Practical Multi-threaded Graph Coloring Algorithms for Shared Memory Architecture

Nandini Singhal, Sathya Peri, Subrahmanyam Kalyanasundaram
Department of Computer Science & Engineering
Indian Institute of Technology Hyderabad
{cs15mtech01004, sathya_p, subruk}@iith.ac.in

ABSTRACT

In this paper, we present multi-threaded algorithms for graph coloring suitable to the shared memory programming model. Initially, we describe shared memory implementations to the algorithms widely known in the literature like Jones Plassman graph coloring. Later, we propose new approaches to solve the problem of coloring using mutex locks while making sure that deadlocks do not occur. Using datasets from real world graphs, we evaluate the performance of all these algorithms on the Intel platform. We compare the performance of sequential graph coloring v/s our proposed approaches and analyze the speedup obtained against the existing algorithms from the literature. The results show that the speedup obtained by our proposed algorithms in terms of the time taken for coloring is consequential. We also provide a direction for future work towards improving the performance further in terms of different metrics.

CCS Concepts

•**Computing methodologies** → **Shared memory algorithms;** *Concurrent algorithms;* Distributed algorithms;

Keywords

Graph coloring; multi-threaded; shared memory; locks; barrier

1. INTRODUCTION

The Graph Coloring Problem pertains with attributing colors to the vertices of a simple graph such that no two adjacent vertices get the same color (also termed as vertex coloring). Proper coloring of an arbitrary graph using number of colors equal to its chromatic number is known to be an NP-hard problem. Thus, the primary goal in the graph coloring problem is to reduce the coloring time and minimize the number of colors used (ensuring proper coloring).

With the growing use of multi-core systems, hardware capability can be completely exploited with parallel algorithms. Each core can independently process a subtask and this can speed up the overall performance of the algorithm.

ICDCN '17, January 04-07, 2017, Hyderabad, India

DOI: http://dx.doi.org/10.1145/3007748.3018281

However, sequential algorithms act only on single core, albeit availability of multi-cores. The development of multi-core systems has been followed by the advent of shared memory programming paradigms like OpenMP and Pthreads, which are very easier to program. In this paper, we simulate using Pthreads for fine grained control over thread management.

The basic motivation behind this paper is to limit the number of threads being created at runtime (irrespective of the size of the graph to be colored). It is a very common practice in the literature dealing with parallel graph algorithms to create a thread corresponding to each vertex in the input graph. This is not practically feasible owing to the constraints on the resources (stack size, etc.) available. Also, increasing the number of threads beyond the hardware capacity does not lead to any improvement in the performance of the algorithm. To highlight this point, we have evaluated a parallel algorithm for increasing number of threads as shown in Figure 1. We see that with continuous increase in number of threads, the performance starts worsening. In this paper, we present algorithms for graph coloring which facilitate the user to input the number of threads to be acted upon depending on the hardware architecture (hardware threads, number of cores) available.

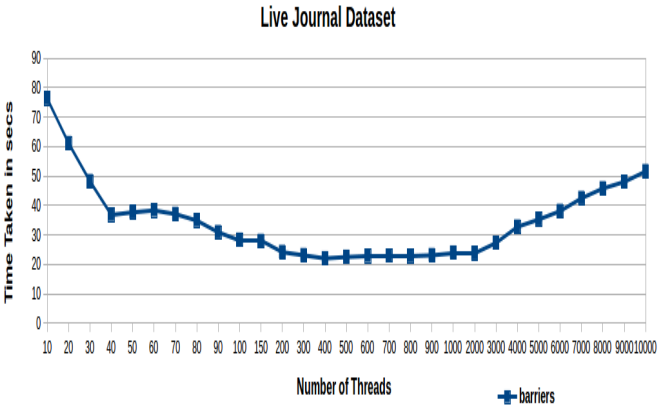

Figure 1: Time Taken in secs v/s Number of Threads

We look at four different approaches: (1) barrier synchronization; (2) jones plassman; (3) locking variants and (4) transactions. The first algorithm deals with the most widely studied technique of coloring using barrier synchronization [1, 2, 5]. A barrier is a command in the source code for a set of threads/processes which acts as a synchronization point for all of them. When the threads execute the barrier point, they must stop and cannot proceed until all remaining

threads/processes have reached the barrier. The literature about parallel graph coloring refers mainly to the algorithm using barrier synchronization. However, they do not prove the correctness of the algorithm. This is crucial because all the threads run in an asynchronous manner and the behaviour of a thread at a particular time instant cannot be determined. In this paper, we present a modified version of this algorithm in section 3 and give a proof of correctness of the algorithm. Subsequently, we also present a shared memory implementation of the Jones Plassman Algorithm. The later subsections puts forth new approaches for graph coloring using some standard locking techniques. We ensure that deadlocks do not occur. In Section 4, we then evaluate the performance of the presented algorithms against the sequential greedy coloring algorithm. We see that the performance of our proposed approach (using locks) outweighs the other existing algorithms on shared memory architecture. Finally, section 5 concludes and provides a direction of future work in this area.

2. BACKGROUND

2.1 Problem Definition

A graph G is represented as a pair (V, E) of a set of vertices V and a set of edges E. The edges are unordered pairs of the form $\{i, j\}$ where $i, j \in V$. Two vertices i and j are said to be adjacent if and only if $\{i, j\} \in E$ and non-adjacent otherwise. The degree of a vertex v is the number of vertices adjacent to v and is denoted by $\deg(v)$.

The Graph Coloring Problem:
A vertex coloring of a simple graph is an allotment of colors to the vertices such that no two adjacent vertices are assigned the same color. It is easy to see that any arbitrary simple graph can be colored with $\Delta + 1$ colors where Δ is the maximum degree of the simple graph.

2.2 Related Work

The problem of parallel graph coloring has been studied extensively, even on multi and many-core architectures [1–4]. Jones & Plassman [8] proposed a distributed graph coloring algorithm in which they process the vertices of a graph in a random order. The difficulty with this approach lies in identifying the most effective ordering of the vertices according to the graph in question. Hasenplaugh, et al. [6] proposed two ordering schemes which performs efficiently on the shared memory implementation of the Jones Plassman Algorithm. Gebremedhin & Manne [5] presented a basic parallel graph coloring algorithm using block partitioning. However, it can be seen that it is highly inefficient owing to the synchronization in each iteration of tentative coloring phase. Also, all the vertices with conflicting colors are colored sequentially. This leads to reduced parallelism in the algorithm where number of conflicts are high. Gebremedhin, Manne & Woods [4] propose several enhancement over [5] to reduce the number of conflicts by the use of graph partitioning packages. However, since the underlying algorithm itself does not completely exploit the parallelism, this algorithm does not fare too well.

Boman, et al. [1] presented a novel distributed graph coloring algorithm based on the previous notions, by improving the parallelism. This algorithm also assumed that the number of processors available would be significantly less as compared to the number of vertices in the graph. Hence it partitioned the input graph and assigned a subset of vertices to each processor. Çatalyürek, et al. [2] extend the previous notion for shared memory model. However, the algorithm create threads for each vertex in the input graph in each iteration of the algorithm. This increases the overhead significantly. Also, the paper does not provide a proof as to why the algorithm would terminate in a finite number of steps or would result in proper coloring of the graph eventually (without the use of locks for accessing shared memory).

2.3 System Model

In this paper, we assume that our system consists of n processors, accessed by p threads/processors that run in a completely asynchronous manner. Hence, we make no assumption about the relative speeds of the processors. We also assume that none of these processors and threads fails.

3. SOLUTION APPROACHES

In this section, we present various approaches suitable for dealing with the Graph Coloring problem on a shared memory programming model which means that threads communicate only by writing to and reading from the shared memory. We begin by assuming that all the vertices in the graph are assigned unique id's from $\{1, 2, \ldots, |V|\}$. Initially, the graph $G = (V, E)$ has been preprocessed by partitioning it uniformly into p partitions where p is the number of threads. Then the vertices with their corresponding id's in $\{1, \ldots, |V|/p\}$ are assigned to partition V_1, $\{|V|/p+1, \ldots, 2|V|/p\}$ get assigned to partition V_2 and so on until V_p. It seems only fair that the graph partitioning preprocessing time be included in the overall time taken for graph coloring. Hence, using this random heuristic based partitioning helps us bring down the overall time taken for coloring. Therefore, we do not use a graph partitioning software for minimizing the crossing edges between various partitions because it effectively increases the coloring time.

The vertices in each partition/block can be classified into:-internal vertices (whose all the neighbouring vertices lie in the same partition) and boundary vertices (those who have neighbours belonging to other partitions). Each thread is responsible for proper coloring of all the vertices in its partition. The subsequent subsections present algorithms using the First Fit Coloring strategy, which assigns each vertex the least legal color available. All the algorithms can be easily adapted to other coloring strategies like Largest Degree First, etc.

3.1 Using Barrier Synchronization

The barrier synchronization based algorithm has been widely explored in the literature [1, 2, 5]. However, the major difference in the algorithm presented here is that [2] has not explicitly used barriers for synchronization. We use barriers because it is more efficient than creating new threads as in [2]. As discussed before, a barrier is a synchronization point amongst threads. This algorithm has two phases: tentative coloring and conflict detection phase. Each thread maintains a copy of colors assigned to its neighbours locally in *ForbiddenColors* List. In the first phase, a thread assigns a color to all the vertices in its partition by taking into account all its previously colored neighbours from the local copy. However, it might result in two threads simul-

taneously coloring the vertices adjacent to each other with the same color. Hence, in the second phase, each thread T_i checks whether the vertices in V_i have been assigned valid colors by comparing the color of each vertex against all its neighbours. If any vertex and its neighbor have the same color, then the vertex in the partition with lower partition id is marked for recoloring.

The first and second phases are synchronized by a barrier that ensures that all the p threads start their execution at the same instant. This is crucial because if a thread were still coloring while other tries to detect conflicts, then this can lead to false detection eventually leading to improper coloring. The algorithm has been described in Algorithm 1.

Algorithm 1 Using Barrier

```
1: Input: p ← no of threads
2: uniform partitioning of V into V_1, V_2, ..., V_p in increasing order of vertex ids
3: m ← maximum degree of graph
4: procedure ParallelGraphColoring(G = (V, E))
5:     for all thread T_i | i ∈ {1, ..., p} do
6:         Identify boundary vertices of partition i
7:         Initialise TotalColors[m + 1] ← {0, 1, ..., m}
8:         for v ∈ V_i do
9:             Create List v.ForbiddenColors
10:            Initialise v.ForbiddenColors to −1
11:        end for
12:        U_i ← V_i
13:        while U_i ≠ ∅ do
14:            for each v ∈ U_i do                          ▷ Phase 1 starts
15:                Assign color(v) ← min{TotalColors − v.ForbiddenColors}
16:                for each u ∈ adjacent(v) | u ∈ V_i do
17:                    Update color(v) in u.ForbiddenColors
18:                end for
19:            end for
20:            Wait for all threads to reach here          ▷ Using barrier
21:            R_i ← ∅                                     ▷ Phase 2 starts
22:            for each v ∈ U_i | v is a boundary vertex in U_i do
23:                for each u ∈ adjacent(v) | u ∉ V_i do
24:                    Update color(u) in v.ForbiddenColors
25:                    if color(u) = color(v) | u ∈ V_j and i < j then
26:                        R_i ← R_i ∪ {u}
27:                    end if
28:                end for
29:            end for
30:            U_i ← R_i
31:            Wait for all threads to reach here          ▷ Using barrier
32:        end while
33:    end for
34: end procedure
```

Lemma 1: *Barrier Synchronization Algorithm results in proper coloring of the graph.*

Proof: Let us prove by contradiction. So, we assume that the barrier synchronization algorithm does not result in proper coloring of the graph meaning that two adjacent vertices in the graph have the same color at the end of the algorithm.

Each round/iteration of the algorithm consists of 2 phases: coloring and conflict detection phase respectively. We denote the coloring phase of i^{th} iteration as $i.1$ and conflict detection phase of i^{th} iteration as $i.2$.

Without loss of generality, let us say that a vertex v_x which is adjacent to a vertex v_y, gets colored c in round $i.1$ and vertex v_y gets assigned the same color in round $j.1$, both belonging to different partitions and i ≤ j.

$$color^{i.1}(v_x) = color^{j.1}(v_y) = c \text{ where } v_x, v_y \text{ belong to different partitions}$$

Now there are two possibilities as follows:

a) v_x was assigned color c in round $(j-1).1$: In this case, in round $(j-1).2$, v_x and v_y would be identified with same color and the vertex in the lower partition id would get recolored in round $j.1$. Hence either v_x or v_y would have a color different from c. Also since $i \leq j$, this means that both vertices get properly colored. Hence this is a contradiction to our initial assumption.

b) v_x was assigned a color different from c in round $(j-1).1$: In this case, v_x got recolored back to color c in round $j.1$, then a conflict will be detected in round $j.2$ and it will be resolved in round $(j+1).1$. Hence it again contradicts our assumption.

Thus we can conclude that eventually all the conflicts get resolved and no two adjacent vertices get assigned to a same color. □

Lemma 2: *Barrier Algorithm terminates after a maximum of $p+1$ iterations.*

Proof: The partitions of the graph are V_1, V_2, ..., V_p. In each round, a vertex in V_i is recolored if it has a conflict with a vertex in V_{i+1}, ..., V_p.

In the 1^{st} round, at least all vertices of V_p get properly colored and all conflicts of the vertices in V_{p-1} with vertices of V_p are identified, which are resolved in the next round.
Similarly, in the 2^{nd} round, all vertices of V_{p-1} get properly colored and all conflicts in V_{p-2} with V_{p-1} are identified, which are resolved in the subsequent round.

Thus by induction, it is easy to see that after $p+1$ iterations, V_1 gets properly colored. Also, the maximum number of times a vertex in partition V_i gets recolored is $(p-i)$. □

3.2 Jones Plassman Algorithm

The Jones Plassman algorithm is a very popularly known distributed graph coloring algorithm. In this subsection, we present a shared memory implementation of the Jones Plassman Algorithm. Initially, each vertex v in the input graph is assigned a distinct random number $\rho(v)$. This is helpful in ordering the vertex amongst its neighbours by computing local data structures, *n-wait*(v) and *send-list*(v). The data structure *n-wait*(v) maintains the count of those neighbours $u \in adj(v)$ which have $\rho(u)$ greater than $\rho(v)$. This implies that vertex v should be colored after coloring of these vertices. Similarly, *send-list*(v) keeps those neighbours $u \in adj(v)$ which have $\rho(u)$ smaller than $\rho(v)$, meaning that once vertex v gets colored, the vertices in *send-list*(v) can be colored.

Algorithm 2 Jones Plassman

```
1: Input: p ← no of threads
2: Assign ρ(v) ∀ v ∈ V                          ▷ distinct, random number
3: procedure ParallelGraphColoring(G = (V, E))
4:     for all thread T_i | i ∈ {1, ..., p} do
5:         Identify boundary vertices of partition i
6:         Initialise TotalColors[m + 1] ← {0, 1, ..., m}
7:         color_list ← ∅
8:         Initialise Concurrent List L_i           ▷ indicating all adjacent vertices of all vertices in T_i
```

Since the vertices of different partitions in the graph communicate by exchanging messages in the distributed algorithm, we achieve the same in the shared memory by using a concurrent list data structure. For each thread T_i, a corresponding Concurrent Set based List L_i is initialised. The

algorithm proceeds in iterations until all the vertices in partition local to each thread get colored. In each iteration, all the vertices with their *n-wait* as 0, say P, are colored indicating their turn in the ordering amongst the set of neighbours. As a result of this, the vertices which have been waiting for P to get colored (*send-list(P)*) have to be informed. So, P is added to the Concurrent List of the corresponding threads (those partition which contains vertices in *send-list(P)*).

```
 9:        for each v ∈ V_i | v is a boundary vertex do
10:            n-wait(v) ← 0
11:            send-list(v) ← ∅
12:            for each u ∈ adjacent(v) | u is a boundary vertex do
13:                if ρ(u) > ρ(v) then
14:                    n-wait(v) ← n-wait(v) + 1
15:                else
16:                    send-list(v) ← send-list(v) ∪ {u}
17:                end if
18:            end for
19:            if n-wait(v) = 0 then
20:                color-list ← color-list ∪ {v}
21:            end if
22:        end for
23:        for v ∈ V_i do
24:            Create List v.ForbiddenColors
25:            Initialise v.ForbiddenColors to −1
26:        end for
27:        Invoke Color(color-list)
28:        Invoke Append_Concurrent_List(color-list)
29:        n-colored ← | color-list |
30:        color_list ← ∅
31:        while n-colored < no of boundary vertices in V_i do
32:            Iterate L_i to get v
33:            for each u ∈ adjacent(v) | u ∈ V_i and u is a boundary
    vertex do
34:                n-wait(u) ← n-wait(u) − 1
35:                Update color(v) in u.ForbiddenColors
36:                if n-wait(u) = 0 then
37:                    color-list ← color-list ∪ {u}
38:                end if
39:            end for
40:            Invoke Color(color-list)
41:            Invoke Append_Concurrent_List(color-list)
42:            n-colored ← n-colored + | color-list |
43:            color_list ← ∅
44:        end while
45:        for each v ∈ V_i | v is a internal vertex in V_i do
46:            color(v) ← min{TotalColors − color(adjacent(v))}
47:        end for
48:    end for
49: end procedure
50: procedure Append_Concurrent_List(color-list)
51:    for each v ∈ color-list do
52:        temp_set ← ∅
53:        for each w ∈ send-list(v) do
54:            temp_set ← temp_set ∪ Partition_id(w)
55:        end for
56:        for each k ∈ temp_set do
57:            L_k.insert(v, color(v))
58:        end for
59:    end for
60: end procedure
61: procedure Color(color-list)
62:    for each v ∈ color-list do
63:        color(v) ← min{TotalColors − v.ForbiddenColors}
64:        for each u ∈ adjacent(v) do
65:            Update color(v) in u.ForbiddenColors
66:        end for
67:    end for
68: end procedure
```

A thread keeps iterating through its concurrent set L_i to check if any other vertex (which it has been waiting on) has been colored. If a new insertion happens in the List then the corresponding adjacent vertex's *n-wait* is decreased by 1. It is to be noted that the Concurrent Set based Linked List with functions for scan and append can be implemented using mutexes or atomic CAS operations. Here, delete operation is not required. Also, the scan function simply checks if the next pointer is not NULL. If not NULL, it indicates a new element has been inserted in the List. The complete algorithm is described in Algorithm 2.

Lemma 3: *At least one vertex at every instant in JP Algorithm has its n-wait as 0.* □

Lemma 4: *Jones Plassman Algorithm results in proper coloring of the graph.*

Proof: This can be seen in a straightforward manner. Improper coloring can only happen if two adjacent vertices of different partitions are colored at the same time by different threads wherein they both read the same colors of the neighbours and assign the adjacent vertices to the same color. Now, as can be seen from Algorithm 2, only vertices with their n-wait as 0 can be colored at a particular time instant. Thus, it is to be shown that no two adjacent vertices have their *n-wait* as 0 at the same time instant. It is known that all vertices in the graph are assigned a distinct, random number. Hence because of lines 15-18 in Algorithm 2, all adjacent vertices would be waiting on one another. This proves that *n-wait* of two adjacent vertices cannot become 0 at the same instant and thus ensures legal coloring. □

Lemma 5: *Jones Plassman Algorithm terminates after a finite iterations.*

Proof: In case of a complete graph of n vertices, each vertex in a partition sends a message to each of the concurrent thread. Say a vertex in the 1st partition is currently having n-wait as 0. Hence it colors itself and writes to Concurrent List of all other p-1 threads. Now all vertices decrease their n-wait by 1. At least one vertex now has its n-wait to be 0. Hence it again colors itself and sends messages to all other p-1 threads. Since each partition contains at max n/p vertices, the maximum number of messages passed by a single thread $= (p-1) * n/p = O(n)$. Now since there are p threads, the total number of messages are upper bounded by $O(n * p)$. □

3.3 Using Mutex Locks

The motivation behind using an alternative to the barrier synchronization approach presented in previous subsection lies in the fact that, since all the threads get synchronized at two points (Lines 20 & 31 in Algorithm 1) in each iteration, there is an unfavourable impact on the performance of the algorithm. The overall goal is to avoid global synchronization of threads and let them run independently. We present algorithms based on the locking of the graph vertices. With locks, coloring the vertex becomes a critical section and a thread can only enter the critical section when it has acquired the lock.

3.3.1 Coarse Grained Locking

With Coarse Grained Locking, there exists a big lock on the complete list of boundary vertices. This implies that at any point, a thread must acquire a lock on this list to color any boundary vertex.

3.3.2 Fine Grained Locking

Coarse Grained Locking can be improvised on by making use of fine grained locks wherein each vertex has a corresponding lock. A thread wishing to color a boundary vertex has to obtain locks on all the neighboring vertices of that

boundary vertex. However, to avoid deadlock, a global ordering of vertices is maintained (based on their vertex ids) and vertices acquire locks in the respective order. The complete algorithm has been described in Algorithm 3.

Algorithm 3 Using Fine Grained Locks

```
1: Input: p ← no of threads
2: procedure ParallelGraphColoring(G = (V, E))
3:     for all thread T_i | i ∈ {1,...,p} do
4:         Identify boundary vertices in V_i
5:         Initialise TotalColors[m + 1] = {0, 1, ...., m}
6:         for each v ∈ V_i | v is a internal vertex in V_i do
7:             color(v) ← min{TotalColors − color(adjacent(v)}
8:         end for
9:         for each v ∈ V_i | v is a boundary vertex in V_i do
10:            List A_i ← adj(v) | adj(v) is a boundary vertex
11:            A_i ← A_i ∪ {v}
12:            Lock all vertices in A_i in increasing order of vertex ids
13:            color(v) ← min{TotalColors − color(adjacent(v)}
14:            Unlock all vertices in A_i
15:        end for
16:    end for
17: end procedure
```

3.3.3 Cutting Waiting Chains

The drawback of using fine grained locks as described in the previous subsubsection is that in case of a chain graph, each vertex could be waiting on its adjacent vertex to acquire the locks. These waiting chains can worsen in denser graphs. This leads to a motivation to develop an algorithm for cutting long transitive waiting chains. Here, we present a variant of the Anderson, et al. [7] algorithm.

The idea here is to maintain a table data structure of boolean fields with number of rows equal to the number of partitions + 1 of the graph and number of columns equal to the number of partitions. A request for coloring a vertex v can be made by positioning itself in the column indexed by *partition_id(v)*, of a particular row. Each true entry present in a row in the table corresponds to the request positioned in that row. Each column is indexed by the partition id's of the graph. The advantage to using this approach is that the size of the table is very small (in order of number of partitions) as compared to the size of input graph. The requests are fulfilled in the increasing order of rows starting from the first row of the table. To maintain validity, we need to ensure that at any instant only one request can be placed by a particular thread in some row.

To place a request for coloring a vertex v of a particular partition in a particular row, the thread needs to check the existing requests placed in the corresponding row. If all vertices corresponding to the requests placed in that row are not adjacent to v in the graph, then the request can be placed in the corresponding row. For this, a thread must know about the vertices which are being colored corresponding to the partition id's whose entry is true in the respective row. To achieve this, an atomic array of the size of the number of partitions is maintained. Each element of the array indexed by a partition id corresponds to the vertex being colored from that partition. This information is updated whenever a new request is placed.

The terminology used in the pseudo code is consistent with [7]. The shared memory field *head* is an atomic field which points to the currently enabled row in the table. Two more atomic arrays *enabled* and *numReq* are maintained. Each row in the table has a corresponding entry in these

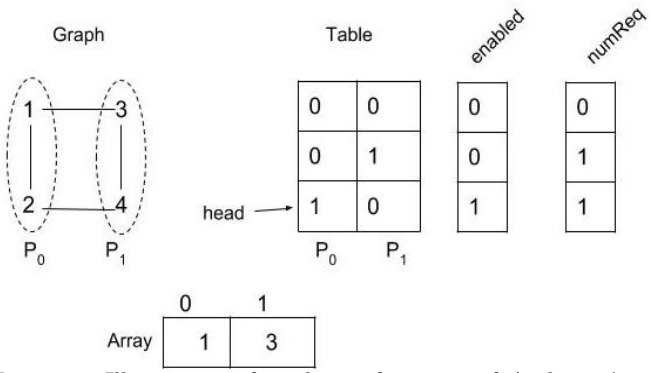

Figure 2: Illustration of working of variant of Anderson's algorithm

arrays indexed by the row number. A true entry in the enabled array indicates that the requests positioned in the corresponding row are fulfilled. At a particular instant, only one row can be enabled. *numReq* array maintains the count of the number of requests in each row. The last request in a row to be fulfilled, marks the next row to be *enabled*. Locks are needed to avoid concurrent changes to a row. Hence the algorithm proceeds in hand-over-hand row wise locking. This prevents deadlocks and allows fine grained concurrent access over the table data structure. Figure 2 illustrates the working of the algorithm when vertices 1 and 3 of partitions P_0 and P_1 are being colored concurrently. This can be identified by atomic *VertexFromPartition* array. It can be seen that the request for vertex 1 is placed in first row and vertex 2 in second row respectively. The first row of the table is referenced by *head* which corresponds to *enabled* being set to true. The number of requests placed in a given row is stored in *numReq*. Algorithm 4 describes the complete pseudo-code of the algorithm.

Algorithm 4 Anderson's Variant

```
1: Input: p ← no of threads
2: A ← Atomic array of p indices
3: Initialise a table data structure of boolean fields with #rows =
   p + 1, #cols = p
4: procedure ParallelGraphColoring(G = (V, E))
5:     for all thread T_i | i ∈ {1,...,p} do
6:         Identify boundary vertices in V_i
7:         Initialise TotalColors[m + 1] = {0, 1, ...., m}
8:         for each v ∈ V_i | v is a internal vertex in V_i do
9:             color(v) ← min{TotalColors − color(adjacent(v)}
10:        end for
11:        for each v ∈ V_i | v is a boundary vertex in V_i do
12:            Invoke request_table(v, i)
13:            color(v) ← min{TotalColors − color(adjacent(v)}
14:            Invoke release_table(v, i)
15:        end for
16:    end for
17: end procedure
18: procedure request_table(v, i)
19:    Update A[i] ← v
20:    Acquire a lock on the head row of the table
21:    start ← head
22:    while true do
23:        Initialise List L ← ∅
24:        for each true entry ∈ table[start][k] do where 1 ≤ k ≤ p
25:            List L ← A[k] ∪ L
26:        end for
27:        flag ← false
28:        for each w ∈ L do
29:            if w is adjacent to v then
```

```
30:                 flag ← true
31:             end if
32:         end for
33:         if flag = false then
34:             Goto line 43
35:         end if
36:         next ← start + 1
37:         Acquire a lock on the next row of the table
38:         Release lock on the start row of the table
39:         start ← next
40:     end while
41:     Set table[start][i] ← true
42:     numReq[start]++
43:     Release lock on the start row of the table
44:     Wait until enabled[start] is set to true
45: end procedure
46: procedure RELEASE_TABLE(v, i)
47:     Acquire a lock on the start row of the table
48:     Set table[start][i] ← false
49:     numReq[start]−−
50:     if numReq[start] = 0 then
51:         next ← start + 1
52:         enabled[start] ← false
53:         head ← next
54:         enabled[next] ← true
55:     end if
56:     Release lock on the start row of the table
57: end procedure
```

At any instant, only one request for coloring a vertex can be placed from each partition in this concurrent data structure. In the case of a complete graph, each request would be placed in a different row. Also since all the requests are fulfilled in the increasing order of the rows, there is no request that cannot be placed on its arrival. Hence the algorithm terminates when all the requests have been enabled. It can be seen that in case of a chain graph over n vertices, the length of the waiting chain equals to 2 only. Every alternate vertex can be placed in the same row of the table. This is an improvement over fine grained locking where the length of the waiting chain could have been n.

3.3.4 Maximal Independent Sets of subgraphs

It is commonly known that computing a Maximal Independent set of a graph is an NP-Complete problem. In this subsection, we present an algorithm for graph coloring which maintains small subgraph of the original graph and computes the MIS on it. The keypoint is that at any instant, the maximum of vertices in the subgraph is equal to the number of partitions. Since the size of the subgraph is very small as compared to the input graph; the algorithm is expected to fare well practically. Whenever a request for coloring a vertex v is made by a thread, a node is added to the subgraph G' corresponding to the thread's partition. If v is adjacent to any vertex in G, then the corresponding edge is added to vertex's partition node in G', if present. The algorithm proceeds in iterations until all the vertices in its partition get colored. In each step, an MIS is identified and those vertices are marked as *active*, which means that they can be colored. Furthermore, after a vertex has been colored, it is removed from the subgraph along with its corresponding edges. To avoid concurrent accesses to the shared subgraph G', a coarse mutex lock is used. The pseudo code is described in Algorithm 5.

Algorithm 5 MIS in subgraphs

```
1: Input: p ← no of threads
2: uniform random partitioning of V in V_1, V_2, . . . ,V_p
3: Initialise an empty graph data structure G'
4: procedure ParallelGraphColoring(G = (V, E))
5:     for all thread T_i | i ∈ {1,...,p} do
6:         Identify boundary vertices in V_i
7:         Initialise TotalColors[m + 1] = {0, 1, ...., m}
8:         for each v ∈ V_i | v is a internal vertex in V_i do
9:             color(v) ← min{ TotalColors − color(adjacent(v)}
10:        end for
11:        for each v ∈ V_i | v is a boundary vertex in V_i do
12:            Invoke request_graph(v, i)
13:            color(v) ← min{ TotalColors − color(adjacent(v)}
14:            Invoke release_graph(v, i)
15:        end for
16:    end for
17: end procedure
18: procedure REQUEST_GRAPH(v, i)
19:     Acquire a lock on the graph G'
20:     Add a vertex i to G' and mark it inactive
21:     Add edges from i to respective vertices ∈ G' and vice versa
22:     if degree of i = 0 then
23:         Mark i as active
24:     end if
25:     Release lock on graph G'
26:     Wait until vertex i becomes active
27: end procedure
28: procedure RELEASE_GRAPH(v, i)
29:     Acquire a lock on the graph G'
30:     Remove vertex i from G' and all corresponding edges
31:     Identify MIS from the set of inactive vertices and mark active
32:     Release lock on G'
33: end procedure
```

3.4 Using Transactions

A transaction is a piece of code which executes atomically. Since the internal vertices are colored without any interaction amongst threads, they can be colored without creating any transaction. However, each boundary vertex has to be colored by creating a transaction. A *read* operation is performed for reading the colors of all the adjacent vertices and finally a *write* is invoked for assigning a valid color to the boundary vertex. If at any point, an operation fails, the transaction has to be restarted. Once the transaction commits, this implies that the vertex has been colored (by writing its color to shared memory graph). This algorithm ensures proper coloring of the graph. We simulate this using Basic Timestamp Ordering (BTO) Protocol. This has been described in Algorithm 6.

4. SIMULATION RESULTS & ANALYSIS

We performed our tests on 24 core Intel Xeon server (X5675) running at 3.07 GHz core frequency. Each core supports 6 hardware threads, clocked at 1600 MHz. In the experiments conducted, the time taken for coloring the graph in the multi-threaded version includes the time taken for partitioning of graph as well. However, time taken for coloring in all versions (sequential & parallel) excludes the time taken to read the graph input. Each data point is obtained after averaging for 10 iterations. To test the performance of the algorithms, we have used real world graph, Live Journal from SNAP [9]. We have evaluated for two metrics: Time Taken to color the graph and Number of Colors Used. We have tested it for all algorithms in previous section by varying the number of threads in the range 1-1000 and noted it for the best result.

As can be clearly observed from the performance results, the barrier synchronization and Jones Plassman Algorithm do not fare well and are not comparable to the sequential coloring. On the other hand, locks and transactions seem to perform fairly well in terms of time taken for coloring maintaining a reasonable number of colors used. We observe that fine grained locking performs significantly better

Algorithm 6 Using Transactions

```
1: Input: p ← no of threads
2: Declare color(v) ∀v ∈ V in shared memory
3: aborts ← 0
4: Initialise Protocol (BTO/SGT)
5: procedure ParallelGraphColoring(G = (V, E))
6:     for all thread T_i | i ∈ {1,...,p} do
7:         Identify boundary vertices in V_i
8:         Initialise TotalColors[m + 1] = {0, 1, ...., m}
9:         for each v ∈ V_i | v is a internal vertex in V_i do
10:            color(v) ← min{TotalColors − color(adjacent(v)}
11:        end for
12:        for each v ∈ V_i | v is a boundary vertex in V_i do
13:            Begin Transaction
14:            List C ← read color(adj(v))
15:            if read fails then
16:                Abort transaction & goto line 15
17:            end if
18:            write color(v) ← min{TotalColors − C}
19:            try_commit() transaction
20:            if try_commit() fails then
21:                Increment aborts & goto line 15
22:            end if
23:        end for
24:    end for
25: end procedure
```

as compared to the sequential coloring. It is important to realise that Jones Plassman Algorithm does not fare well in terms of the time taken for coloring, partly because of the inefficient random ordering assigned to vertices. Hence even though it uses a reasonable number of colors, it is not practically feasible.

In the literature, there exists many ordering schemes which can further reduce the number of colors used such as Largest Degree First, Saturation Degree, etc [10]. It is important to note that since these orderings require some sorting of the vertices to order them, these will incur additional cost in terms of time taken. For parallel graph coloring, such heuristics can be used to order the vertices of each partition. Hence if such techniques are employed, they will lead to a rise in the time taken proportionally amongst all the algorithms (including the greedy sequential one). Thus, there will be no impact on the relative performance of the algorithms employed for a different ordering heuristic.

Table 1: Results of Live Journal Dataset

Algorithm	**#threads**	**Time Taken (secs)**	**#colors used**
Fine Grained Locks	70	6.18	334
Transactions (BTO)	200	8.26	335
Sequential algorithm	1	13.86	334
Coarse grained locks	100	17.75	333
Maximal Independent Set	2	18.36	336
Anderson's variant	14	19.26	335
Barrier synchronization	400	21.99	334
Jones Plassman	40	64954	334

5. CONCLUSION & FUTURE WORK

We have presented parallel algorithms for graph coloring suitable to the shared memory programming model. We have looked into the most commonly used approach for coloring using barrier synchronization and Jones Plassman Algorithm. We have also proposed new approaches using locks. Using the SNAP dataset, we evaluated the performance of the algorithms on the Intel platform. The results show that the improvement is noteworthy. This gives a motivation that the overhead of locking and unlocking operations is less and they do scale well with increasing number of threads as compared to the existing approaches.

We intend to test these algorithms for other types of graphs including dense ones. It seems that the algorithm can be improved by exploring pushing ahead of requests in the table used in Anderson's algorithm. Also these locking ideas can be extended for specific categories of graphs such as trees, star, etc. Furthermore, cutting the waiting chains caused by fine grained locking in graphs is an active research problem.

6. REFERENCES

[1] Erik G Boman, Doruk Bozdağ, Umit Catalyurek, Assefaw H Gebremedhin, and Fredrik Manne. A scalable parallel graph coloring algorithm for distributed memory computers. In *Euro-Par 2005 Parallel Processing*, pages 241–251. Springer, 2005.

[2] Ümit V. Çatalyürek, John Feo, Assefaw Hadish Gebremedhin, Mahantesh Halappanavar, and Alex Pothen. Graph coloring algorithms for multi-core and massively multithreaded architectures. *Parallel Computing*, 38(10-11):576–594, 2012.

[3] Mehmet Deveci, Erik G. Boman, Karen D. Devine, and Sivasankaran Rajamanickam. Parallel graph coloring for manycore architectures. In *2016 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2016, Chicago, IL, USA, May 23-27, 2016*, pages 892–901, 2016.

[4] Assefaw H Gebremedhin, Fredrik Manne, and Tom Woods. Speeding up parallel graph coloring. In *Applied Parallel Computing. State of the Art in Scientific Computing*, pages 1079–1088. Springer, 2006.

[5] Assefaw Hadish Gebremedhin and Fredrik Manne. Scalable parallel graph coloring algorithms. *Concurrency - Practice and Experience*, 12(12):1131–1146, 2000.

[6] William Hasenplaugh, Tim Kaler, Tao B. Schardl, and Charles E. Leiserson. Ordering heuristics for parallel graph coloring. In *26th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA '14, Prague, Czech Republic - June 23 - 25, 2014*, pages 166–177, 2014.

[7] Catherine E. Jarrett, Bryan C. Ward, and James H. Anderson. A contention-sensitive fine-grained locking protocol for multiprocessor real-time systems. In *Proceedings of the 23rd International Conference on Real Time and Networks Systems, RTNS 2015, Lille, France, November 4-6, 2015*, pages 3–12, 2015.

[8] Mark T. Jones and Paul E. Plassmann. A parallel graph coloring heuristic. *SIAM J. Scientific Computing*, 14(3):654–669, 1993.

[9] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, June 2014.

[10] Md Mostofa Ali Patwary, Assefaw H Gebremedhin, and Alex Pothen. New multithreaded ordering and coloring algorithms for multicore architectures. In *Euro-Par 2011 Parallel Processing*, pages 250–262. Springer, 2011.