CS406 Parallel Computing

Homework 2 - High Performance Distance-1 Graph Coloring

ORAN CAN ÖREN

11 November 2017

1 Task Description

Graph coloring is a widely applicable graph problem; it has use cases in task scheduling, register allocation, social network analysis and many more areas. Furthermore, the second DIMACS implementation challenge is on graph coloring, and hence it is a notable problem.

The focus of our task is to implement algorithms to solve the distance-1 graph coloring problem, i.e. a coloring of the vertex set such that any pair of adjacent vertices receive distinct colors. Graph coloring is a hard problem to compute solutions for, hence our aim is to implement greedy heurisitcs that utilize parallelism.

2 Implementation Details and Results

I have implemented the ideas introduced by [1] to obtain parallelized, efficient and accurate algorithms solving the distance-1 graph coloring problem. The software runs on CPU; parallelism was achieved with OpenMP on a shared-memory environment.

The following subsections discuss implementation details that gradually increase the performance of the greedy algorithm [1]. The algorithm is outlined below.

Algorithm 1: Parallel Graph Coloring for Manycore Architectures

```
Result: color $c_i$ for $\forall v_i \in V$
1 while $\exists v_i \in V$ not colored do
2     assignColors();
3     detectConflicts();
4     forbidColors();
5 end
```

2.1 Algorithm v0.01 - whole traversal & linked lists

2.1.1 The algorithm

The first algorithm I designed keeps a linked-list of forbidden colors for each vertex, that is initially empty. Then, for assign-colors step the algorithm traverses over each vertex v_i of the graph and assigned the minimum color that v_i can be assigned according to its forbidden color list, $f[i]$. The detect conflict phase traverses over each vertex $v_i \in V$ and checks all neighbors of v_i, n_i^j to see if there is a matching color; if this is the case, then the vertex having the smaller label has its color reset to unknown. For the forbid color phase, the algorithm traverses all vertices and their neighbors once againf to find v_i and n_i such that exactly one of v_i and n_i are colored; then the uncolored one forbids the color of the colored one.

2.1.2 Performance analysis

This algorithm is naive, it traverses *all vertices* of the graph and their neighbors two times per iteration. Also, keeping the forbidden colors for each vertex as linked list is costly because linked lists cannot utilize spatial locality in contrast to arrays. Another drawback of this is that it pushes forbidden colors without taking into account if the color is already in the forbidden colors list; however, performance profiling have shown that this didn't have a major impact on the performance.

One of the most costly operations of this algorithm is the atomic operations. They were handled by critical regions in OpenMP and such frequent mutual exclusive code block caused frequent idling of threads.

Algorithm 2: Parallel algorithm v0.01

```
∀i∈ $\{1, 2, ..., |V|\}$ initialize $f[i]$ as empty linked-list
∀i∈ $\{1, 2, ..., |V|\}$ initialize $c[i] = 0$
numColored = 0
while $numColored < |V|$ do
    for $v_i \in V$ in parallel do
        if $c[i] = 0$ then
            $c[i]$ =next available number in $f[n_i]$
            atomic numColored += 1
        end
    end
    for $v_i \in V$ in parallel do
        for $n_i \in Neighbors[v_i]$ do
            if $c[v_i]$=$c[n_i]$ then
                c[$min(v_i, n_i)$]=0
                atomic numColored -= 1
            end
        end
    end
    for $v_i \in V$ in parallel do
        for $n_i \in Neighbors[v_i]$ do
            if $c[v_i] \neq 0$ and $c[n_i] = 0$ then
                push $c[v_i]$ to $f[n_i]$
            end
            else if $c[v_i] = 0$ and $c[n_i] \neq 0$ then
                push $c[n_i]$ to $f[v_i]$
            end
        end
    end
end
```

2.1.3 Execution times

For a relatively small graph (coPapersDBLP) the algorithm took about 270 seconds on a single thread and about 240 seconds on four threads to terminate. Both threads have yielded 377 colors.

2.2 Algorithm v0.02 - decreased mutual exclusion & using trees

2.2.1 The algorithm

The purpose of the critical regions in version 0.01 was to count the number of colored vertices. This was needed because the main loop termination condition is dependent on it. Instead, another mechanism is to traverse all vertices and check if there is at least one vertex without a color assigned to it.

Moreover, in this version forbidden colors were stored in red black trees. The reason for this was to reduce the cost of search operation on forbidden colors. Subsequently, the forbid colors phase now searches the tree prior to insertion of a color and the color is inserted only if the color is non-existent in the tree.

Algorithm 3: Main loop with better control condition

```
Result: color $c_i \forall v_i \in V$
coloringFinished = false
while not coloringFinished do
    assignColors();
    detectConflicts();
    forbidColors();
    coloringFinished=isAllColored();
end
```

2.2.2 Performance analysis

The removal of critical regions has improved the overall running time significantly; however, using trees to store forbidden colors actually increased the overall time spent in forbid colors phase. This is anticipated since the previous version just pushed an item to the linked list without performing search. In contrast, the elapsed time to search for a color to assign to a vertex has shortened.

2.2.3 Execution times

The coPapersDBLP graph took about 100 seconds on a single thread and about 85 seconds on four threads; both yielding 377 colors.

2.3 Algorithm v0.03 - a clean approach

2.3.1 The algorithm

Instead of traversing the whole vertex array at each iteration in assign colors, detect colors and forbid colors phases a better solution is to keep the uncolored vertices in a data structure and traverse this data structure at where necessary. In each iteration, at least one vertex is permanently colored[1]; therefore the number of vertices that are going to be processed will decrease at each iteration.

My choice of data structure to keep the uncolored vertices is the array structure. There are two reasons for this; first, arrays are random access structures and therefore parallelising the traversal of arrays comes with almost no additional overheads. Secondly, arrays utilize spatial locality unlike pointer-dereference based dynamic structures such as trees and lists.

2.3.2 Execution times

Parallel speedup can be observed from the table below. All runs have yielded 377 colors.

Table 1: Execution times of algorithm v0.03 on coPapersDBLP graph

	1 thread	**2 threads**	**4 threads**	**8 threads**	**16 threads**	**32 threads**
coPapersDBLP	42 seconds	30 seconds	20 seconds	10 seconds	6.8 seconds	5.3 seconds

2.4 Algorithm v0.04 - forbidden colors by traversal [coloring.cpp]

2.4.1 The algorithm

Using linked lists and trees have caused additional complexities to paralellism by need of critical regions. In this version, the algorithm doesn't store the forbidden colors at all. Instead a naive yet thread-friendly approach was used. To receive an appropriate color for a given vertex, the algorithm simply traverses over each neighbor to see if a given color is available. Using this method, for each number starting from 1, the algorithm finds the minimum color available.

2.4.2 Performance analysis

By removing the storage for keeping forbidden colors and reading neighbors, the algorithm makes use of spatial and temporal locality by using the CSR (compressed sparse row) data only. With linked lists and trees, the algorithm could not utilize such cache features.

Algorithm 5: Simple GetNextColor

```
Input: $v_i$: vertex to get color for
$N = \{n_1, n_2, ..., n_3\}$: neighbors of $v_i$
$c$: array keeping colors
Output: $c_i$: appropriate color for $v_i$
1  for $i \in \{1, 2, ...\}$ do
2      colorValid = true
3      for $n_i \in N$ if colorValid do
4          if $c[n_i] = i$ then
5              colorValid = false
6          end
7      end
8      if colorValid then
9          return $i$
10     end
11 end
```

2.4.3 Execution times

Table 2: Execution times and number of colors used for algorithm v0.04

	1 thread	**2 threads**	**4 threads**	**8 threads**	**16 threads**	**32 threads**
af_shell3	98 ms [25]	70 ms [30]	45 ms [30]	36 ms [30]	35 ms [30]	66 ms [30]
bone010	350 ms [39]	295 ms [48]	164 ms [48]	102 ms [45]	140 ms [48]	131 ms [51]
coPapersDBLP	542 ms [337]	513 ms [337]	447 ms [337]	241 ms [337]	156 ms [337]	222 ms [337]
nlpkkt120	159 ms [2]	158 ms [2]	107 ms [4]	95 ms [5]	115 ms [5]	167 ms [5]
nlpkkt240	1418 ms [2]	1036 ms [2]	873 ms [3]	780 ms [3]	790 ms [5]	785 ms [4]

3 Performance Comparison Charts

Execution time plot below show the time elapsed for the coloring procedure of algorithm v0.04.

Figure 1: running times on four graphs vs threads

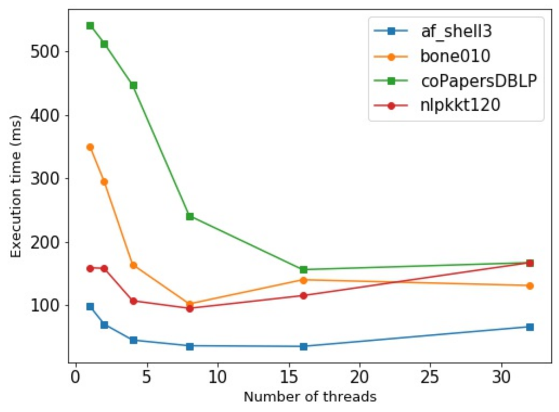
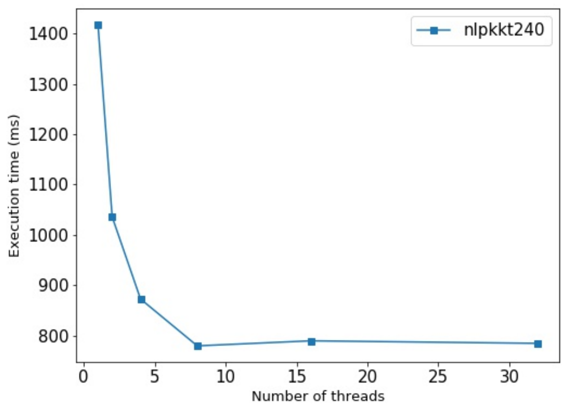


Figure 2: running times on nlpkkt240 graph vs threads



References

[1] Deveci, Mehmet Boman, Erik D. Devine, Karen Rajamanickam, Siva. (2016). Parallel Graph Coloring for Manycore Architectures. 892-901. 10.1109/IPDPS.2016.54.