Tournament Based Task Allocation in a Parallel MIS Algorithm

by Chidubem L. Ezeaka

[Previous/Other Information: i.e.: S.B., C.S. M.I.T., 2013]
Submitted to the Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science at the
Massachusetts Institute of Technology

May 2014 [JUNE 2014]

Author: Signature redacted

Department of Electrical Engineering and Computer Science.
May 28, 2014

Certified by: Signature redacted

Prof. Nir Shavit, Thesis Supervisor.
May 28, 2014

Accepted by: Signature redacted

Prof. Albert R. Meyer,
Chairman, Masters of Engineering Thesis Committee.

Tournament Based Task Allocation in a Parallel MIS Algorithm
by Chidubem L. Ezeaka
Submitted to the Department of Electrical Engineering and Computer Science

May 28, 2014
In Partial Fulfillment of the Requirements for the Degree of Master of Engineering in Electrical Engineering and Computer Science

Abstract

This paper explores the use of a tournament data structure for task allocation and dependency tracking in a parallel Maximal Independent Set (MIS) algorithm as a way to reduce contention in counter updates and improve runtime. The tournament data structure adds a noticable overhead to the algorithm that causes T_1 time of the algorithm to increase but then there is a steady improvement in performance as we increase the number of worker threads. Running the tournament algorithm with 12 threads in our experimental setup, we are able to get an average speedup of 1.13 for our test suite of 7 real-world graphs.

1 Introduction

One common issue in parallelizing computations is avoiding situations between processors that threaten the atomicity of actions. . Runtime can significantly be improved if we ensure that all the active processing cores are working cohesively and not competing adversely for resources, harmfully blocking each other or repeating work. This can be achieved by avoiding memory contention, increasing parallelism and having efficient methods for task allocation. Contention can be hard to avoid when implementing graph algorithms on modern multicore machines expecially when dealing with computations on large graphs where the result in one node is dependent on a precomputed value from a preceding vertex. We will study a parallel graph algorithm for computing a maximal independent set to analyze this problem and try to propose a generalized solution.

This paper demonstrates that using a tournament data structure for task allocation could help reduce contention in performing updates for the parallel MIS algorithm.

- Section 2 introduces the maximal independent set problem and gives a simple sequential algorithm as an example.
- Section 3 explains a parallel MIS algorithm that uses counters to track predecessor dependencies so that we can understand the issues that cause contention and how runtime can be improved.
- Section 4 thoroughly describes the tournament structure to be used in the parallel MIS algorithm. It then gives some insight as to why it works and how it can be expected to improve the counter based parallel MIS algorithm.
- Section 5 evaluates the effectiveness of the tournament based parallel MIS algorithm by comparing its performance to the performance of the regular counter based parallel MIS algorithm on 7 real-world graphs.

- Section 6 offers some concluding comments

2 The Maximal Independent Set Problem

Given an undirected graph, $G = (V, E)$, an independent set is a set of vertices, $I \subseteq V$, such that no two vertices in I are adjacent in the graph (alternatively, that there are no vertices in the set, I, that are neighbors in G). This definition can equivalently be rephrased to say that each edge in the graph, $e \in E$, has at most one endpoint in I.

A maximal independent set is defined as an independent set such that it is not possible to add any other vertex to the set while having it remain as a valid independent set. A maximal independent set is therefore an independent set that is not a subset of any other independent set. Based on the definitions above, we can identify an MIS as any set of vertices, I, for a graph that satisfies the following two conditions:

- For any given vertex, v, in the independent set, I, none of its neighbors are also in the set (independent set).
- Every vertex, v, not in the MIS is has at least one neighbor that is (maximal).

The maximal independent set problem has applications in scheduling, information theory and molecular biology very similar to the maximum independent set problem and is much easier to solve for. It is preferable in cases where a set that is an approximation of the maximum is all that is required.

2.1 Sequential MIS Algorithm

The sequential algorithm for computing the MIS of a graph is a simple greedy algorithm that takes in inputs G, the graph, and π, an arbitrary list of priorities for each vertex. The priority list, π is decides what order we visit the vertices

in. The algorithm, shown in Figure 1 below, iterates over the vertices in order of decreasing priority, marking a vertex as being in the resulting MIS, I, if and only if no neighboring vertex has previous been marked as being in the MIS. To simplify the task of checking that no previous neighbors has already been added to the MIS, we will modify algorithm such that whenever a vertex is marked as being in the MIS, we loop over all its neighbors and mark them as not in the MIS. So for each vertex in the input graph, G, we first check if it has already been marked. If so, we continue to the next vertex else we mark it as being in the MIS and loop over its neighbors and mark them as not part of the MIS.

```
1  int* serial_mis(sparseRowMajor sparse_rep, int * priorityList) {
2      // Initialize output array
3      int* mis_array = (int*) calloc(sparse_rep.numRows, sizeof(int));
4      for (int i = 0; i < sparse_rep.numRows; i++) {
5        mis_array[i] = -1;
6      }
7
8      // Arrange verticeIDs into orderedVertices based on priorityList
9      orderedVertices = (unsigned int*) malloc(sizeof(unsigned int) * sparse_rep.numRows);
10     ...
11
12     // Loop over orderedVertices in order
13     for (int k = 0; k < sparse_rep.numRows; k++) {
14       int vid = orderedVertices[k];
15       if (mis_array[vid] == 0) {
16         continue;
17       }
18
19       mis_array[vid] = 1;
20       int* neighbors = &sparse_rep.ColIds[sparse_rep.Starts[vid]];
21       unsigned int degree = (sparse_rep.Starts[vid + 1] - sparse_rep.Starts[vid]);
22       for (int i = 0; i < degree; i++) {
23         mis_array[neighbors[i]] = 0;
24       }
25     }
26
27     return mis_array;
28 }
```

Figure 1: Greedy Serial MIS Algorithm

3 Counter Based Parallel Algorithm for MIS

The serial algorithm given in the previous section is not trivially parallelize by allowing for multiple vertices to be processed concurrently because conflicts that are likely to arise in marking the vertices can cause the algorithm to return an incorrect set. Instead, we should only allow a vertex to be added to the MIS when it has no higher-priority neighbor left unmarked. This parallel algorithm returns the same MIS on an input graph as the sequential algorithm for a given priority list. A vertex, v, will only be marked after all of its higher priority neighbors have been marked. If its earlier neighbors are marked the same as in the sequential algorithm, then it is too. So it follows by induction on the vertices in order of priorities that the result of the two algorithms are the same. The only difference is that the parallel algorithm is likely to accept some vertices into the MIS at an earlier time than the sequential algorithm, but the MIS produced for an input graph is deterministically the same for a given priority list.

To run this parallel algorithm, we perform some precomputation on the vertices in our graph that allows us to more efficiently tell when all the higher priority neighbors for a given vertex have been marked. We partition the neighbors of every vertex, $v \in V$, into predecessors and successors based on the given priorities. A neighbor, u, is a predecessor if it is higher priority than vertex, v and a successor if it has a lower priority according to the given priority list. If u and v have the same priority then we choose to use their vertexIDs as a tie breaker to determine if u is a predecessor or a successor. Therevore we can form an ordered pair $(\pi[u], u)$ which is compared lexicographically to establish an ordering among all vertices.

$$Predecessor : \{u \in V : (u, v) \in E and (\pi[u], u) >> (\pi[v], v)\}$$

$$Successor : \{u \in V : (u, v) \in E and (\pi[u], u) << (\pi[v], v)\}$$

Every neighbor of a given vertex is either one of its predecessors or one of its successors. Also since the comparison reverses when we switch u and v, we easily notice that for each edge in the graph, one vertex becomes the predecessor and the other becomes the successor. With this method above, we are successfully able to turn any undirected graph G into a directed acyclic graph (DAG) with the same vertices and edges as the input graph, G, but with the edges now directed from higher priority vertices to lower priority ones according to π. We call this DAG the priority DAG. This gives a dependence structure among the results at each vertex. The result at a vertex in our MIS computation will only depend on the result of its predecessors in the priority DAG. So we have enough information to decide how to mark a given vertex when we have marked all of its predecessors. Following the MIS definition, a vertex is marked as being in the MIS if and only if none of its predecessors are marked as being in the MIS. The algorithm begins by marking the vertices that do not have any predecessors as being in the MIS and then it recursively marks all successor vertices based on the value of their predecessors until we have traversed the whole priority DAG and all the vertices in the input graph are marked.

For each vertex, we will store a counter that let's up keep track of how many of its predecessors have not been marked. We also include a mutex and a variable for us to store the current belief of the marking. A thread that does the work of marking a vertex is also responsible for performing an update, decrementing the internal counter, for each of its successors. It grabs the mutex and decrements the counter but before releasing the mutex, the thread checks if the counter is now zero, meaning that it was the last required update and all the predecessors of that vertex have now been marked. We are now sure that there is all the required in formation in the local inMIS variable for the updating thread to go ahead and decide how to mark the current vertex.

On further inspection, we will notice that is however not always necessary to wait until all predecessors of a vertex have been marked before we can decide

```
1  struct CounterClass
2  {
3    volatile unsigned long inMIS;
4    volatile unsigned int counter;
5    volatile unsigned short mutex;
6
7    signed int update_counter(unsigned int _inMIS);
8  };
```

Figure 2: Counter

how to mark the vertex. From the MIS definition, knowing that any one of a vertex's predecessors is marked as being in the MIS tells us that it is not allowed to also be in the MIS. So after any vertex has been decided to be in the MIS, it is correct to mark all of its successors as not being in the MIS without having to wait for all the other predecessor's updates. Doing this saves all other threads subsequent threads the work required to update the counter. It is important to note though that the only way for us to decide that a vertex should be marked as being in the MIS is if none of its predecessors is also in the MIS. So to decide this, we would need to wait for all the predecessors to update the counter which could still lead to a lot of memory contention that is detrimental to performance.

4 Tournament Based Parallel Algorithm for MIS

To reduce the adverse effects of some of this kind of contention on the runtime of the parallel algorithm, we are looking at the effects of replacing the individual predecessor vertex counter at each vertex with a tournament tree. In particular, this implementation serves to distribute the counter updates and combine the various results in a manner specific to the MIS algorithm to efficiently determine which thread gets assigned the task of marking the given node.

The tournament for each vertex is a balanced binary tree with each counter value stored at a leaf node. Each predecessor is assigned to a leaf node it will update and start from to compete in the tournamet. The proposed tournament proceeds in rounds, which happen at all nodes of a tournament graph. To update

```
1  inline signed int Counter::update_counter(unsigned int _inMIS)
2  {
3      while ( true ) {
4          if (__sync_lock_test_and_set(&mutex, 1) == 0 ) {
5              if (counter == 0) {
6                  mutex = 0;
7                  return −1;
8              }
9
10             finalInMIS |= _inMIS;
11             if (finalInMIS == 1) {
12                 counter = 0;
13                 mutex = 0;
14                 return 0;
15             }
16
17             counter−−;
18             if ( counter == 0 ) {
19                 mutex = 0;
20                 return 1;
21             } else {
22                 mutex = 0;
23                 return −1;
24             }
25         } else {
26             while( mutex != 0 );
27         }
28     }
29 }
```

Figure 3: Early Exit Update Counter Method

the state of the successor, a thread starts at the leaf assigned to the predecessor it is assigning the value for an works its way up to the root. Two workers compete in each round with at most one winner following an edge up the tournament, closer to the root. The winner at a particular node is free to move up and goes on to compete in the next level. The winner at the tournament root node is then assigned the task of marking the MIS status of the vertex. Only the winning worker needs to know that the task can now be completed as opposed to requiring that all participating workers wait while they decide that the task can now be assigned. Any losing thread leaves the tournament to do any other work. The winning thread propagates the needed information up the tree, while the losing thread returns a value of -1 indicating that it lost and is free to perform other tasks.

At each node in the tournament, the competing threads can be any one of two different kinds of players based on what information it has about a particular predecessor vertice:

1. Lazy Player: This corresponds to an update for a predecessor that was marked as not being in the MIS, meaning that a given neighbor, v, could possibly be in the MIS. It is not able to decide on its own for sure and still has to hear from other predecessors of v to ascertain before the task of marking the vertex can be completed. This kind of player requires other players to have arrived at the tournament node. It wins at a node if it is the last lazy player to arrive.

2. Eager Player: This corresponds to an update for a predecessorwas marked as beoing in the MIS. By definition this kind of update means that the competition vertex could not possibly be in the MIS. This update is certain and as such, does not need to check with any other update. This kind of player does not care if any other players have arrived at the tournament node. It wins at a node if it is the first eager player to arrive.

It is easy to see that there is always a winner and a loser for any pair players based on the winning criteria enumerated for each kind of player described above. This guarantees us that there is always at least one thread that gets assigned to perform the task at the root node. For our implementation, we have the losing thread move on to perform other tasks. This second constraint ensures that there is at most one thread that gets assigned the desired task, letting us avoid any repeated work in our computations.

The state value at each node is initialised to a zero value, meaning that no participants have visited that node yet. To participate in a round at a node in the tournament graph, a player uses a TestAndSet operation to atomically retrieve the previous state and write its type value to the state of the node to inform any other subsequent participants what kind of player had visited the

```
1  signed int Vertex::compete_in_tournament (
2      unsigned int hash, unsigned int predInMIS,
3      unsigned char *tournamentArray, unsigned int scsrID)
4  {
5      unsigned int numN = sparse_rep.Starts[scsrID + 1] - sparse_rep.Starts[scsrID];
6
7      unsigned int size = 1 << (unsigned int) logSize;
8      unsigned char *tournament = &tournamentArray[(edgeIndex+7)&(~0x07)];
9      unsigned long *nodes = (unsigned long *) tournament;
10     unsigned int index = (hash & (size - 1));
11     LeafClass *leaf = (LeafClass *) &(nodes[size]);
12     signed long meInMIS = leaf[index].update_counter(predInMIS);
13     if( meInMIS != -1 ) {
14         // compete in tournament
15         unsigned long type;
16         if (meInMIS == 1) {
17             type = 1; // lazy
18         } else {
19             type = 2; // eager
20         }
21         index += size;
22
23         while( index > 1 ) {
24             index = (index >> 1);
25             unsigned long result = __sync_lock_test_and_set(&nodes[index], type);
26             if( result == 1 ) {
27                 // Move up in the tournament
28             } else if ( result == 2 ) {
29                 return -1;
30             } else if ( type == 1 ) {
31                 // Only eager player moves up in this case
32                 return -1;
33             }
34         }
35         return meInMIS;
36     } else {
37         return -1;
38     }
39 }
```

Figure 4: compete_in_tournament method from the Vertex class

node before (Figure 4). We choose to have the lazy players write 1 and the eager players write 2 as their type values.

As shown in Figure 5, after a vertice is marker we iteratively call the update function on all of its successors. For example, suppose a thread is performing an update from a predecessor of the competition vertex. If that predecessor was marked as not being in the MIS then the thread will act a lazy player for this

update because the predecessor on its own does not have enough information to decide how to mark the vertice. Alternatively, if the predecessor was marked as part of the MIS then the updating thread will act as an eager player. The update starts from the predecessor's assigned leaf node and works its way towards the root till it either loses and leaves the update to perform another task or it wins at the root node and gets assigned the task of marking the vertice. To compete in the tournament at a given node, the participant performs a TestAndSet operation, allowing it to atomically retrieve the previous state and write its type to the state of the node. The TestAndSet operation suffices for this since we are traversing a tournament tree. Therefore, there will only ever be at most two visitors to a node. The TestAndSet operation has a consensus number of two so any updates passing through a node will be able to tell if it is first or second based on the value it receives from the atomic operation. Since we initialize all the nodes to 0 before the tournament begins, we can say for sure that any thread that gets a 0 back from the TestAndSet was first at that node. Since the first thread always writes a non-zero value (1 or 2 corresponding to the type of player), if an update gets back a non-zero value from the atomic operation, then we are sure that it was the second update to visit that node. The possible result values from the TestAndSet for any updates are 0, 1 or 2.

Lazy Player (Type 1)

- **0** - Loses and returns -1 because it does not have enough information to proceed
- **1** - Wins this round and moves up to the next round in the tournament. It is effectively propagating both Lazy updates to the next round of the tournament.
- **2** - Loses and returns -1 because it defers to an eager player

Eager Player (Type 2)

0 - Wins this round and moves up to the next round in the tournament. Any other incoming player defers to it.

1 - Wins this round and moves up to the next round in the tournament.

2 - Loses and returns -1 because it defers to the first eager player

The winning thread at the root will have enough information to decide if the competition vertex should be included in the MIS so it now assigned the task of marking it appropriately. It is easy to prove that the behavior of both the lazy and eager players will always lead to a correct decision as to whether the given vertex should be included in the MIS.

```
1  void Vertex::mark_vertex (
2      Vertex *vertices, unsigned char *trntArray,
3      int * mis_array, unsigned int *neighbors, signed int inMIS)
4  {
5      mis_array[vertexID] = inMIS;
6      unsigned int *successors = &(neighbors[edgeIndex]);
7
8      cilk_for( int i = 0; i < numSuccessors; i++ ) {
9          signed int inMIS = vertices[successors[i]].compete_in_tournament(hashValue,
10                                                          inMIS, trntArray, successors[i]);
11         if( inMIS != -1 ) {
12             vertices[successors[i]].mark_vertex(vertices, trntArray,
13                                                 mis_array, neighbors, inMIS);
14         }
15     }
16 }
```

Figure 5: The mark_vertex method shows how the MIS marking task proceeds from predecessors to successors. As with the counter algorithm, we start by marking the vertices that do not have predecessors and recursively mark all successor vertices until all the vertices in the input graph are marked.

In the case where a vertex from the input graph has very many predecessors, it might not make sense for each predecessor to have its own leaf node because this can be very space inefficient and cause adverse performance with a lot of cache misses while moving up the tournament tree. To try mitigate this issue, we decide to allow each leaf node to be a counter similar to the one we used without the tournament tree. We can then assign a certain number of predecessors to each counter. The results from the counter will then be used by the

decideing predecessor to compete in the tournament as described above. This subtle modification allows us to have the same task allocation structure in a more space conscious and ultimately more efficient manner.

5 Performance

This section evaluates the effectiveness of the tournament based parallel MIS algorithm by comparing its performance to the performance of the regular counter based parallel MIS algorithm on 7 real-world graphs. We first describe the experimental setup used for the benchmarking and then proceed to compare the execution times of both algorithms. It is again important to note that both algorithms produce the same MIS for a given priority list so this performance analysis will only focus on the runtimes.

5.1 Experimental Setup

We implemented both the regular counter and the Tournament Tree parallel MIS algorithms in C++ using Intel CilkPlus [1]. We used a random priority list as input so we ran each experiment several times and reported the average value after removing the highest and lowest runtimes.

These implementations were tested on a suite of 7 real-world graphs (shown in Figure 6) that came from the Large Network Dataset Collection provided by Stanfords SNAP project [2]. We ran all our experiements on a dual-socket Intel Xeon X5650 with a total of 12 processor cores operating at 2.67- GHz capable of running only one thread per core, NUMA architecture with 2 NUMA nodes, 49 GB of DRAM, two 12-MB L3-caches each shared between 6 cores, and private L2- and L1-caches of 128 KB and 32 KB, respectively.

Graph	$\|E\|$	$\|V\|$	$\|E\|/\|V\|$	Δ
com-orkut	234.37M	3.07M	76.28	33313
soc-LiveJournal	85.70M	4.85M	17.68	20333
cit-Patents	33.04M	6.01M	5.50	793
as-skitter	22.19M	11.10M	2.00	35455
wiki-Talk	9.32M	2.39M	3.89	100029
web-Google	8.64M	0.92M	9.43	6332
com-youtube	5.98M	1.16M	5.16	28754

Figure 6: Metrics for each of the 7 real-world graphs used as benchmarks to evaluate the performance of the tournament parallel MIS algorithm

5.2 Parallel Overhead

We can see that there is more overhead involved in the tournament algorithm than in the algorithm that uses only counters. This can be attributed to the extra work involved in initializing the tournament nodes and the costs associated with the updating threads competing in the tournament structure. In Figure 7, we see that the overhead is consistent across all the benchmark graphs.

Graph	Serial	Counter	Tournament
com-orkut	1.268	5.194	6.094
soc-LiveJournal	2.201	2.381	2.657
cit-Patents	2.345	1.732	1.997
as-skitter	4.211	0.781	0.900
wiki-Talk	1.104	0.224	0.252
web-Google	0.292	0.311	0.346
com-youtube	0.390	0.205	0.225

Figure 7: T_1 for each benchmark graph taken from Stanfords SNAP project [2]. The values are time in seconds for each MIS algorithm discussed. Each measurement is the median of several independent trials.

5.3 Speedup

Though the tournament structure introduces some significant overhead to the parallel MIS algorithm that causes T_1 to increase, we see that the difference in performance reduces to the point where we actually see the Tournament algo-

rithm performing better with 12 worker threads. In Figure 8, we see that we get an average speedup of 1.13 across all the benchmark graphs from using the tournament structure.

	Speedup					
Graph	S_1	S_2	S_4	S_8	S_{10}	S_{12}
com-orkut	0.852	0.890	0.894	0.946	1.002	1.100
soc-LiveJournal	0.896	0.911	0.925	0.938	1.067	1.122
cit-Patents	0.867	0.875	0.880	0.900	1.033	1.050
as-skitter	0.868	0.920	0.854	0.919	0.977	1.159
wiki-Talk	0.886	0.901	0.894	0.917	0.847	1.001
web-Google	0.898	0.922	0.889	0.929	1.014	1.305
com-youtube	0.910	0.935	0.966	0.952	1.055	1.110

Figure 8: Performance measurements for a set of real-world graphs taken from Stanfords SNAP project [2]. The values shown are the speedups with 1, 2, 4, 8, 10 and 12 workers.

6 Conclusions

This paper discusses using the tournament based data structure only in the context of reducing contention in counter updates for a parallel Maximal Independent Set (MIS) algorithm but it is possible to generalize this structure into a more generic interface that can be used for task allocation in any other kinds of data graph computations. There is a significant increase in overhead in using some of these parallel algorithms but in an environment with a lot of contention and enough workers, the benefits to speedup would far outweigh these costs.

References

[1] Intel. Intel cilk plus. Available from *http://software.intel.com*, 2013.

[2] J. Leskovec. Snap: Stanford network analysis platform. Available from *http://snap.stanford.edu/data/index.html*, 2013.