A Parallel Maximal Independent Set Algorithm

Mark Adams *

January 29, 1998

Abstract

The parallel construction of *maximal independent sets* is a useful building block for many algorithms in the computational sciences, including graph coloring and multigrid coarse grid creation on unstructured meshes. We present an efficient asynchronous maximal independent set algorithm for use on parallel computers, for "well partitioned" graphs, that arise from *finite element* (FE) models. For appropriately partitioned bounded degree graphs, it is shown that the running time of our algorithm under the P-RAM computational model is of $O(1)$, which is an improvement over the previous best P-RAM complexity for this class of graphs. We present numerical experiments on an IBM SP, that confirm our P-RAM complexity model is indicative of the performance one can expect with practical partitions on graphs from FE problems.

Key words: maximal independent sets, multigrid, parallel algorithms, graph coloring

1 Introduction

An *independent set* is a set of vertices $I \subseteq V$ in a graph $G = (V, E)$, in which no two members of I are adjacent (i.e. $\forall v, w \in I, (v, w) \notin E$); a *maximal independent set* (MIS) is an independent set for which no proper superset is also an independent set. The parallel construction of an MIS is useful in many computing applications, such as graph coloring and coarse grid creation for multigrid algorithms on unstructured FE meshes. In addition to requiring an MIS (which is not unique), many of these applications want an MIS that maximizes a particular application dependent quality metric. Finding the optimal solution in many of these applications is an NP-complete problem; for this reason greedy algorithms in combination with heuristics are commonly used for both the serial and parallel construction of MISs. Many of the graphs of interest arise from physical models; these graphs are sparse and their vertices are connected to only their *nearest neighbors*. The vertices of such graphs have a bound Δ on their maximum degree. We will discuss our method of attaining $O(1)$ P-RAM complexity bounds for computing an MIS on such graphs, namely finite element (FE) models in three dimensional solid mechanics. Our algorithm is notable in that it does not rely on global random vertex ordering (see [5, 7] and the references therein) to achieve correctness in a distributed memory computing environment but explicitly uses knowledge of the graph partitioning to provide for the correct construction of an MIS in an efficient manner.

The complexity model of our algorithm also has the attractive attribute that it requires far fewer processors than vertices, in fact we need to restrict the number of processors used in order to attain optimal complexity. Our P-RAM model uses $P = O(n)$ processors ($n = |V|$) to compute an MIS, but we restrict P to be at most a fixed fraction of n to attain the optimal theoretical complexity. The upper bound on the number of processors is however far more than the number of processors that are generally used in practice on common distributed memory computers of today; so given the common use of relatively *fat* processor nodes in modern computers, our theoretical model allows for the use of many more processors than one would typically use in practice. Thus, in addition to obtaining optimal P-RAM complexity bounds, our complexity model reflects the way that modern machines are actually used. Our numerical experiments confirm our

*Department of Civil Engineering, University of California Berkeley, Berkeley CA 94720 (madams@cs.berkeley.edu). This work is supported by DOE grant No. W-7405-ENG-48

claim and thus we believe that our algorithm is not just of theoretical interest but of practical interest as well.

We will not include the complexity of the graph partitionings in our complexity model, though our method explicitly depends on these partitions. We feel justified in this as it is reasonable to assume that the MIS program is embedded in a larger application that requires partitions that are usually much better than the partitions that we require.

This paper is organized as follows. In §2 we describe a new asynchronous distributed maximal independent set algorithm, in §2 we show that our algorithm has optimal performance characteristics under the P-RAM communication model for the class of graphs from discretized PDEs. Numerical results of the method are presented in §3, and we conclude in §5 with possible directions for future work.

2 An asynchronous distributed memory maximal independent set algorithm.

Consider a graph $G = (V, E)$ with vertex set V, and edge set E, an edge being an unordered pair of distinct vertices. Our application of interest is a graph which arises from a FE analysis, where elements can be replaced by the edges required to make a clique of all vertices in each element, see figure 1. FE methods, and indeed most discretization methods for PDEs, produce graphs in which vertices only share an edge with their nearest physical neighbors, thus the degree of each vertex $v \in V$ can be bounded by some constant Δ. We will restrict ourselves to such graphs in our complexity analysis. Furthermore to attain our complexity bounds we must also assume that vertices are "partitioned well" (which will be defined later) across the machine.

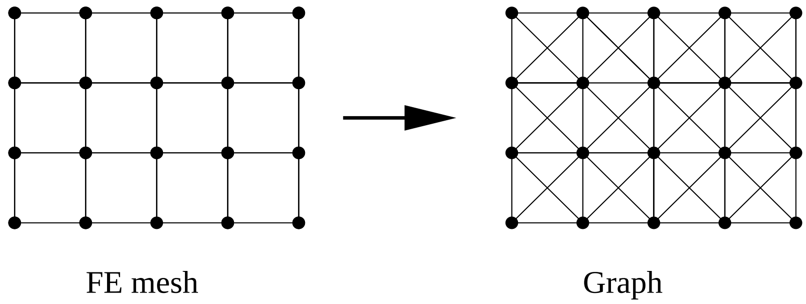

Figure 1: Finite element quadrilateral mesh - graph

We will introduce our algorithm by first describing the basic random greedy MIS algorithms described in [7]. We will utilize an *object oriented* notation from common programming languages, as well as set notation, in describing our algorithms; this is done to simplify the notation and we hope it does not distract the uninitiated reader. We endow vertices v with a mutable data member *state*, $state \in \{selected, deleted, undone\}$. All vertices begin in the *undone* state, and end in either the *selected* or *deleted* state; the MIS is defined as the set of *selected* vertices. Each vertex v will also be given a list of adjacencies *adjac*.

Definition 1: The adjacency list for vertex v is defined by $v.adjac = \{v1 \mid (v, v1) \in E\}$

We will also assume that $v.state$ has been initialized to the *undone* state for all v and $v.adjac$ is as defined in *Definition* 1 in all of our algorithm descriptions. With this notation in place we show the basic MIS algorithm (BMA) in figure 2.

For parallel processing we partition the vertices onto processors and define the vertex set V_p owned by processor p of P processors. Thus $V = V_1 \cup V_2 \cup ... \cup V_P$ is a disjoint union, and for notational convenience we give each vertex an immutable data member *proc* after the partitioning is calculated to indicate which processor is responsible for it. Define the *edge separator* set E^S to be the set of edges (v, w) such that $v.proc \neq w.proc$. Define the *vertex separator* set $V^S = \{v \mid (v, w) \in E^S\}$ (G is undirected, thus (v, w) and (w, v) are equivalent). Define the *processor vertex separator* set, for processor p, by $V_p^S = \{v \mid (v, w) \in E^S$

forall $v \in V$
  if $v.state = undone$ **then**
    $v.state \leftarrow selected$
    forall $v1 \in v.adjac$
      $v1.state \leftarrow deleted$
$I \leftarrow \{v \in V \mid v.state = selected\}$

Figure 2: Basic MIS Algorithm (BMA) for the serial construction of an MIS

and $(v.proc = p$ or $w.proc = p)\}$. Further define a processors *boundary* vertex set by $V_p^B = V_p \cap V^S$, and a processors *local* vertex set by $V_p^L = V_p - V_p^B$. Our algorithm provides for correctness and efficiency in a distributed memory computing environment by first assuming a given ordering or numbering of processors so that we can use inequality operators with these processor numbers. As will be evident later, if one vertex is placed on each processor (an activity of theoretical interest only), then our method will degenerate to one of the well known random types of algorithms [7].

We define a function $mpivs(vertex\_set)$ (an acronym for "maximum processor in vertex set"), which operates on a list of vertices:

$$Definition\ 2{:}\ mpivs(vertex\_set) = \left\{ \begin{array}{ll} max\{v.proc \mid v \in vertex\_set, v.state \neq deleted\} & if\ \ vertex\_set \neq \emptyset \\ -\infty & if\ \ vertex\_set = \emptyset \end{array} \right\}$$

Given these definitions and operators, our algorithm works by implementing two rules within the BMA running on processor p, as shown below.

- *Rule* 1: Processor p can *select* a vertex v only if $v.proc = p$.
- *Rule* 2: Processor p can *select* a vertex v only if $p \geq mpivs(v.adjac)$.

Note that *Rule* 1 is a *static* rule, because *v.proc* is immutable, and can be enforced simply by iterating over V_p on each processor p when looking for vertices to select. In contrast, *Rule* 2 is *dynamic* because the result of $mpivs(v.adjac)$ will in general change (actually monotonically decrease) as the algorithm progresses and vertices in $v.adjac$ are deleted.

2.1 Shared Memory Algorithm

Our Shared Memory MIS Algorithm (SMMA) in figure 3, can be written as a simple modification to BMA.

while $\{v \in V_p \mid v.state = undone\} \neq \emptyset$
  forall $v \in V_p$ - - implementation of *Rule* 1
    if $v.state = undone$ **then**
      if $p \geq mpivs(v.adjac)$ **then**   - - implementation of *Rule* 2
        $v.state \leftarrow selected$
        forall $v1 \in v.adjac$
          $v1.state \leftarrow deleted$
$I \leftarrow \{v \in V \mid v.state = selected\}$

Figure 3: Shared Memory MIS Algorithm (SMMA) for MIS, running on processor p

We have modified the vertex set that the algorithm running on processor p uses (to look for vertices to select), so as to implement *Rule* 1. We have embedded the basic algorithm in an iterative loop and added a test to decide if processor p can select a vertex, for the implementation of *Rule* 2. Note, the last line of figure 3 may delete vertices that have already been deleted, but this is inconsequential.

There is a great deal of flexibility in the order which vertices are chosen in each iteration of the algorithm. Herein lies a simple opportunity to apply a heuristic, as the first vertex chosen will always be *selectable* and the probability is high that vertices which are chosen early will also be selectable. Thus if an application can identify vertices that are "important" then those vertices can be ordered first and so that a less important vertex can not delete a more important vertex. For example, in the automatic construction of coarse grids for multigrid equation solvers on unstructured meshes one would like to give priority to the boundary vertices [1]. This is an example of a *static* heuristic, that is a *ranking* which can be calculated initially and does not change as the algorithm progresses. *Dynamic* heuristics are more difficult to implement efficiently in parallel. An example is the saturation degree ordering (SDO) used in graph coloring algorithms [3]: SDO colors the the vertex with a maximum number of different colored adjacencies; the degree of an uncolored vertex will increase as the algorithm progresses and its neighbors are colored. We know of no MIS application, that does not have a quality metric to maximize - thus it is of practical importance that an MIS algorithm can accommodate the use of heuristics effectively. Our method can still implement the "**forall**" loops, with a serial heuristic, i.e. we can iterate over the vertices in V_p in any order that we like. To incorporate static heuristics globally (i.e. a ranking of vertices), one needs to augment our rules and modify SMMA, see [1] for details, but in doing so we lose our complexity bounds, in fact if one assigns a random rank to all vertices this algorithm would degenerate to the random algorithms described in [7, 5].

To demonstrate correctness of SMMA we will proceed as follows: show termination; show that the computed set I is maximal; and show that independence of $I = \{v \in V \mid v.state = selected\}$ is an invariant of the algorithm.

- Termination is simple to prove and we will do so in §3.
- To show that I will be maximal we can simply note that if $v.state = deleted$ for $v \in V$, v must have a selected vertex $v1 \in v.adjac$ as the only mechanism to delete a vertex is to have a selected neighbor do so. All deleted vertices thus have a selected neighbor and they can not be added to I and maintain independence, hence I is maximal.
- To show that I is always independent first note that I is initially independent - as I is initially the empty set. Thus it suffices to show that when v is added to I, in line 7 of figure 3, no $v1 \in v.adjac$ is selected. Alternatively we can show that $v.state \neq deleted$ in line 7, since if v can not be deleted then no $v1 \in v.adjac$ can be selected. To show that $v.state \neq deleted$ in line 7 we need to test three cases for the processor of a vertex $v1$ that could delete v:
  - Case 1) $v1.proc < p$: v would have blocked $v1.proc$ from selecting $v1$, because $mpivs(v1.adjac) \geq v.proc = p > v1.proc$, so the test on line 6 would not have been satisfied for $v1$ on processor $v1.proc$.
  - Case 2) $v1.proc = p$: v would have been deleted, and not passed the test on line 5, as this processor selected $v1$ and by definition there is only one thread of control on each processor.
  - Case 3) $v1.proc > p$: as $mpivs(v.adjac) \geq v1.proc > p$ thus $p \not\geq mpivs(v.adjac)$ the test on line 6 would not have succeeded, line 7 would not be executed on processor p.

Further we should show that a vertex v with $v.state = selected$ can not be deleted, and $v.state = deleted$ can not be selected. For a v to have been selected by p it must have been selectable by p (i.e. $\{v1 \in v.adjac \mid v1.proc > p,\ v1.state \neq deleted\} = \emptyset$). However for another processor $p1$ to delete v, $p1$ must select $v1$ ($p1 = v1.proc$), this is not possible since if neither v nor $v1$ are deleted then only one processor can satisfy line 6 in figure 3. This consistency argument will be developed further in §3. Thus, we have shown that $I = \{v \in V \mid v.state = selected\}$ is an independent set and, if SMMA terminates, I will be maximal as well.

2.2 Distributed Memory Algorithm

For a distributed memory version of this algorithm we will use a message passing paradigm and define some high level message passing operators. Define $send(proc, X, Action)$ and $receive(X, Action)$ - $send(proc, X, Action)$ sends the object X and procedure $Action$ to processor $proc$, $receive(X, Action)$ will receive this message on processor *proc*. Figure 4 shows a distributed memory implementation of our MIS algorithm running on processor p. We have assumed that the graph has been partitioned to processors 1 to P, thus defining V_p, V_p^S, V_p^L, V_p^B, and $v.proc$ for all $v \in V$.

A subtle distinction must now be made in our description of distributed memory version of the algorithm in figure 4 and 5: vertices (e.g. v and $v1$) operate on local copies of the objects and not to a single shared object. So, for example, an assignment to $v.state$ refers to assignment to the local copy v on processor p. Each processor will have a copy of the set of vertices $V_p^E = V_p \cup V_p^S$, i.e. the local vertices V_p and one layer of "ghost" vertices. Thus all expressions will refer to the objects (vertices) in processor p's local memory.

```
while {v ∈ V_p | v.state = undone} ≠ ∅
    forall v ∈ V_p^B                              - - implementation of Rule 1
        if v.state = undone then
            if p ≥ mpivs(v.adjac) then            - - implementation of Rule 2
                Select(v)
                proc_set ← {proc | v ∈ V_proc^S} − p
                forall proc ∈ proc_set
                    send(proc, v, Select)
    while receive(v, Action)
        if v.state = undone then
            Action(v)
    forall v ∈ V_p^L                              - - implementation of Rule 1
        if v.state = undone then
            if p ≥ mpivs(v.adjac) then            - - implementation of Rule 2
                Select(v)
                if v1 ∈ V^B then
                    proc_set ← {proc | v1 ∈ V_proc^S} − p
                    forall proc ∈ proc_set
                        send(proc, v1, Delete)
I ← {v ∈ V | v.state = selected}
```

Figure 4: Asynchronous Distributed Memory MIS Algorithm (ADMMA) running on processor p

```
procedure Select(v)
    v.state ← selected
    forall v1 ∈ v.adjac
        Delete(v1)
procedure Delete(v1)
    v1.state ← deleted
```

Figure 5: ADMMA "Action" procedures running on processor p

Note that the value of $mpivs(v.adjac)$ will monotonically decrease as $v1.state$ ($v1 \in v.adjac$) are deleted, thus as all tests to select a vertex, are of the form $p \geq mpivs(v.adjac)$ some processors will have to wait for other processors to do their work (i.e. select and delete vertices). In our distributed memory algorithm in figure 4 the communication time is added to the time that a processor may have to wait for work to be done by another processor; this does not effect the correctness of the algorithm but it may effect the resulting MIS. Thus ADMMA is not a *deterministic* MIS algorithm; although the synchronous version - that we use for our numerical results - is deterministic for any given partitioning.

The correctness for ADMMA can be shown in a number of ways, but first we need to define a *weaving monotonic path* (WMP) as a path of length t in which each consecutive pair of vertices ($(v_i,v_j) \in E$) satisfies $v_i.proc < v_j.proc$, see figure 6.

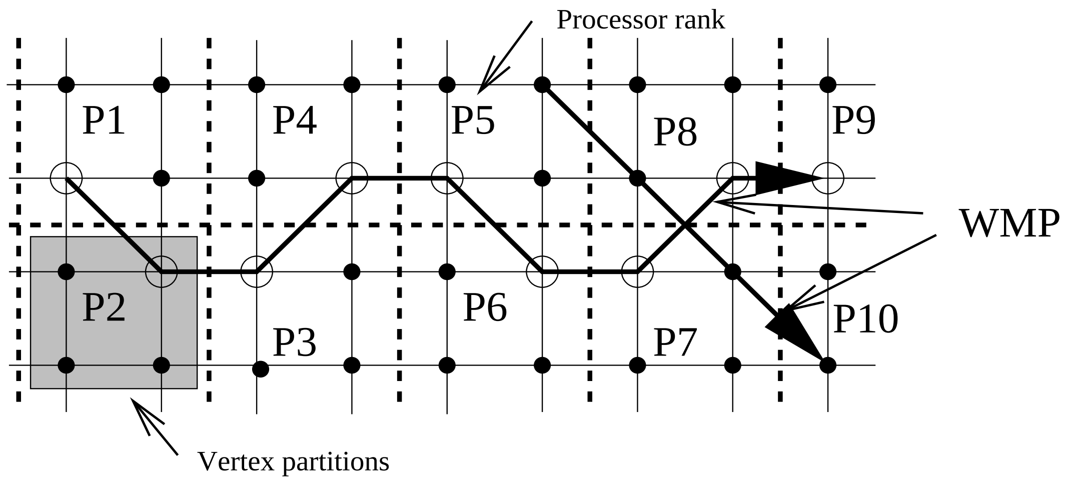

Figure 6: Two weaving monotonic paths (WMP)s in a 2D FE mesh

One can show that the semantics of ADMMA run on a partitioning of a graph is equivalent to a random algorithm with a particular set of "random" numbers. Alternatively, we can use the correctness argument from the shared memory algorithm and show consistency in a distributed memory environment. To do this first make a small isomorphic transformation to ADMMA in figure 4:

- Remove $v.state \leftarrow selected$ in figure 5, and replace it with a memoization of v so as to avoid "selecting" v again.
- Remove the "Select" message from figure 4 and modify the *Select* procedure in figure 5 to send the appropriate "Delete" messages to processors that "touch" $v1 \in v.adjac$.
- Redefine I to be $I = \{v \in V \mid v.state \neq deleted\}$ at the end of the algorithm.
- Change the termination test to: **while** $(\exists v1 \in v.adjac \mid v \in V_p, v.state \neq deleted, v1.state \neq deleted)$, or simply **while** ($I$ is not independent).

This does not change the semantics of the algorithm but removes the *selected* state from the algorithm and makes it mathematically simpler (although less concrete of a description). Now only *Delete* messages need to be communicated and $v.state \leftarrow deleted$ is the only change of state in the algorithm. Define the directed graph $G^{WMP} = (V^S, E^{WMP})$, $E^{WMP} = \{(v,w) \in E \mid v.proc < w.proc\}$; in general G^{WMP} will be a forest of acyclic graphs. Further define $G_p^{WMP} = (V_p^S, E_p^{WMP})$, $E_p^{WMP} = \{(v,w) \in E \mid w.state \neq deleted, v.proc = p\}$. G_p^{WMP} is the current local view of G^{WMP} with the edges removed for which the "source" vertices have been deleted. *Rule* 2 can now be restated: processor p can only select a vertex that is not a leaf in G_p^{WMP}. Processors will delete "down stream" edges in E^{WMP} and send messages so as other processors can delete their "up stream" copies of these edges, thus G_p^{WMP} will be pruned as p deletes vertices and receives delete messages.. Informally, consistency for ADMMA can be inferred as the only information flow (explicit delete messages between processors) moves down acyclic graphs in G^{WMP}; as the test, $(\forall v1 \in v.adjac \mid v1.proc \leq p$ or $v1.state = deleted)$ for processor p to select a vertex v, requires that *all* edges (in E^{WMP}) *to* v are "deleted". Thus the order of the reception of these delete messages is inconsequential and there is no opportunity for race conditions or ambiguity in the results of the MIS. More formally we can show that these semantics insure that $I = \{v \in V \mid v.state \neq deleted\}$ is maximal and independent:

- I is independent as no two vertices (in I) can remain dependent forever. To show this we note that the only way for a processor p to *not* be able to select a vertex v is for v to have a neighbor v_1 on a higher processor. If v_1 is deleted then p is free to "select" v. Vertex v_1 on processor p_1 can in tern be selected unless it has a neighbor v_2 on a higher processor. Eventually the end of this WMP will be reached and processor p_t will process v_t and thus release p_{t-1} to select v_{t-1} and on down the line. Therefore no pair of undone vertices will remain, and I will eventually be independent.

- I is maximal as the only way for a vertex to be deleted is to have a selected neighbor. To show that no vertex v that is "selected" can ever be deleted, as in our shared memory algorithm, we need to show that three types of processors p_1 with vertex v_1 can not delete v.
  - For $p = p_1$: we have the correctness of the serial semantics of BMA to ensure correctness, i.e. v_1 would be deleted and p would not attempt to select it.
  - For $p > p_1$: p_1 will not pass the $mpivs(v_1.adjac)$ test as in the shared memory case.
  - For $p < p_1$: p does not pass the $mpivs(v_1.adjac)$ and will not "select" v in the first place.

Thus I is maximal and independent.

3 Complexity of the asynchronous maximal independent set algorithm

In this section we derive the complexity bound of our algorithm under the P-RAM computational model. To understand the costs of our algorithm we need to bound the cost of each outer iteration, as well as, bound the total number of outer iterations. To do this we will first need to make some restrictions on the graphs that we work with and the partitions that we use. We assume that our graphs come from physical models, that is vertices are only connected by an edge to its nearest neighbors so the maximum degree Δ of any vertex is bounded. We will also assume that our partitions satisfy a certain criterion (for regular meshes we can illustrate this criterion with regular rectangular partitions and a *minimum* logical dimension that depends only on the mesh type). We can bound the cost of each outer iteration by requiring that the sizes of the partitions are independent of the total number of vertices n. Further we will assume that the asynchronous version of the algorithm is made synchronous by including a barrier at the end of the "receive" **while** loop, in figure 4, at which point all messages are received and then processed in the next **forall** loop. This synchronization is required to avoid more than one leg of a WMP from being processed in each outer iteration. We need to show that the work done in each iteration on processor p is of order N_p ($N_p = |V_p|$). This is achieved if we use $O(n)$ processors and can bound the *load imbalance* (i.e. $max\{N_p\}/min\{N_p\}$) of the partitioning.

LEMMA 3.1. *With the synchronous version of ADMMA, the running time of the P-RAM version of one outer iteration in figure 4 is* $O(1) = O(n/P)$, *if* $max\{N_p\}/min\{N_p\} = O(1)$

Proof. We need to bound the number of processors that *touch* a vertex v i.e. $|v|_{proc} \equiv \left|proc \mid v \in V^S_{proc}\right|$ In all cases $|v|_{proc}$ is clearly bounded by Δ. Thus, $max\,|v|_{proc} * N_p$ is $O(1)$ and is an upper bound (and a very pessimistic bound) on the number of messages sent in one iteration of our algorithm. Under the P-RAM computational model we can assume that messages are sent between processors in constant time and thus our communication costs in each iteration is $O(1)$. The computation done in each iteration is again proportional to N_p and bounded by $\Delta * N_p$, the number of vertices times the the maximum degree. This is also a very pessimistic bound that can be gleaned by simply following all the execution paths in the algorithm and successively multiplying by the bounds on all of the loops (Δ and N_p). The running time for each outer iteration is therefore $O(1) = O(n * \Delta/P)$.
◇

Notice for regular partitions $|v|_{proc}$ is bounded by 4 in 2D, and 8 in 3D, and that for *optimal* partitions of large meshes $|v|_{proc}$ is about 3 and 4 for 2D and 3D respectively. The number of *outer* iterations, in figure 4, is a bit trickier to bound. To do this we will need to look at the mechanism by which a vertex *fails* to be selected.

LEMMA 3.2. *The running time in the P-RAM computational model, of ADMMA, is bounded by the maximum length weaving monotonic path in G.*

Proof. To show that the number of outer iterations is proportional to the maximum length WMP in G, we need to look at the mechanism by which a vertex can *fail* to be selected in an iteration of our algorithm and thus potentially require an additional iteration. For a processor p_1 to fail to *select* a vertex v_1, v_1 must have an *undone* neighbor v_2 on a higher processor p_2. For vertex v_2 to not be selectable, v_2 in turn must have an *undone* neighbor v_3 on a higher processor p_3 and so on until v_t is the top vertex in the WMP. The vertex v_t at the end of a WMP will be processed in the first iteration as there is nothing to stop $v_t.proc$ from selecting or deleting v_t. Thus, in the next iteration, the top vertex v_t of the WMP will have been either selected or deleted; if v_t was selected then v_{t-1} will have been deleted and the *Undone* WMP (UWMP), a path in G^{WMP}, will be at most of length $t-2$ after one iteration; and if v_t was deleted (the worst case) then the UWMP could be of at most length $t-1$. After t outer iterations the maximum length UWMP will be of length zero, thus all vertices will be selected or deleted. Therefore, the number of outer iterations is bounded by the longest WMP in the graph.
◇

COROLLARY 3.1. *ADMMA will terminate.*

Proof. Clearly the maximum length of a WMP is bounded by the number of processors P. By LEMMA 3.2 ADMMA will terminate in a maximum of P outer iterations.
◇

To attain our desired complexity bounds, we want to show that a WMP can not grow longer than a constant. To understand the behavior of this algorithm we begin with a few observation about regular meshes. Begin by looking at a regular partitioning of a 2D FE quadrilateral mesh. Figure 6 shows a 2D mesh and a partitioning with regular blocks of four $(2*2)$ and a particular processor order. This is just small enough to allow for a WMP to traverse the mesh indefinitely, but clearly a nine $(3*3)$ vertex partitions would break this WMP and only allow it *walk* around partition intersections. Note that the $(2*2)$ case would require just the right sequence of events to happen on all processors for this WMP to actually govern the run time of the algorithm. On a regular 3D FE mesh of hexahedra the WMP can *coil* around a line between four processors and the required partition size, using the same arguments as in the 2D case, would be five vertices on each side (or one more than the number of processors that share a processor interface line).

For irregular meshes one has to look at the mesh partitioning mechanism employed. Partitions on irregular meshes in scientific and engineering applications will generally attempt to reduce the number of edges cut (i.e. $|E^S|$) and balance the number of vertices on each partition (i.e. $|V_p| * p/n \approx 1$). We will assume that such a partitioner is in use and make a few general observations. First the partitions of such a mesh will tend to produce partitions in the shape of a hexagon in 2D for a large mesh with relatively large partitions. This is because the partitioner is trying to reduce the *surface* to *volume* ratio of each partition. These partitions are not likely to have skinny regions where a WMP could *jump* through the partition, and thus the WMP is relegated to following the lines of partition intersections. We do not present statistical or theoretical arguments as to the minimum partition size N that must be employed to bound the growth of a WMP for a given partitioning method; though clearly some constant N exists that, for a give FE mesh type and a given reasonable partitioning method, will bound the maximum WMP length by a constant. This constant is roughly the number of partitions that come *close* to each other at some point, an optimal partitioning of a large D dimensional mesh will produce partitioning in which $D+1$ partitions meet at any given *point*. Thus, when a high quality mesh partitioner is in use, we would expect to see the algorithm terminate in at most four iterations on adequately well partitioned and sized three dimensional FE meshes.

4 Numerical results

We present numerical experiments on an IBM SP. An extended version of the Finite Element Analysis Program (FEAP)[8], is used to generate out test problems and produce our graphics. We use ParMetis [6] to calculate our partitions, and PETSC [2] for our parallel programming and development environment. Our code is implemented in C++, FEAP is implemented in FORTRAN, PETSc and ParMetis are implemented in C. We want to show that our complexity analysis is indicative of the actual behavior of the algorithm with real (imperfect) mesh partitioners. Our experiments confirm our P-RAM complexity model is indicative of the performance one can expect with practical partitions on graphs of FE problems. Due to a lack of processors we are not able to investigate the asymptotics of our algorithm throughly.

Our experiments will be used to demonstrate that we do indeed see the behavior that our theory predicts. Additionly we will use numerical experiments to quantify lower bound on the number of vertices per processor that our algorithm requires before growth in the number of outer iterations is observed. We will use a parameterized mesh from solid mechanics for our test problem. This mesh is made of eight vertex hexahedral trilinear "brick" elements and is almost regular; the maximum degree Δ of any vertex is 26 in the associated graph. Figure 7 shows one mesh (13882 vertices). The other meshes that we test are of the same physical model but with different scales of discretization.

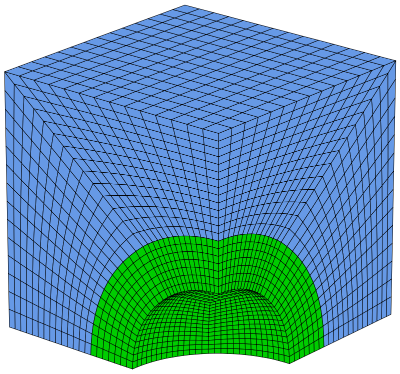

Figure 7: 5296 Vertex 3D FE mesh

We add synchronization to ADMMA on each processor by receiving *all* messages from neighboring processors in each iteration, to more conveniently measure the maximum length WMP that actually governs the number of outer iterations. Table 8 shows the results of the number of iterations required to calculate the MIS. Each case was run 10 times, as we do not believe that ParMetis is deterministic, but all 10 iteration counts were identical, thus it seems that this did not effect any of our results. A perfect partitioning of a large mesh with a large number of vertices per processor will result in $D + 1$ processors intersecting at a "point", and D partitions sharing a "line". If these meshes are optimal we can expect that the length of these lines (of partition boundaries) will be of approximately uniform length. The length of these lines required to halt the growth of WMPs is $D + 1$ vertices on an edge, as discussed in §3. If the approximate average size of each partition is that of a cube with this required edge length, then we would need about 64 vertices per partition to keep the length of a WMP from growing past 4. This assumes that we have a perfect mesh, which we do not, but none the less this analysis gives an approximate lower bound on the number of vertices that we need per processor to maintain our constant maximum WMP length.

Processors	8	16	24	32	40	48	56	64	72	80
427 vertices	3	3	3	3	4	4	4	4	6	6
1270	2	4	3	3	4	4	4	3	4	3
2821	3	3	4	3	3	3	4	3	4	4
5296	2	2	3	3	3	3	4	3	3	3
8911	3	3	4	3	4	3	4	3	3	3
13882	3	3	3	3	3	3	3	3	3	3

Figure 8: Average number of iterations

Figure 9 shows a graphic representation of this data for all partitions. The growth in iteration count for constant graph size is reminiscent of the polylogarithmic complexity of *flat* or vertex based random MIS algorithms [5]. Although ParMetis does not specify the ordering of processors, it is not likely to be very random. These results show that the largest number of vertices per processor that "broke" the estimate of our algorithms complexity bound is about 7 (6.5 average) and the smallest number of vertices per processor that stayed at our bound of 4 iterations was also about 7 (7.3 average). To demonstrate our claim of $O(1)$ P-RAM complexity we only require that there exists a bound N on the number of vertices per processor that is required to keep a WMP form growing beyond the region around a point where processors intersect. These experiments do not show any indication that that such a N does not exist. Additionally these experiments show that our bounds are quite pessimistic for the number of processors that we were able to use. This data suggests that we are far away from the asymptotics of this algorithm, that is, we need many more processors to have enough of the longest WMPs so that one consistently governs the number of outer iterations.

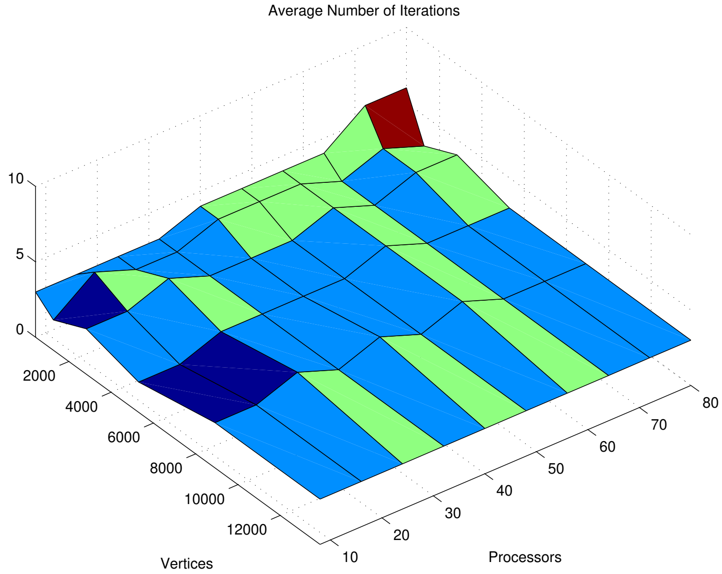

Figure 9: Average Iterations vs. number of processors and number of vertices

5 Conclusion

We have presented a new maximal independent set algorithm that, for graphs arising from finite element analysis, possesses optimal (i.e. $O(1)$ P-RAM complexity), if an adequate mesh partitioner is employed. The particular mesh partitions that we require for our complexity analysis have been shown to be attainable (based on numerical experiments using a publicly available mesh partitioner). That is with ParMetis and about a hundred vertices per processor, our algorithm terminates in a small (≤ 4) number of iterations. We thus feel that our method is of practical and potentially of theoretical interest. Our algorithm is novel in that it explicitly utilizes the partitionings that are freely available to stop the growth of monotonic paths which are responsible for the polylogarithmic complexity of *flat* or vertex based algorithms.

We have concentrated on the practical issues of our algorithm but have not fully explored the theoretical issues. Some areas of future work could be:

- Can this method provide for an improvement in the complexity bounds of more general graphs?
- Can the graph partitioning be incorporated into a complexity model of this method and maintain a theoretically *optimal* complexity bound? Some potential directions for such partitioners are
  - Geometric partitioners [4].
  - A level set partitioning algorithm.

Acknowledgments. We would like to thank Jim Demmel for his numerous suggestions and careful readings of this paper. This work is supported by DOE (grant No. W-7405-ENG-48), and we would like to thank Steve Ashby for his constant support of our efforts. We gratefully acknowledge Argonne National

Laboratory for the use of their IBM SP for the program development and numerical results presented in this paper. We would also like to thank R.L. Taylor at the University of California, Berkeley for his helpful comments, and for providing and supporting FEAP.

References

[1] Mark Adams. Heuristics for the automatic construction of coarse grids in multigrid solvers for finite element matrices. Technical Report UCB//CSD-98-994, University of California, Berkeley, 1998.

[2] S. Balay, W.D. Gropp, L. C. McInnes, and B.F. Smith. PETSc 2.0 users manual. Technical report, Argonne National Laboratory, 1996.

[3] D. Brelaz. New method to color the vertices of a graph. *Comm ACM, 22, pp 251-256*, 1979.

[4] John R. Gilbert, Gary L. Miller, and Shang-Hua Teng. Geometric mesh partitioning: implementation and experiments. Technical report, Xerox Palo Alto Research Center, 1994.

[5] Mark T. Jones and Paul E. Plassman. A parallel graph coloring heuristic. *SIAM J. Sci. Comput., Vol. 14 No. 3, pp. 654-669*, 1993.

[6] George Karypis and Kumar Vipin. Parallel multilevel k-way partitioning scheme for irregular graphs. *Supercomputing*, 1996.

[7] M. Luby. A simple parallel algorithm for the maximal independent set problem. *SIAM J. Comput., 4, pp. 1036-1053*, 1986.

[8] O.C. Zienkiewicz and R.L. Taylor. *The Finite Element Method, 4th ed.* McGraw-Hill, London, 1989.