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

ICDCN AADDA 2017

Outline of the Presentation

Outline of the Presentation

How do you parallelise an algorithm?

- Decomposing an Algorithm into independent tasks
- Distributing the parts as tasks which are worked on by multiple processes simultaneously
- Coordinating work and communications of those processes i.e. called as Synchronization

Here, synchronization is being achieved on Shared Memory Model

How do you parallelise an algorithm?

- Decomposing an Algorithm into independent tasks
- Distributing the parts as tasks which are worked on by multiple processes simultaneously
- Coordinating work and communications of those processes i.e. called as Synchronization

Here, synchronization is being achieved on Shared Memory Model

Parallelising Graph Coloring Algorithm

Problem Statement

Given a simple graph $G = (V, E)$.

Assign colors to the vertices of the graph such that no two adjacent vertices are assigned the same color.

Outline

Algorithm 1 - Jones Plassman Algorithm [SIAM1993]

- Input: $G = (V, E)$
- Assign a random priority to each vertex given by $\rho(v)$
- For each vertex v,
  1. vertices with priority less than it are its predecessors
  2. vertices with priority greater than it would be its successors
- Each vertex maintains a *count* for all its neighbours which are its predecessors
- A vertex gets colored when its *count* equals 0
- When a vertex gets colored, it informs all its successors which are its neighbours to decrease their *count* by 1.

Algorithm 1 - Jones Plassman Algorithm [SIAM1993]

- Input: $G = (V, E)$
- Assign a random priority to each vertex given by $\rho(v)$
- For each vertex v,
  1. vertices with priority less than it are its predecessors
  2. vertices with priority greater than it would be its successors
- Each vertex maintains a *count* for all its neighbours which are its predecessors
- A vertex gets colored when its *count* equals 0
- When a vertex gets colored, it informs all its successors which are its neighbours to decrease their *count* by 1.

Algorithm 1 - Jones Plassman Algorithm [SIAM1993]

- Input: $G = (V, E)$
- Assign a random priority to each vertex given by $\rho(v)$
- For each vertex v,
  1. vertices with priority less than it are its predecessors
  2. vertices with priority greater than it would be its successors
- Each vertex maintains a *count* for all its neighbours which are its predecessors
- A vertex gets colored when its *count* equals 0
- When a vertex gets colored, it informs all its successors which are its neighbours to decrease their *count* by 1.

Algorithm 1 - Jones Plassman Algorithm [SIAM1993]

- Input: $G = (V, E)$
- Assign a random priority to each vertex given by $\rho(v)$
- For each vertex v,
  1. vertices with priority less than it are its predecessors
  2. vertices with priority greater than it would be its successors
- Each vertex maintains a *count* for all its neighbours which are its predecessors
- A vertex gets colored when its *count* equals 0
- When a vertex gets colored, it informs all its successors which are its neighbours to decrease their *count* by 1.

Algorithm 1 - Jones Plassman Algorithm [SIAM1993]

- Input: $G = (V, E)$
- Assign a random priority to each vertex given by $\rho(v)$
- For each vertex v,
  1. vertices with priority less than it are its predecessors
  2. vertices with priority greater than it would be its successors
- Each vertex maintains a *count* for all its neighbours which are its predecessors
- A vertex gets colored when its *count* equals 0
- When a vertex gets colored, it informs all its successors which are its neighbours to decrease their *count* by 1.

Algorithm 1 - Jones Plassman Algorithm [SIAM1993]

- Input: $G = (V, E)$
- Assign a random priority to each vertex given by $\rho(v)$
- For each vertex v,
  1. vertices with priority less than it are its predecessors
  2. vertices with priority greater than it would be its successors
- Each vertex maintains a *count* for all its neighbours which are its predecessors
- A vertex gets colored when its *count* equals 0
- When a vertex gets colored, it informs all its successors which are its neighbours to decrease their *count* by 1.

Algorithm 1 - Drawback of Jones Plassman

- Uses random function for assigning the priorities
- If the input graph is a chain of vertices and the numbering of vertices correspond to their priorities, then there is no parallelism exhibited by this algorithm

Algorithm 1 - Drawback of Jones Plassman

- Uses random function for assigning the priorities
- If the input graph is a chain of vertices and the numbering of vertices correspond to their priorities, then there is no parallelism exhibited by this algorithm

Algorithm 2 - Block Partitioning based Algorithm [Gebremedhin2000]

- Each thread is responsible for proper coloring of vertices in its partition
    1. Tentative coloring of vertices
- Synchronization of all threads
- Sequential coloring of conflicting vertices

- Downside - not much parallelism exploited

Algorithm 2 - Block Partitioning based Algorithm [Gebremedhin2000]

- Each thread is responsible for proper coloring of vertices in its partition
  1. Tentative coloring of vertices
- Synchronization of all threads
- Sequential coloring of conflicting vertices
- Downside - not much parallelism exploited

Algorithm 2 - Block Partitioning based Algorithm [Gebremedhin2000]

- Each thread is responsible for proper coloring of vertices in its partition
    1. Tentative coloring of vertices
- Synchronization of all threads
- Sequential coloring of conflicting vertices

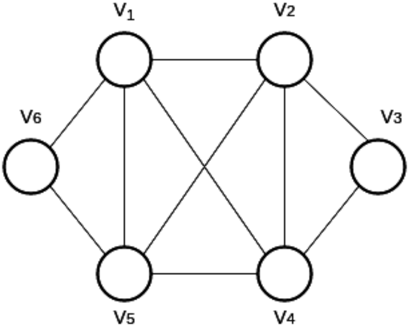
- Downside - not much parallelism exploited

Algorithm 2 - Block Partitioning based Algorithm [Gebremedhin2000]

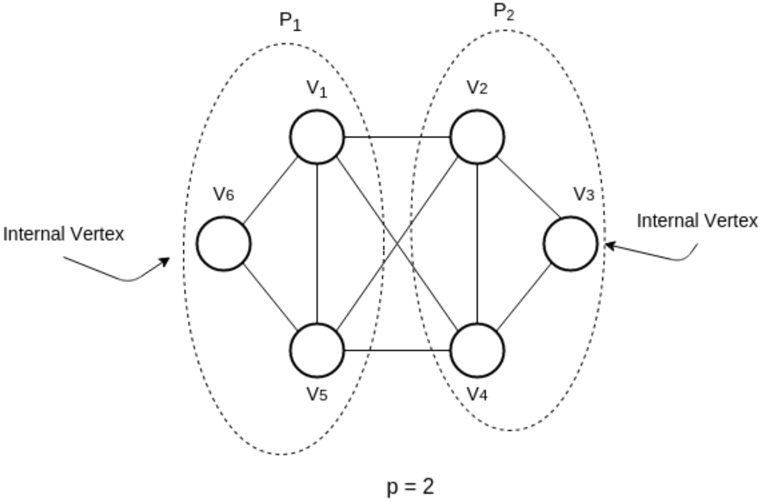
- Each thread is responsible for proper coloring of vertices in its partition
  1. Tentative coloring of vertices
- Synchronization of all threads
- Sequential coloring of conflicting vertices

- Downside - not much parallelism exploited

Algorithm 3 - Iterative Distributed Algorithm [GebremedhinEuroPar2005]

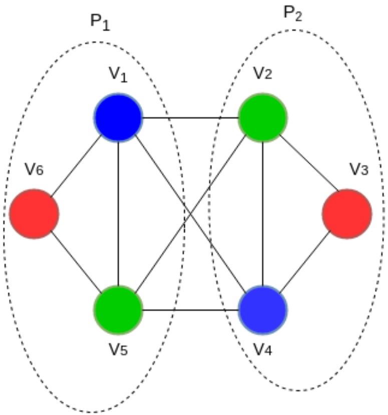


Algorithm 3 - Iterative Distributed Algorithm [GebremedhinEuroPar2005]



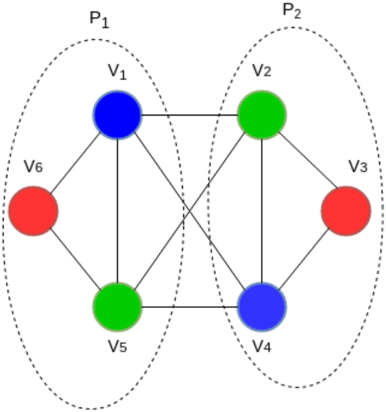
Algorithm 3 - Iterative Distributed Algorithm [GebremedhinEuroPar2005]

Phase 1:



Algorithm 3 - Iterative Distributed Algorithm [GebremedhinEuroPar2005]

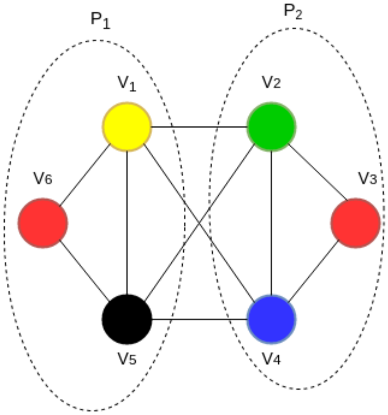
Phase 2:



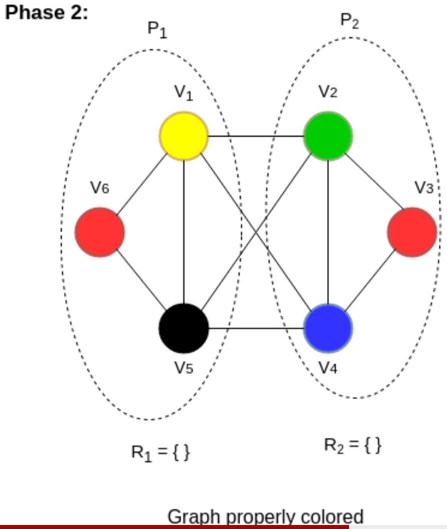
$R_1 = \{V_1, V_5\}$ $\quad$ $R_2 = \{\}$

Algorithm 3 - Iterative Distributed Algorithm [GebremedhinEuroPar2005]

Phase 1:



Algorithm 3 - Iterative Distributed Algorithm [GebremedhinEuroPar2005]



Algorithm 3 - Iterative Distributed Algorithm [GebremedhinEuroPar2005]

For each thread,

- Repeat until all the vertices in its partition are properly colored
  1. Tentative coloring of vertices - Each vertex has local copy of colors used by neighbours and it is assigned a color different from it. But still can result in improper coloring across partitions
  2. Synchronization of threads
  3. Identifying conflicts and marking nodes in lower partition for recolor
  4. Synchronization of threads

- Downside - number of conflicts increase with increase in number of partitions; random partitioning increases the number of cross edges

Algorithm 3 - Iterative Distributed Algorithm [GebremedhinEuroPar2005]

For each thread,

- Repeat until all the vertices in its partition are properly colored
  1. Tentative coloring of vertices - Each vertex has local copy of colors used by neighbours and it is assigned a color different from it. But still can result in improper coloring across partitions
  2. Synchronization of threads
  3. Identifying conflicts and marking nodes in lower partition for recolor
  4. Synchronization of threads

- Downside - number of conflicts increase with increase in number of partitions; random partitioning increases the number of cross edges

Algorithm 3 - Iterative Distributed Algorithm [GebremedhinEuroPar2005]

For each thread,

- Repeat until all the vertices in its partition are properly colored
  1. Tentative coloring of vertices - Each vertex has local copy of colors used by neighbours and it is assigned a color different from it. But still can result in improper coloring across partitions
  2. Synchronization of threads
  3. Identifying conflicts and marking nodes in lower partition for recolor
  4. Synchronization of threads
- Downside - number of conflicts increase with increase in number of partitions; random partitioning increases the number of cross edges

Algorithm 3 - Iterative Distributed Algorithm [GebremedhinEuroPar2005]

For each thread,

- Repeat until all the vertices in its partition are properly colored
  1. Tentative coloring of vertices - Each vertex has local copy of colors used by neighbours and it is assigned a color different from it. But still can result in improper coloring across partitions
  2. Synchronization of threads
  3. Identifying conflicts and marking nodes in lower partition for recolor
  4. Synchronization of threads

- Downside - number of conflicts increase with increase in number of partitions; random partitioning increases the number of cross edges

Algorithm 3 - Iterative Distributed Algorithm [GebremedhinEuroPar2005]

For each thread,

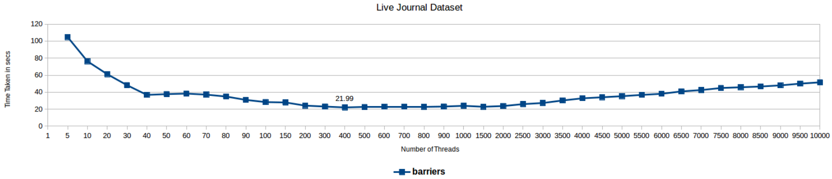
- Repeat until all the vertices in its partition are properly colored
  1. Tentative coloring of vertices - Each vertex has local copy of colors used by neighbours and it is assigned a color different from it. But still can result in improper coloring across partitions
  2. Synchronization of threads
  3. Identifying conflicts and marking nodes in lower partition for recolor
  4. Synchronization of threads

- Downside - number of conflicts increase with increase in number of partitions; random partitioning increases the number of cross edges

Algorithm 4 - Improved Iterative Parallel Algorithm [Gebremedhin2012]

- Parallel implementation of the previous algorithm using OpenMP on different architectures
- Downside - overhead of thread creation in each iteration

- In our algorithms, we aim to limit the number of threads acting on a partition of vertices.
- We simulate the iterative parallel algorithm using barrier for increasing number of threads as follows:



- It is noted that with increasing number of threads, performance declines

Basic Layout of the Proposed Algorithms

- Input: $G = (V, E)$, p threads
- Partition V into $\{V_1, V_2, \ldots, V_p\}$ uniformly at random
- Vertices in each partition are classified as:-
  1. Internal Vertices
  2. Boundary Vertices
- Each thread is responsible for proper coloring of vertices in its partition

Outline

Algorithm 1 - Using locks on vertices

Using 1 Coarse Grained Lock

For coloring any boundary vertex, a global lock must be acquired

Using Multiple Fine Grained Locks

1. Each boundary vertex has a corresponding lock
2. For coloring any boundary vertex, acquire respective locks on all adjacent boundary vertices in increasing order of ids (to avoid races)

Algorithm 1 - Using locks on vertices

Using 1 Coarse Grained Lock

For coloring any boundary vertex, a global lock must be acquired

Using Multiple Fine Grained Locks

1. Each boundary vertex has a corresponding lock
2. For coloring any boundary vertex, acquire respective locks on all adjacent boundary vertices in increasing order of ids (to avoid races)

Algorithm 1 - Drawback of using mutex locks

If two vertices v_1 & v_2 getting colored (from different partitions), are adjacent to a vertex v_3, then there is no need to lock v_3 if it is not getting colored.

Idea

A vertex to be colored acquires shared locks on neighbouring vertices; exclusive lock on the vertex itself in the order of increasing vertex ids.

Algorithm 1 - Drawback of using mutex locks

If two vertices v_1 & v_2 getting colored (from different partitions), are adjacent to a vertex v_3, then there is no need to lock v_3 if it is not getting colored.

Idea

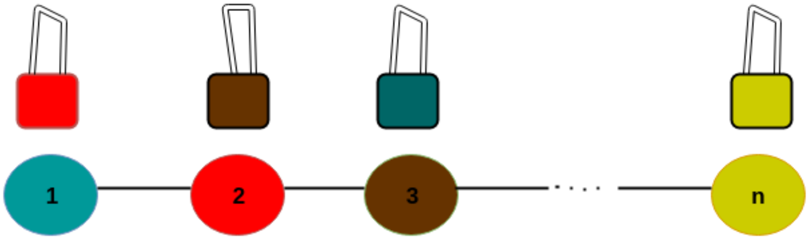
A vertex to be colored acquires shared locks on neighbouring vertices; exclusive lock on the vertex itself in the order of increasing vertex ids.

Algorithm 2 - Reduced number of locks

Each partition has a corresponding lock and a array element indexed by partition id
For coloring any boundary vertex v,

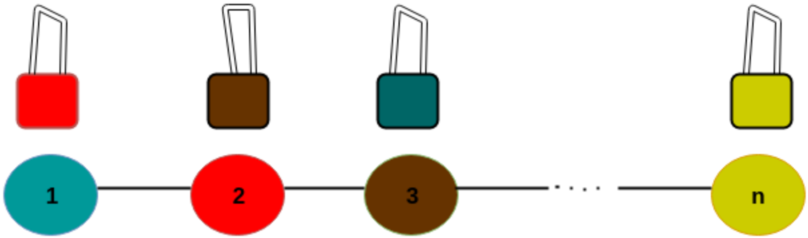
1. Acquire exclusive lock on the corresponding partition of v
2. Update v in the array index of the partition
3. Release exclusive lock on partition of v
4. for each partition in random order
    1. Acquire shared lock on partition
    2. Check if the vertex in that array index is neighbouring to v
    3. If not, release shared lock
5. end-for
6. Assign v, a least color different from all its neighbouring vertices
7. Release all acquired shared locks

Long transitive chains get created as below:



All threads are blocked! But alternating vertices could be colored still in parallel!
$\Rightarrow$ Need to cut waiting chains

Long transitive chains get created as below:

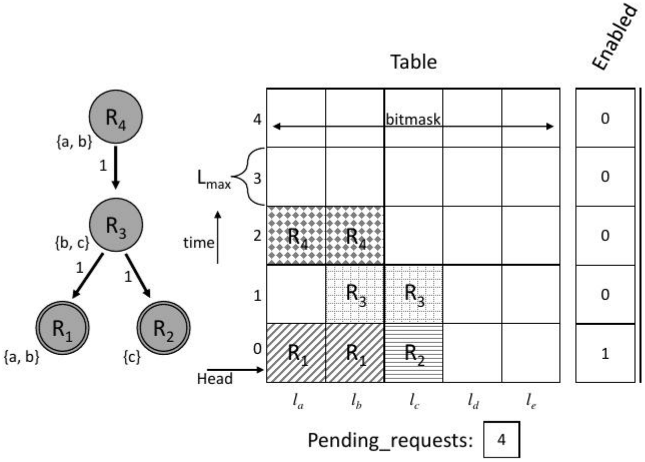


All threads are blocked! But alternating vertices could be colored still in parallel!
$\Rightarrow$ Need to cut waiting chains

Algorithm 3 - Scheduling requests [Anderson2015]

- Maintain a table data structure of boolean fields with number of rows = number of partitions + 1 and number of cols = number of vertices
- Each row corresponds to the requests being positioned there.
- Each column corresponds to a vertex
- When a vertex v is to be colored,
  1. It starts looking row by row where its request can be placed.
  2. Lock the head row
  3. l1: Check all the cols entry corresponding to v's neighbours in that row
  4. If any entry is not false, lock next row and repeat
  5. If the row where you can place your request is found, mark all corresponding entries as true
  6. Unlock the locked row
  7. Wait for your row to get enabled.

The last thread to fulfill its request in a particular row, sets the next row's enabled to allow all the requests in the next row to get satisfied.



When a row gets enabled, all the requests placed in it get fulfilled.

Why number of rows = number of partition works? (wrap-around)

Algorithm 4 - Using MIS

Instead of computing MIS in the original graph, we maintain a small subgraph and in each iteration, identify the Maximal Independent sets of vertices that can be colored in parallel.

Maximum number of vertices in graph = number of partitions

For coloring each boundary vertex,

1. Lock graph
2. add edges for all adjacent partitions
3. Unlock graph and wait for the vertex to become active

When a vertex gets colored and exits from the graph, it identifies the MIS of vertices that can be colored in parallel and make them active.

Outline

Experimental Setup

- 24 core Intel Xeon X5675, each core with 6 h/w threads
- Simulation using Pthreads
- Dataset: Real-World graphs from SNAP

Table: Results of Live Journal Dataset

	#threads	Time Taken in secs	#Colors used
Fine Grained Locks	70	6.18	334
BTO algorithm	200	8.26	335
Anderson improved	2	13.48	335
Sequential algorithm	1	13.86	334
Coarse grained locks	100	17.75	333
static graph	2	17.11	335
MIS	2	18.36	336
Barrier synchronization	400	21.99	334
Jones Plassman	40	64954	334

To be observed..

All these algorithms have been simulated by using First Fit Coloring scheme.

When using other coloring schemes like Largest Degree First, etc.
1. time taken for all algorithms increase proportionately due to sorting of vertices according to their degrees
2. number of colors used decreases.

Outline

Conclusion

In this presentation, we covered:

1. Existing parallel implementations of Graph Coloring
2. Presented new Graph Coloring Algorithm using locks
3. Described the evaluation results of the various algorithms on real world graphs

Future Work

1. Exploring ways of cutting waiting chain in fine grained locking
2. Think about pushing ahead of requests in Anderson's table
3. Converting undirected graph to DAG to exploit the parallelism of algorithms
4. Profile the code to see where performance lags
5. Extend these ideas for trees

References I

Erik G. Boman, Doruk Bozdag, Ümit V. Çatalyürek, Assefaw Hadish Gebremedhin, and Fredrik Manne.
A scalable parallel graph coloring algorithm for distributed memory computers.
In *Euro-Par 2005, Parallel Processing, 11th International Euro-Par Conference, Lisbon, Portugal, August 30 - September 2, 2005, Proceedings*, pages 241–251, 2005.

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.

JR Allwright, R Bordawekar, PD Coddington, K Dincer, and CL Martin.
A comparison of parallel graph coloring algorithms.
Technical report, Citeseer, 1995.

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.

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.

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.

References II

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

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.

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

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

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 Networks and Systems, RTNS 2015, Lille, France, November 4-6, 2015*, pages 3–12, 2015.

Thank You

Questions ?