fashion. Depending on the particular intersection, the implementation may vary.

The $cf$ vector is easily used to determine the intersection of a row or column with $C_i$ by checking the value of $cf(j)$, where $j \in S_n$. If $cf(j) = i$, then $j \in C_i$. (Effectively the intersection is really a membership test in $C_i$, since the algorithm is typically scanning all members of a row or column.)

To determine the intersection, of say, two rows, $S_n$ and $S_v$, with or without an additional intersection of $C_i$, the $cf$ vector can also be used. The values of $cf(j)$ where $j$ is in $S_n$ or possibly a coarse node in $S_n \cap C_i$ are temporarily set to $-i$. Common coarse nodes that are the destination of edges from $n$ and $v \in S_n$ can be found by checking $cf(j)$ for $j \in S_v$. Note that in this implementation, $S_n \cap C_i$ is marked in $cf$

once and all $S_v$ for $v \in S_n$ are processed before the $cf(j) = -i$ ($j \in S_n \cap C_i$) must be reset to $i$. A similar procedure can also be used for a scattered $S_n$ or $S_n^T$. Note the key in the scattered intersection processing is the ability to clear the original scattered information efficiently. If the row or column scattered is updated by the loop in which the intersection is computed then the original scattered indices must be saved or it must be possible to reconstruct them, or it must be known that the algorithm has reset them appropriately as a side effect of processing in the loop.

It is also possible to process the scattered vector so as to remove elements while scattered and then process them by either marking them for removal, reordering them, and later removing them, or by immediate removing them. Suppose $S_n^T$ is to have elements removed based on a computation that indicates which element of $S_n^T$ is to be removed, but that is not indexed by a loop through $S_n^T$. As a result, removing an element $j \in S_n^T$ might require a search through $S_n^T$ to find it and mark or remove it. If, however, the indices of $S_n^T$ have been scattered and marked in the $cf$ vector, then there are effectively two copies of $S_n^T$. The compressed version can be kept and used to drive the gather loop. The scattered version can be used to indicate membership in $S_n^T$, via a negative value of a $cf$ entry, and to have the element removed by resetting the $cf$ entry to its original positive value. This can be done, of course, without knowing the position of the index for the element to be removed in the compressed $S_n^T$ structure. When the elements are gathered back using $S_n^T$, the mismatch between the index in $S_n^T$ and the entry in $cf$ having been reset to a positive value can be used to mark or remove the element in $S_n^T$.

If a work vector other than $cf$ is available, then it is possible to perform the update for several intersections without reinitializing by clearing. One need only initialize the work vector for each coarse set iteration. This can be done when processing the coarse nodes in the second pass that removes nodes in $C_i$, i.e., you can reset the corresponding element of the work vector. The processing associated with each element of the $c \in S_n$ is simplified to a write only and when gathering for each $n$, one does not have to update the work vector. The result is a very efficient computation of a series of unions, elements of columns in $S_n$, and the intersections associated with each at the cost of an extra integer work vector.

```
for each $n \in G_i - C_i$
        for each $c \in S_n$
            for $j \in S_c^T$
                $work(j) = n$
            end for
        end for
        (at this point the union of the $S_c^T, c \in S_n$
        have $n$ in the associated $work$ locations)
        for $j \in S_n^T$
            if $work(j) = n$ then
                remove $j$ from $S_n^T$
            else
                keep $j \in S_n^T$
            end if
        end for
end for
```

| matrix | CSR | CSR, swap | CSC | CSC, swap | mod | mod, swap | transpose |
|---|---|---|---|---|---|---|---|
| 1 | 0.51 | 0.41 | 0.38 | 0.38 | 0.47 | 0.41 | 0.03 |
| 2 | 4.10 | 2.53 | 1.36 | 1.17 | 1.89 | 1.50 | 0.11 |
| 3 | 0.34 | 0.24 | 0.10 | 0.08 | 0.14 | 0.12 | 0.01 |

Table 1: Times in seconds for various test problems using different implementations

| matrix | CSR | CSR, swap | CSC | CSC, swap | mod | mod, swap | transpose |
|---|---|---|---|---|---|---|---|
| 1 | 1.54 | 1.37 | 1.20 | 1.13 | 1.51 | 1.36 | 0.15 |
| 2 | 10.43 | 7.97 | 4.32 | 2.59 | 5.95 | 4.83 | 0.56 |
| 3 | 0.85 | 0.64 | 0.33 | 0.28 | 0.47 | 0.39 | 0.02 |

Table 2: Times in seconds for various test problems on ASCI Blue Pacific

The keep and remove operations can be done immediately as before. The scattering of the value $n$ allows the unions of the node patterns in each $S_n$ to be distinguished. Reads and writes have been reduced to the minimal amount.

Intersections need not always be found using information scattered into a vector of length $N$, i.e., recoverable using the global index. If the indices in the row or column are sorted, an intersection is easily computed using one pass through the row or column. Maintaining the sorted order, however, does complicate the removal process when vectors are used. If all of the edges to be removed are marked, then it is possible to move them to the removed region and maintain sorted order while making the equivalent of two passes through the active edges in the row or column. Of course, this means that for efficiency's sake it might be useful to add a "marked for removal" state to the column-oriented versions even if they do not require them from an algorithmic point of view. An alternative to maintaining all rows or columns sorted is to only sort when you get to a final form, e.g., sort columns of coarse nodes. Then a moderate cost intersection can be accomplished by using a binary search for each element in the column of the noncoarse node. Sorted linked lists for indices within a row or column are easily updated at the cost of doubling storage.

## 4.4 Sequential results

Various implementations using different data structures have been coded, and applied to three test matrices:

1. a 7-point Laplace operator on a $50 \times 50 \times 50$ grid,
2. a 27-point operator on a $50 \times 50 \times 50$ grid,
3. an unstructured elasticity matrix with 10,923 variables and an average of 71 nonzeros per row.

The timings of the application of the coarsening to each matrix are presented in Table 1 for a 1.5 Ghz Dell linux work station, in Table 2 for one processor of ASCI Blue Pacific and in Table 3 for one processor of ASCI White.

The results show that the CSR implementation, which requires the least amount of storage, is the slowest. However, using the swap option, described in Section 4.1, it can

| matrix | CSR | CSR, swap | CSC | CSC, swap | mod | mod, swap | transpose |
|---|---|---|---|---|---|---|---|
| 1 | 0.62 | 0.50 | 0.44 | 0.40 | 0.58 | 0.49 | 0.06 |
| 2 | 5.81 | 3.73 | 1.93 | 1.59 | 2.96 | 2.26 | 0.25 |
| 3 | 0.48 | 0.32 | 0.14 | 0.13 | 0.21 | 0.16 | 0.01 |

Table 3: Times in seconds for various test problems on ASCI White

be significantly improved. The modified implementation is faster than the CSR implementation, but slower than the CSC implementation. This version is more interesting in the parallel case, since the availability of both $S$ and $S^T$ decrease communication, but require the update of $S^T$ as the CSC implementation, but also some update of $S$, which is not needed in the CSC implementation. The overall fastest version is the CSC implementation with the swap option. Here the use of swapping does not lead to improvements as significant as in the case of the CSR implementation. The overall improvements achieved from CSR to CSC, swap range from 1.25 for a matrix with few nonzeros per row to a factor of 4 for a denser matrix. Improvements also vary on different machines. The smallest improvements could be observed on 1 processor of the ASCI Blue Pacific computer.

## 5 Distributed Memory Implementation

In this section, the implementation of this algorithm on a distributed memory machine is described. We assume here that each processor $p$ owns a nonoverlapping set of points. In order to correctly perform the algorithm, each processor requires the following information from their neighbor processors: the set $\delta p$ of offprocessor neighbors of local nodes, a sequence of *ghost graphs*, $G_i^\delta$, with $G_0^\delta = \delta p$, the weights of the nodes in $\delta p$ (*ghost weights*), their status information (*ghost cf values*) and their row and/or column information (*ghost rows* or *ghost columns*). The implementation defines what type of communication is required.

### 5.1 CSR implementation

The CSR implementation requires the row information $S_\delta$ of the ghost nodes. We assume that the information in $S_\delta$ has been filtered to include only local nodes and nodes in $\delta p$, since all others are not reachable from any local node via one edge, and since this leads to a more efficient implementation.

The second pass requires the status of the nodes (whether they ar coarse or not). The filtered $S$ is denoted $S^f$. The index information in $(S_v)^\delta$ need not be updated during the coarsening. It is only used in computing $S_v \cap (S_n \cap C_i)$. Since it is assumed that $C_i$ and $S_n$ are updated, nodes that have been removed from $S_v$ during its updates on processor $q$ are irrelevant. They have either been removed by the initial filtering or are ignored during the computation of the intersection. Their presence may of course influence efficiency, but the intersection can be driven by $(S_n \cap C_i)$ to mitigate any degradation. (Or an updated row could be sent if a substantial change in the population of $S_v$ has occurred.)

The main communication problem for the row-oriented version occurs due to the need to update the ghost weights. This implies the need to maintain a decrement count for nodes in $\delta p$, to communicate it to the owning processors, to receive decrements for nodes in $nodes(p)$ that are members of $\delta q$ for some $q \neq p$, to update the weights, communicate the final weights to/from the other processors, determine the next set of coarse nodes, and finally to transmit the new set of coarse nodes.

Also note that this affects the definition of $\delta p$. Since a purely row-oriented coarse determination loop is used, nodes that are on processors that point to an interior node on $p$ must have their weights made available to $p$. Of course, their status is not determined by $p$ but is needed to determine the status of interior nodes in $p$. An alternative to this approach would be to add another communication exchange, which transfers the status of the boundary points to the neighboring processor after the local maxima have been determined and compares the status of the changes made on the neighbor processor to those compared on the current processor. A true local maximum would be recognized as such on all involved processors. In this way the status of points that were falsely identified as local maximum could be corrected.

The communication of the second loop and the synchronization within the row-oriented implementation are therefore the main problems with this approach.

The complete algorithm with focus on required communication can be described as follows:

```
send ghost measures and receive measure decrements
update measures
send measures and receive ghost measures
for each $n \in G_i$
        if $n \in C_i$ or $n \in F_i$
             remove $n$ from $G_i$
        else
             for $v \in S_n$
                  if $\mu(n) > \mu(v)$ then $v \notin C_{i+1} \cup C^{\delta}_{i+1}$
                  if $\mu(v) > \mu(n)$ then $n \notin C_{i+1}$
             end for
        end if
end for
for each $n \in G^{\delta}_i$
        if $n \in C^{\delta}_i$ or $n \in F^{\delta}_i$
             remove $n$ from $G^{\delta}_i$
        else
             for $v \in S^{\delta}_n$
                  if $\mu(n) > \mu(v)$ then $v \notin C_{i+1} \cup C^{\delta}_{i+1}$
                  if $\mu(v) > \mu(n)$ then $n \notin C^{\delta}_{i+1}$
             end for
        end if
end for
compute global graph size via global exchange
if global graph size = 0 stop
send ghost $cf$ values and receive $cf$ values
compare received $cf$ values with current $cf$ values
        and correct falsely identified local maxima
send $cf$ values and receive ghost $cf$ values
```

```
for each $n \in G_i$
        if $n \in C_i$ (the node is coarse)
            for each $v \in S_n$
                mark edge $(n, v)$ for removal
                decrement $\mu(v)$ by 1
            end for
        else (the node is undetermined)
            for each $v \in S_n \cap (C_i \cup C_i^\delta)$
                mark edge $(n, v)$ for removal
                decrement $\mu(v)$ by 1
            end for
            for each $v \in S_n$ such that
                $(n, v)$ is not marked for removal (noncoarse)
                if $S_n \cap (S_v \cup S_v^\delta) \cap (C_i \cup C_i^\delta) \neq \emptyset$ then
                    mark edge $(n, v)$ for removal
                    decrement $\mu(v)$ by 1
                end if
            end for
        end if
end for
```

An alternative to the accumulation of changes in the weights of nodes in the boundary and their communication to the processors owning the boundary nodes is redundantly computing information for the boundary nodes. This approach reverses the point of view taken when designing the computation. The boundary rows are filtered to include only indices within $P \cup \delta P$ and the boundary nodes are processed as if they were internal nodes. This allows the boundary nodes effect on the weights of internal nodes to be computed. Note that the computation on the boundary nodes does not give the correct weight and status value for the boundary nodes themselves. These are still computed in the processor that owns the boundary nodes. The keys to efficiency are the filtering of the indices in the boundary rows to bound the number of nodes that need to be included in the redundant computation and the small number of boundary nodes relative to the number of internal nodes on each processor. The latter is not always true and large redundancy can result as the grid becomes more densely connected. Ideally, a hybrid of the two approaches would be used. Leading evidence however suggests that the first approach involving computing and communicating the changes to the weights of the boundary nodes is more widely applicable.

## 5.2 CSC Implementation

The CSC implementation, having been the most successful in the sequential implementation, has an additional difficulty in the implementation for distributed memory. Since only the information of incoming edges for each node is available and not the outgoing edges, the case of a node that has an outgoing edge to an offprocessor node without an incoming edge is problematic. Such as a node might not even be included in the ghost graph and therefore invisible to the processor. Therefore just as in the previous implementation, it is necessary to communicate ghost measures back and forth, in order to not obtain false local maxima.

Another difficulty arises due to the fact that edges connecting non-coarse nodes with a common coarse neighbor need to be eliminated. In this case, it is possible that a coarse point on processor $p$ has two neighbor nodes, whose common edge needs to be eliminated on processor $q$. Only processor $p$ has this information and needs to communicate it to $q$. This requires to send columns in $(S_n^T)^\delta$ to processor $q$, to find the edge in $S_n^T$ on $q$ and to eliminate it there.

Taking all this into account, the distributed memory CSC implementation requires the most expensive communication exchange.

The complete CSC implementation with required communication steps is presented below.

```
send ghost measures and ghost columns for the points,
        for which edges have been deleted
update measures and columns if necessary
receive ghost measures
for each $n \in G_i$
        if $n \in C_i$ or $n \in F_i$
            remove $n$ from $G_i$
        else
            for $v \in S_n^T$
                if $\mu(n) > \mu(v)$ then $v \notin C_{i+1} \cup C_{i+1}^\delta$
                if $\mu(v) > \mu(n)$ then $n \notin C_{i+1}$
            end for
        end if
end for
for each $n \in G_i^\delta$
        if $n \in C_i^\delta$ or $n \in F_i^\delta$
            remove $n$ from $G_i^\delta$
        else
            for $v \in (S_n^T)^\delta$
                if $\mu(n) > \mu(v)$ then $v \notin C_{i+1}C_{i+1}^\delta$
                if $\mu(v) > \mu(n)$ then $n \notin C_{i+1}^\delta$
            end for
        end if
end for
compute global graph size via all reduce
if global graph size = 0 stop
receive ghost $cf$ values
compare ghost $cf$ values with current $cf$ values
        and correct falsely identified local maxima
send ghost $cf$ values
for each $n \in G_i$
        if $n \notin C_i$ then (the node is not coarse)
            for each $v \in S_n^T \cap (C_i \cup C_i^\delta)$
                remove edge $(v, n)$ and decrement $\mu(n)$
            end for
        else (node is coarse)
```

```
                for each $v \in S_n^T$
                    Compute $S = S_n^T \cap (S_v^T \cup (S_v^T)^\delta)$
                    Remove edges $(p, v)$ from $(S_v^T \cup (S_v^T)^\delta)$ where $p \in S$
                    Decrement $\mu(v)$ by $\|S\|$
                end for
            end if
        end for
        for each $n \in G_i^\delta$
            if $n \notin C_i^\delta$ then (the node is not coarse)
                for each $v \in (S_n^T)^\delta \cap (C_i \cup C_i^\delta)$
                    remove edge $(v, n)$ and decrement $\mu(n)$
                end for
            else (node is coarse)
                for each $v \in (S_n^T)^\delta$
                    Compute $S = (S_n^T)^\delta \cap (S_v^T \cup (S_v^T)^\delta)$
                    Remove edges $(p, v)$ from $(S_v^T \cup (S_v^T)^\delta)$ where $p \in S$
                    Decrement $\mu(v)$ by $\|S\|$
                end for
            end if
        end for
```

## 5.3 Modified Implementation

In the modified implementation, the synchronization within a node is simplified considerably. Assuming, that both $S$ and $S^T$ are available, many of the previous communication steps can be omitted, since we have the information of both ingoing and outgoing edges from and to offprocessor nodes. For the most efficient implementation with the least amount of communication, it is necessary to define $\delta p$ as the union of all neighbors that are connected to nodes located on $p$, regardless whether the connection is incoming or outgoing. For a highly nonsymmetric matrix, this could lead to a set $\delta p$ that is significantly larger than the sets of ghost values considered in the CSR and CSC implementations.

The ghost graph, the ghost measures, the ghost $cf$ values, and the ghost column information $(S_n^T)^\delta$ are based on this definition of $\delta p$. The modified implementation is now defined below.

```
send measures and receive ghost measures
for each node $n \in G_i$
    if $n \in C_i$ or $n \in F_i$
        remove $n$ from $G_i$
    else (node stays for $G_{i+1}$)
        for $v \in S_n^T$
            if $\mu(n) > \mu(v)$ then $v \notin C_{i+1} \cup C_{i+1}^\delta$
            if $\mu(n) < \mu(v)$ then $n \notin C_{i+1}$
        end for
        for $v \in S_n \cap \delta p$
            if $\mu(n) > \mu(v)$ then $v \notin C_{i+1}^\delta$
```

```
                if $\mu(n) < \mu(v)$ then $n \notin C_{i+1}$
            end for
        end if
end for
compute global graph size via all reduce
if global graph size $= 0$ stop
send $cf$ values and receive ghost $cf$ values
for each node $n \in G_i - C_i$
        for each $v \in S_n^T$
            if $v \in C_i \cup C_i^\delta$
                remove edge $(v, n)$ from $S_n^T$ and decrement $\mu(n)$
            end if
        end for
        for each $v \in S_n$
            if $v \in C_i$
                for each $w \in S_n^T \cap S_v^T$
                    remove edge $(w, n)$ from $S_n^T$ and decrement $\mu(n)$
                end for
            end if
            if $v \in C_i^\delta$
                for each $w \in S_n^T \cap (S_v^T)^\delta$
                    remove edge $(w, n)$ from $S_n^T$ and decrement $\mu(n)$
                end for
            end if
        end for
end for
```

The boundary is, as with the CSR implementation, determined by a combination of the first and second loop. The second loop requires columns from other processors after they have become coarse. However, as with the CSR implmementation, they can be exchanged at the beginning and need not to be updated during the computation. Exchanging them only after they have become coarse would allow the development of an algorithm that would not need space for all of the boundary column information. As nodes that have become coarse are received, they could be treated by an alternative algorithm that would complete the fan-in required to update each local column. The details and tradeoffs for such an approach have yet to be considered, but a simple compromise seems viable. At the beginning of the algorithm, space is allocated for the entire boundary, but no index information is maintained, only fine/coarse status. When a node is made coarse, the current active indices in the column associated with that node are sent to its fanout processors, where they are filtered relative to locally owned indices and translated appropriately. This would require a more complicated message type structure.

These columns are for all nodes $v \notin nodes(p)$ such that $v \in S_n$ for some $n \in nodes(p)$. These are the same nodes as those in $\delta p$ from the first loop of the row-oriented implementation. As with the row-oriented version, only nodes in columns that are in $\delta p$ need to be kept, since they are the only ones that can appear in $S_n \cap C_i$ for $n \in nodes(p)$. Note that this implies that $S_n$ for $n \in nodes(p)$ must contain indices

| # of procs | CSR | CSR, swap | CSC | CSC, swap | mod | mod, swap | transpose |
|---|---|---|---|---|---|---|---|
| 1 | 0.67 | 0.63 | 0.49 | 0.49 | 0.70 | 0.69 | 0.05 |
| 8 | 0.82 | 0.79 | 0.67 | 0.65 | 0.84 | 0.85 | 0.07 |
| 64 | 1.37 | 1.36 | 1.23 | 1.21 | 1.36 | 1.41 | 0.13 |
| 216 | 2.69 | 2.67 | 2.48 | 2.54 | 2.67 | 2.69 | 0.28 |
| 512 | 6.98 | 5.01 | 5.03 | 4.95 | 5.18 | 5.52 | 0.88 |

Table 4: Times in seconds for the 3d 7-point Laplace operator on ASCI White

of the nodes not in $nodes(p)$.

The weight information for the boundary nodes determined by the $S_n$ is needed along with that for nodes $v \notin nodes(p)$ such that $v \in S_n^T$ for some $n \in nodes(p)$. Therefore the set of nodes in $\delta p$ for which weight information must be maintained is the same for the modified column and row-oriented versions.

Since the modified form has row and column information about $nodes(p) \cup \delta p$ and weight decrements need not be exchanged, the communication associated with the second loop is simpler than that for the row-oriented form.

First, the final weights of the boundary nodes need to be sent and received. The new coarse points can then be finalized using a row/column pointer algorithm, i.e., explicitly computing coarse points rather than the implicit form discussed above. Finally, the new coarse/fine state of boundary nodes needs to be communicated. The last exchange could also send the index information for coarse nodes for the version that delays sending their indices.

It is arguable that this hybrid form that sends each column to its neighboring processor only when it is coarsened is the minimal communication form of the algorithm. Each coarse boundary column is sent to its neighbor exactly once. If we only send an update in status, when a node becomes coarse or fine, then its status is sent to its neighboring processor once (it can be initialized as undetermined without communication). The only information that can be sent multiple times is the weight. This also can be sent only, when it changes, to minimize communication. The maximum number of weight changes, a communication to neighbor bound, is the number of elements per column and row.

## 5.4 Numerical Results

Tables 4, 5 and 6 show numerical results for various test matrices on varying number of processors on the ASCI White computer.

The results for a 3-dimensional 7-point Laplace operator in Table 4 show only small changes in the performance between the various implementations. Note that the difference between CSC and modified implementation decreases with an increasing number of processors, showing the influence of the additional communication that is required in the CSC implementation.

Table 5 shows results for a 27-point operator, with a larger amount of nonzeros per row. Just as in the previous experiment, the performance difference between the various implementations decreases with an increasing number of processors, showing again the influence of communication. Now the factor of fastest implementation to slowest implementation is only about 2.3 compared to almost 4 in the one processor case. Also

| # of procs | CSR | CSR, swap | CSC | CSC, swap | mod | mod, swap | transpose |
|---|---|---|---|---|---|---|---|
| 1 | 5.83 | 4.10 | 2.08 | 1.70 | 3.09 | 2.52 | 0.22 |
| 8 | 6.38 | 4.59 | 2.53 | 2.14 | 3.47 | 2.90 | 0.32 |
| 64 | 8.16 | 5.54 | 3.48 | 3.12 | 4.29 | 3.71 | 0.45 |
| 216 | 8.76 | 6.90 | 5.50 | 4.42 | 5.51 | 5.03 | 0.61 |
| 512 | 15.95 | 12.13 | 9.87 | 6.92 | 8.59 | 7.92 | 0.92 |

Table 5: Times in seconds for a 27-point operator on ASCI White

| # of procs | CSR | CSR, swap | CSC | CSC, swap | mod | mod, swap | transpose |
|---|---|---|---|---|---|---|---|
| 16 | 0.59 | 0.47 | 0.28 | 0.28 | 0.32 | 0.27 | 0.04 |

Table 6: Times in seconds for an elasticity problem on ASCI White

note that the influence of communication on the performance for the CSC and the modified implementation is even more noticable. Whereas, the CSC implementation is faster for a small numberof processors, it is about the same for 216 processors and even slower for the 512 processor case.

Finally, Table 6 contains the results for a 3-dimensional elasticity problem with 215,055 variables and an average of 56 nonzeroes per row.

# 6 Shared Memory

## 6.1 CSR Implementation

The CSR implementation described above would at first glance seem to be highly parallel. In fact, there are some synchronization difficulties that must be addressed. Assuming a parallel iteration, where each row, $i$, is processed on a single processor, the update of the column weights of nodes that are removed from $S_i$ by that iteration is the most obvious operation in need of synchronization via a lock variable. The major synchronization problem, however, arises due to the manner in which the $S_i$ data structure is manipulated. In the version above, when nodes are marked for removal, they are swapped to the end of the active list, where they form a new first node in the marked list and removed on the second pass. As a result, the data structure for any row is modified by the iteration, and it is possible for another processor processing, say $S_j$, where $j \in S_i$, to modify $S_j$ during the scan of $S_j$ by the iteration processing $S_i$. To see this, assume both iterations are active, and the iteration processing $S_i$ has read up to the middle of the index list in $S_j$. Now suppose at that time the iteration processing $S_j$ detects the first element on the list is coarse and should be removed. It swaps the first index in the active node list of $S_j$ with the last node. So when the iteration processing $S_i$ gets to the last index in the active list of $S_j$, it will read an index that it has already processed and, since it has been swapped to the first position, the index originally in the last position in $S_j$ will not be read by the iteration processing $S_i$. Note this example also demonstrates that simply locking $S_j$ during individual swaps does not fix the problem. One could attempt to lock a row, while it is being processed, i.e., no other iteration can read a row, until it is finished updating itself. The main difficulty with this approach is, when such locking is done, it is possible to create deadlock between two nodes that appear in each other's row list. The most reliable and probably inefficient way to fix this is having a row, $i$, attempt to lock the row structure of all nodes, $j$, whose indices it has in its row list. As it succeeds in locking each row, $j$, it copies the $S_j$ current list into workspace private to the processor executing the iteration processing $S_i$, then unlocks $S_j$. When all rows, $j \in S_i$ have been copied to workspace, $S_i$ can lock itself (assuming no other iteration is making a local copy) and proceed with its update. This approach requires considerable workspace and may have significant synchronization delay.

The easiest way around the extra space and synchronization above is to remove the modification of the row structure and revert to a very simple row-oriented version that simply marks nodes for removal in situ. The loops in such a simple row-oriented version are essentially parallel. The basic iteration body comprises visiting each active node and its row and performing updates. Only the iteration associated with each node performs writes to the node's information, therefore no synchronization is required, when marking an edge for removal. Iterations processing other nodes may read the updated information, but the mark for removal is irrelevant to their operations and, since the information in the list is not moving, nothing is lost. This approach, of course, suffers from the problem of having to pass through all indices on the active list any time a row is processed.

As noted earlier, synchronization is required for each edge removal due to the update of the weights. This will require additional storage, and the number of elements of the *weight* vector most effectively protected by a single lock should be evaluated. Note, for CLJP the weight is not an integer. So it is not possible to exploit a memory-based fetch and decrement. If such an instruction were available, then it would be worthwhile considering splitting the $\mu(i)$ into two components: the integer portion based on incoming edge count and the random number between 0 and 1 that is initially added to the count. The update to the weights and the comparisons for active/removed need only be done on the integer portion.

We can therefore create the following simple shared memory row-oriented version of the first pass through the active nodes. Dynamic scheduling is assumed, i.e., nothing is done that assumes static scheduling.

```
parallel do each node $n \in G_i$
        if $n \in C_i$ then (the node is coarse)
            for each node $v \in S_n$
                mark edge $(n, v)$ for removal in $S_n$
                critical–section(decrement $\mu(v)$ by 1)
            end for
        else $n \notin C_i$ ( the node is not coarse)
            for each node $v \in S_n$
                if $v \in C_i$
                   mark edge $(n, v)$ for removal
                   scatter $v$ in work vector
                   critical–section(decrement $\mu(v)$ by 1)
                end if
            end for
            (At this point $T = S_n \cap C_i$ is scattered)
            for each node $v \in S_n$
```

```
                    if $(S_v \cap T) \neq \emptyset$ then
                        mark edge $(n, v)$ for removal
                        critical–section(decrement $\mu(v)$ by 1)
                    end if
                end for
                for each node $v \in S_n$
                    if $v \in C_i$
                        clear $v$ in work vector
                    end if
                end for
            end if
        end parallel do
```

Nothing is explicitly considered in the suggested CSR implementation to enhance locality so as to better exploit the two levels of cache present in each node. Since each row is scanned and updated, there is spatial locality with respect to those particular reads and writes. Each row performs repeated reads of the $cf$ vector to check for membership of destination nodes in $C_i$. The only locality (spatial or temporal) that would result in these reads would depend upon the ordering of the nodes and their connectivity. If nodes were ordered by a scan that placed nodes near their neighbors (similar to the ordering required for distributed memory) the level 2 cache behavior of access to $cf$ might improve. The repeated locking and updates of the weights could cause false sharing problems. It is perhaps worthwhile to group the cf(i), int–measure(i), rand–measure(i), and lock(i) into a single row of a matrix or a structured element of a vector. This assumes a 4 word cache line in the node.

A difficulty with the simple version above or the more complicated synchronized version is the method, in which scattered vectors are handled, when preparing for membership checks, e.g., intersections. The main difference between the shared memory parallel and sequential implementation concerns the use of $cf$ or a work vector in the computation of the various intersections discussed above. Since multiple rows will be active simultaneously, care must be taken to provide the necessary private workspace per processor. If this workspace is deemed excessive, then the use of ordered indices within the row vectors should be considered with appropriate changes to the computation of the intersections.

For example, suppose $cf$ is used to compute the intersections. The sequential iteration for node $n$ uses $cf$ to mark $S_n \cap C_i$. When done in parallel, the values in $cf$ would reflect the union of $p$ intersections and therefore, when the intersection with $S_v$, where $v \in S_n$, is computed on a processor, it is impossible to tell of which intersection a particular entry in $cf$ is a member. This could be remedied easily, if it was known that the $S_{n_j} \cap C_i$ $j = 1, \ldots, p$ were disjoint. Since they are not, a more complicated approach is required.

If the number of processors is less than the number of bits in the integers used to represent an element of $cf$, then the problem can be solved. Node $j$ can be a member of at most $p$ intersections at any given time. This can be represented by, say, the $p$ low order bits in the representation of $cf(j)$ (recall $cf(j)$ will be a negative integer, $-i$ if $j \in C_i$ in the sequential implementation). Membership in $C_i$ and at least one of the currently active $p$ intersections will be indicated by $-a$ where $0 < a < 2^p$. When membership in a particular intersection, i.e., from the processor's making the inquiry, the appropriate bit is checked. At the end of each iteration, when the $S_{n_j} \cap C_i$ is cleared, this amounts to clearing a particular bit (and restoring $i$ to the location, if it is the last intersection to be cleared for node $j$). All writes must be done under lock to avoid erroneously handling multiple clears and marking, but reads can be done irrespective active updates, since the processor that clears a particular bit is also the one that examines it.

Note that the clearing portion of the iteration requires rescanning all of $S_n$ or saving the indices of the intersection $S$. Storing $S$ requires work space of the size of the maximum row count private to each processor. If this is acceptable, the synchronization required in the approach above can be avoided by scanning the entire list of $S$. This could be accelerated by sorting $S$ after its computation. However, if rows are fairly small, then simply storing and scanning for membership is probably the best way to go.

In the current row-oriented form of the algorithm, the second pass through $G_i$ performs the removal of coarse and fine nodes from the *nodes* vector. In the sequential implementation, this simply requires altering a pointer to the end of the active part of the vector and swapping the removed node with the last active node. When dynamic scheduling is used in a shared memory implementation, this simple strategy is no longer adequate. Locking the pointer is possible as is swapping, but there is no guarantee in a general dynamic scheduling algorithm that the swapped in node is not already being processed, or, if not, that it ever will be, when moved to the new location in the vector. (This assumes that one is using an OPENMP-like dynamic scheduler, where given a loop of iterations $i = 1, n$ the compiler and runtime library are free to schedule in any way they see fit without user knowledge or approval.)

There are three main ways to solve this problem. One is to implement a specific dynamic scheduling mechanism that would have more coordination between executing the iterations that may be swapped from the end of the list. This, of course, would increase the scheduling overhead and is probably not worth the effort at this point. The second is to move to statically scheduled loops, where each processor would have its own local portion of the removed locations that it would expand as some of its nodes are identified as coarse or fine. This is a simple solution, the main problem becomes load balancing. In this case the portion of the vector assigned to each processor would vary and the removed locations would become fragmented adding overhead to the scan of the list, i.e., more than just active nodes would be visited. As a result, occasional garbage collection would be required. Even then the algorithm for static scheduling would have to be carefully considered. The processing time for each node varies considerably, and therefore simply balancing the number of nodes sent to each processor may not be an adequate strategy.

The most straightforward solution at this point is to use extra storage (similar to what would be needed, if the active nodes were maintained in a linked list). In this case, the first pass through $G_i$ would remove $C_i$, not by explicit action on the coarse nodes, but by adding noncoarse nodes to a second nodes vector, *nodes*2. This list would grow by locking a pointer to the first open location in the vector, and a processor would add one or more noncoarse nodes in $G_i$ to it. At the end of the first pass through $G_i$, the vector *nodes*2 would have $G_i - C_i$. The second pass through the graph identifies $F_i$ and removes it in a similar manner, i.e., nodes not in $F_i$ would be written to a new version of the original *nodes* vector. As a result, the *nodes* vector would contain $G_{i+1}$ and the subset $C_{i+1}$ would be identified in the vector $cf$. Note that not all of the versions described above assume that $C_i$ is removed during the first pass. All of them however allow this option.

The actual updating during the second pass of the status of nodes in *nodes*2 from assumed coarse to noncoarse by multiple writes per node to $cf$ does not require any synchronization, although there should be some performance degradation due to the coherence mechanism.

Finally, note that when a node is removed from the graph, its row indices can be restored to their original input values. This must be done at some point, before the algorithm returns, and the second pass is a convenient point so to do. The coarse nodes may have been removed and restored in the first pass. Below we assume this has not been done. (The parallel pseudocode shared memory implementation does assume this.) The second pass is completely parallel so the code above is easily changed into the following.

```
parallel do each node $n \in G_i$
        if $n \in C_i$ then (the node is coarse)
            restore all edges in $S_n$
            remove $n$ from $G_i$
        else if $\mu(n) < 1$ and $\|S_n\| = 0$ then (the node is fine)
            add node to $F_i$
            remove node from $G_i$
            restore all edges in $S_n$
        else (node stays for $G_{i+1}$)
            remove all marked edges from $S_n$
            for $v \in S_n$ (all unmarked edges)
                if $\mu(n) > \mu(v)$ then $v \notin C_{i+1}$
                if $\mu(n) < \mu(v)$ then $n \notin C_{i+1}$
            end for
        end if
end parallel do
```

## 6.2 CSC Implementation

The synchronization required by a parallel version of the suggested column-oriented approach is simpler than the row-oriented version. A noncoarse node only scans and updates its own list. It does not update any other node. A coarse node scans and updates its own list and scans and may update the columns of noncoarse nodes in its column list. As a result, deadlock cannot occur from locking a column.

The parallel column-oriented strategy (including the immediate removal and swapping of elements on the index list) is the same as the suggested sequential version, except for processing the main loop over the active nodes in parallel and adding the following synchronization considerations:

- A column that is not coarse locks itself at the beginning of its iteration.
- A column, $i$, that is coarse need not lock itself, when processing its own column index list. It must, however, obtain, in turn, the lock for each of the columns

that are not coarse and contained in $S_i^T$ and release it only after completing the update.

Given such a simple synchronization strategy, the main concern with the parallel column-oriented algorithm is the processing strategy for the simultaneous scattered vectors.

## 6.3 Modified Implementation

Both suggested sequential versions of the modified column-oriented approach have trivial synchronization. A parallel loop over the nodes only requires the barrier implicit in the starting and ending of parallel execution. During the loop, no synchronization is required between parallel iteration. The source of this efficiency is easily seen for both versions. Coarse nodes are not updated during the iteration. A noncoarse node, $i$, updates only its own list by removing coarse nodes and by combining with the columns of coarse nodes contained in $S_i$. The two suggested implementations differ only in the manner, in which the processing of this combination is done. As a result, from a synchronization point of view the modified column approaches are optimal.

Of course the complicating factor, as with all of these shared memory approaches is the parallel scattering workspace. The first suggested implementation of the modified column approach can use the bit per processor approach in a work vector as described above. The second implementation, however, depends heavily on having a work vector of integers with length equal to the number of nodes in the graph per processor. This is feasible only for a moderate number of processors.

## 7 Conclusions

The use of different data structures and their effect on the implementation of the CLJP algorithm, a parallel coarsening algorithm, has been investigated in detail. Numerical results for both sequential and distributed parallel implementations show that the fastest implementation is the CSC implementation with a swapping mechanism, even if the transpose of the strength matrix is not available, and a transpose needs to be performed. In this case, memory usage is, however, increased, and the CSR implementation would be the one with the lowest memory usage.

On a parallel distributed memory computer with high communication cost, the modified implementation might be the fastest, since it has the lowest communication cost.

Finally, the theoretical investigation of the shared memory implementations shows that the CSR implementation is the most problematic implementation, whereas the CSC and the modified implementation can be parallelized in a fairly straightforward way.

When deciding on a data structure, one needs to also consider that the coarsening algorithm is just one part of a larger algorithm, the algebraic multigrid algorithm. The final choice of data structure is affected by other factors as well, such as memory usage, and which data structures are suitable for other parts of the code. For example, for the interpolation routine a CSR data structure would be the more natural choice.

Nevertheless, this investigation shows how important it is to carefully select the data structures before implementing an algorithm, since they can have a significant effect on the final performance.

## References

[1] Cleary, A., Falgout, R., Henson, V., Jones, J.: Coarse-grid selection for parallel algebraic multigrid. in Proc. of the 5th Intern. Sympos. on Solving Irregularly Structured Problems in Parallel, Lecture Notes in Computer Science **1457** (1998) 104–115.

[2] Henson, V. E., Yang, U. M.: *BoomerAMG*: a parallel algebraic multigrid solver and preconditioner. Applied Numerical Mathematics **41** (2002) 155–177.

[3] Jones, M., Plassman, P.: A parallel graph coloring heuristic. SIAM J. Sci. Comput. **14** (1993) 654–669.

[4] Luby, M.: A simple parallel algorithm for the maximal independent set problem. SIAM J. on Computing **15** (1986) 1036–1053.