banner above paper title

Evaluating Graph Coloring on GPUs

Name1
Affiliation1
Email1

Name2 Name3
Affiliation2/3
Email2/3

Abstract

This paper evaluates features of graph coloring algorithms implemented on graphics processing units (GPUs), comparing coloring heuristics, partitioning strategies, and thread decompositions. As compared to prior work on parallel graph coloring for other parallel architectures, we find that the large number of cores and relatively high global memory bandwidth of a GPU lead to different strategies for the parallel implementation. Specifically, we find that a simple block partitioning of nodes often outperforms, in terms of execution time or number of colors, the partition found by METIS, which tries to minimize cross-processor edges. We also discover that our coloring heuristics lead to the same or fewer colors than a prior approach for a distributed-memory cluster architecture. Our algorithm resolves many coloring conflicts across partitioned blocks on the GPU by iterating through the coloring process up to three times, before returning to the CPU to resolve remaining conflicts, and this iteration dramatically impacts speedups as compared to a single pass. Overall, parallel graph coloring speeds up well for the sparse graphs we studied, commonly yielding speedups above 1000X over a baseline CPU implementation.

Categories and Subject Descriptors CR-number [*subcategory*]: third-level

General Terms term1, term2, term3

Keywords Graph coloring, Parallel algorithm, CUDA

1. Introduction

Graph coloring refers to the assignment of labels or colors to elements of a graph (vertices or edges) subject to certain constraints. In this paper, we consider the specific problem of assigning colors to vertices so that no two neighboring vertices (vertices connected by an edge) have the same color. There are several known applications of graph coloring like assigning frequencies to wireless access points, time-tabling and scheduling, register allocation and printed circuit testing, iterative solution of sparse linear systems [], preconditions [], sparse tiling [], and eigenvalue computation [].

Graph coloring is NP-hard, and even computing a $n^{1-\epsilon}$-approximation of the chromatic number of a graph is NP-hard [**?**]. Therefore, a number of heuristics have been developed to assign colors to vertices; some commonly used heuristics include First Fit[], Largest Degree Order[] and Saturation Degree Order[]. These heuristics tend to trade off minimizing the number of colors and minimizing execution time. Combinations of these algorithms have also been used to create better heuristics. These algorithms are generally based on the same general greedy framework: a. vertex is selected according to some predefined criteria and colored with the smallest valid color. The selection and coloring continues until all the vertices in the graph are colored.

For coloring large graphs, a parallel implementation seems natural as we can split the graph into small subgraphs, color each of them separately in parallel, and then combine results for all of the subgraphs to derive the final coloring. Occasionally, this combining step will encounter conflicts, where two adjacent vertices that span multiple subgraphs have been assigned the same color. A parallel algorithm must resolve conflicts, by assigning one of the vertices involved in the conflict a different color. Processing conflicts partially serializes the computation and may also lead to the use of a higher number of colors.

This paper examines a graphics processing unit (GPU) mapping of parallel graph coloring. Prior parallel graph coloring algorithms have been evaluated on conventional shared-memory multiprocessors [reference] or distributed systems [reference], but to the best of our knowledge, this is the first study of how GPU architectures affect performance gains and number of assigned colors in a parallel implementation. Our study demonstrates that features of the GPU architecture significantly impact the algorithms selected. Specifically, the support for efficient fine-grain multithreading facilitates strong performance gains over CPU implementations (as high as 1000X) because hundreds or even thousands of threads can be applied to the parallel coloring. Further, multi-threading can hide the latency to memory of the irregular accesses that on a conventional CPU do not effectively utilize caches. In addition, most parallel graph algorithms seek to partition graphs carefully to minimize the number of edges that cross processor boundaries, since cross-processor accesses typically have very high communication latencies [need a reference, possibly just a METIS reference]. On a GPU, where the entire graph is stored in a global device memory, the latency of accessing vertices inside or outside the current subgraph is no different, making the partitioning process less important.

Our study also identifies several features of graphs and the GPU implementation that impact both performance and number of assigned colors. The size and sparsity of a graph affects the number of conflicts, which as suggested above, affects both performance and the quality of the coloring. Our study evaluates parallelizing polynomial graph coloring algorithms that are known to derive fewer colors than lower complexity algorithms, but for small graphs these more efficient algorithms may also lead to the same quality solution at lower cost. Therefore, certain graph properties determine whether using a GPU is profitable. We will evaluate alternative coloring heuristics, partitioning strategies and granularity of the partitioning, to find the solution that leads to a high-quality coloring and a significant performance gain over a baseline CPU implementation. We compare our coloring results to a prior parallel algorithm on the same set of graphs [from where] [reference?].

The remainder of this paper is structured as follows: we first review previous graph coloring work will be presented. The following section discusses the details of the parallel algorithm and the coloring heuristics we evaluated. We present extensive experimental results followed by a conclusion.

2. Target platform

GPUs have different architectures compared to parallel computers where most of the parallel graph-coloring algorithms are run. The main differences and how they impact on algorithm are described below: i) Number of processors. Graphics cards have many processors: an NVIDIA GTX 260 has 192 cores while a Tesla S1070 processor has 240 cores. This allows us to have much more parallelism (and having many more threads running at the same time) compared to other architectures. Moreover, as thread creation is inexpensive on a GPU, we will be running a lot to fully use the power of the GPU. Consequently our algorithm will be using many more threads than existing algorithms.

ii) Processor speed. While a state-of-the-art processor typically operates at around 3 GHz processor cores on GPUs are much slower: one core on a GTX 260 operates at 1242 MHz while a core on a Tesla S1070 operates between 1.296 to 1.44 GHz. Hence we need to use the many threads to match or better the speed of a standard CPU. Moreover, trying to get speedups for O(N) algorithms are hard as the cores are individually slower and some time is wasted copying data to and from the GPU

iii) Memory. The memory available to parallel computers is normally bigger but it is sharing data between processors is usually much more expensive. On a GPU, all the data can be stored in the global memory and all the cores can readily access it.

3. Related Work

In this section we give an overview of existing graph coloring algorithms, both sequential and parallel. In this discussion, a graph $G = (V, E)$, where V is a set of n vertices, and E is a set of m edges. The degree of a vertex $v \in V$ is denoted by $d(v)$. The maximum vertex degree of a graph is denoted by Δ.

3.1 Sequential greedy coloring

The problem of sequential graph coloring has been studied extensively. Among all the sequential approaches, greedy algorithms are proven to be very effective in practice, such as First Fit, Largest degree ordering, Saturation degree ordering and Incidence degree ordering[citations?]. Such greedy algorithms iterate over the vertices to be colored in a certain order, and at each step assign the current vertex the smallest permissible color. Given a graph G with maximum degree Δ, these procedures will use a maximum of $\Delta+1$ colors since each vertex has at most Δ neighbors. We now describe each of these greedy coloring algorithms in more detail.

First Fit (FF): This is the simplest algorithm of all greedy coloring heuristics[**?**]. To color a vertex, a permissible color is chosen from the interval $[1, C]$, where C is the largest color currently used. A new color $C + 1$ is used if all C colors have been assigned to the neighbors of the vertex. FF runs in time $O(m + n)$.

Largest degree ordering (LDO): LDO first sorts vertices by degree in nonincreasing order and then runs First Fit. It runs in time $O(m + n \log n)$.

Saturation degree ordering (SDO): The saturation degree of a vertex is defined as the number of different colors its neighbor vertices have been assigned. SDO greedily picks the vertex with largest saturation degree and colors it. Since this step could change the saturation degree of other vertices, SDO is more computationally intensive; a naive implementation takes time $O(m^2/n)$, and this can be reduced to $O(m \log n)$ using hash tables and dynamic search trees. Notice that the naive implementation runs in time $O(n^2)$ for sparse graphs.

Incidence degree ordering (IDO): The incidence degree of a vertex is defined as the number of its adjacent colored vertices. The IDO algorithm always picks the vertex with maximum incidence degree to be colored next. It runs in time $O(m \log n)$.

In general, First Fit is more efficient than the other heuristics, but uses more colors. SDO returns the fewest colors of all the heuristics[**?**], while being more expensive. Since it is often the case that many vertices have the same saturation degree, tiebreaking rules must be employed to pick the next vertex to color. A useful choice of tiebreaker is to use LDO for tiebreaking[**?**]; Algorithm **??** describes the resulting procedure that we term SDO & LDO.

Algorithm 1 SDO & LDO

```
SDO & LDO algorithm:
while $N_{colored} < n$ do
  $max = -1$
  for $i = 1\ to\ n$ do
    if !$colored(v_i)$ then
      $sd = SaturatedDegree(v_i)$
      if $sd > max$ then
        $max = sd$
        $index = i$
      if $sd = max$ then
        if $d(v_i) > d(v_{index})$ then
          $index = i$
      $Color\ v_{index}$
      $N_{colored}++$
```

3.2 Parallel graph coloring

A number of existing parallel algorithms are based on the concept of a maximal independent set (MIS), which is a maximal set of vertices in which no two are connected by an edge. Since all such vertices can be colored with the same color, a parallel graph coloring algorithm repeatedly finds an MIS, colors it, and then removes the vertices from the graph. Luby [**?**] first proposed a randomized parallel MIS algorithm that yielded an independent set in $O(\log n)$ parallel steps; this was subsequently improved by Jones and Plassman[**?**] to $O(\log n/ \log\log n)$ paralle steps for bounded-degree graphs.

Gjertsen*et al* [**?**] and Allwright*et al* [**?**] adapted this idea for distributed memory architectures. Graphs are uniformly partitioned into subgraphs and each subgraph is associated with a specific processor. To avoid color conflicts, two adjacent vertices on different processors cannot be colored in the same time step. This restriction can be overcome by Johanssons algorithm[], where each processor is assigned a single vertex and vertices are colored randomly in the same time step. Color inconsistency may occur in this scheme, so the coloring procedure is recursively re-run for the vertices receiving illegal colors.

Gebremedhin and Manne proposed a parallel coloring algorithm algorithm (G-M) for shared memory architectures, which achieves a linear speedup in the PRAM model [**?**]. In followup work [**?**] they improved the G-M algorithm by ordering the vertices to reduce cache misses and performing vertex-to-processor assignments based on graph partitioning rather than arbitrary allocation.

The G-M algorithm consists of four phases: (1) Partitioning: Assuming the number of processors is p, in this phase the graph is broken down into p independent subgraphs of almost the same size, and each subgraph is assigned to one processor. (2) Pseudo-coloring: Each processor tentatively colors the vertices assigned to it in sequence. When two adjacent vertices reside on different processors are colored simultaneously, coloring inconsistency may arise. (3) Conflict detection: Each processor checks the colors of vertices assigned to it for consistency and identify a set of vertices which have color conflicts. (4) Conflict resolving: Re-coloring the detected vertices with conflicts.

4. Algorithm

In this paper, we propose a framework based on the G-M algorithm, which is adapted for an NVIDIA GPU platform. In addition, two new heuristics are proposed to match the parallelism exploited by the GPU, which are shown to give better results than existing heuristics such as FF and SDO.

4.1 The Graph Coloring Framework

Algorithm 2 Graph Coloring Framework

```
phase 1 : $Graph\ Partitioning - CPU$
  Logically partition graph into subgraphs
  Indentify boundary nodes
  Count neighbors outside the subgraph for each vertex

phase 2 : $Graph\ Coloring\&Conflict\ Solving\ - GPU$
  while Number of color conflicts is high do
    Color graph using the specified heuristic
    Identify color conflicts

phase 3 : $Sequential\ Conflicts\ Rsolution\ - CPU$
  Residual conflicts are eliminated
```

Graph Partitioning: The graph is initially padded with empty nodes so that it is exactly divisible by the total number of threads. We experiment with two different graph partition approaches: 1) partitioning the graph into arbitrary regular subgraphs; 2) using a graph partitioning system like METIS[Karypis] to achieve a better partition (in which case the graph is not padded). In both cases the number of partitions is set to be equal to the number of threads used, and each thread is responsible for coloring the subgraph assigned to it. For uniform partition, to color a graph of N vertices using p threads, each thread equally takes N/p vertices, which is a load-balanced approach.And when METIS is used, the resulting subgraphs tend to be unevenly sized. If some threads deal with much bigger graphs than the others do, the overall performance will be impacted.

Boundary Nodes: When a graph is divided into subgraphs, some of the vertices found in a subgraph have connected neighbors only in the same subgraph while others have neighbors outside.The former are referred as interior nodes and the latter are boundary nodes.Within a subgraph, the coloring procedure is conducted in sequence, so interior nodes are not likely to have color conflicts. However,since all subgraphs are colored in parallel without inter-thread communication, the boundary nodes are likely to be assigned with illegal colors conflicting with their neighbors residing in different subgraphs. For many graphs, especially if METIS is not used and the partitions are small, the number of boundary nodes is roughly equal to the total number of vertices. However, it can help speedup in some case and when heuristics like MIN OUT and MAX OUT are used, we need to know the number of neighbors of a vertex which is outside the subgraph.

Phase 2: Graph Coloring & Conflicts detection:

Graph Coloring Each thread assigns colors to its subgraph but checks the whole graph before allocating colors. We tried four heuristics for color allocation:

i) First Fit

ii) SDO & LDO

iii) MAX OUT

iv) MIN OUT

First Fit and SDO & LDO have already bee described in Section 3.1. We will focus on the 2 new heuristics we have devised:

Algorithm 3 Max Out Algorithm

```
while $N_{colored} < N$ do
  $max = -1$
  for $i = 1\ to\ N$ do
    if $!colored(v_i)$ then
      $no = Numberofneighborsoutsidepartition$
      if $no > max$ then
        $max = no$
        $index = i$
      if $no == max$ then
        if $d(v_i) > d(v_{index})$ then
          $index = i$
      $Color\ v_{index}$
      $N_{colored} + +$
```

Algorithm 4 Min Out Algorithm

```
while $N_{colored} < N$ do
  $min = maxdegree$
  for $i = 1\ to\ N$ do
    if $!colored(v_i)$ then
      $no = Numberofneighborsoutsidepartition$
      if $no < min$ then
        $min = no$
        $index = i$
      if $no == min$ then
        if $d(v_i) > d(v_{index})$ then
          $index = i$
      $Color\ v_{index}$
      $N_{colored} + +$
```

For both of the above algorithms as well as for SDO & LDO, when we have chosen a vertex to color, we consider all the neighbors of the latter both inside and outside the subgraph.

Conflicts Detection Given that coloring is being done in parallel, conflicts occur frequently. They are identified by assigning one thread per boundary node that checks whether there is a conflict with one of its neighbors. If that happens, the color of that node is reset to NULL.

Dropping to the CPU to solve the conflicts is common in many approaches but given that we have many small partitions, we tend to have lots of conflicts and resolving that on the CPU can be quite slow. So we do several passes of coloring and conflict detection until we have few conflicts which can be quickly solved sequentially on the CPU.

Name	n	m	Δ	Avg Degree
Structural Engineering				
ct20stif	52,329	1,375,396	206	50
nasasrb	54,870	1,366,097	275	47
pwtk	217,918	5,926,171	179	52
Civil Engineering				
pkustk10	81,920	2,114,154	89	52
pkustk11	87,804	2,565,054	131	55
pkustk13	94,893	3,260,967	299	68
Automotive				
hood	220,542	5,273,947	76	47
msdoor	417,792	9,912,536	76	46
ldoor	958,464	22,785,136	76	46

Table 1. Graph properties and source

Name	Colors	Time	Colors	Time
	FF	FF	SDO&LDO	SDO&LDO
ct20stif	49	29	48	203387
nasasrb	41	29	36	646113
pwtk	48	116	48	5000000
pkustk10	42	39	42	226838
pkustk11	57	48	51	250112
pkustk13	57	66	51	612960
hood	42	105	36	1709180
msdoor	42	197	35	7281740
ldoor	42	450	35	34584800

Table 2. Sequential coloring results for FF and SDO&LDO

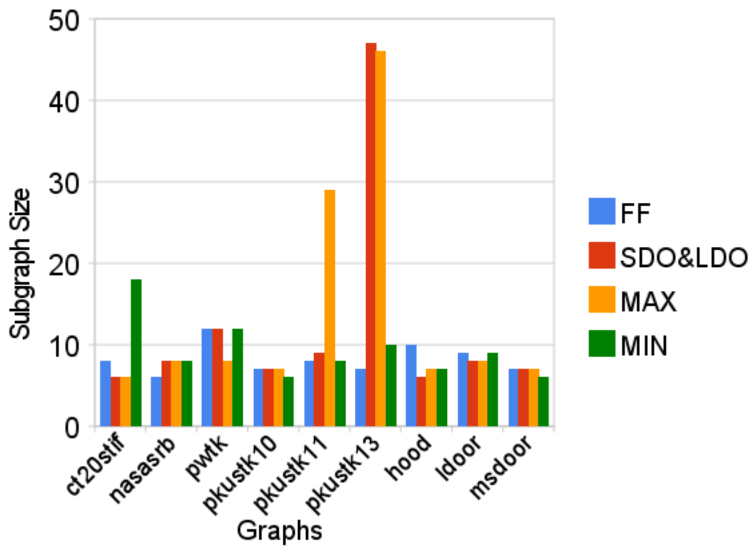

Figure 1. Subgraph size for Optimal color per algorithm and graph

Parallel Algorithm	Average	Standard Deviation
FF	8.2	1.9
SDO&LDO	12.2	13.2
MAX OUT	14.0	14.0
MIN OUT	9.3	3.8

Table 3. Subgraph size for optimal coloring

5. Experiments and Results

The algorithm has been implemented using the CUDA API and the tests were carried out on a Tesla S1070. The configuration is as follows:

- Number of Cores:
- Memory:

The tests were carried out on these 9 real graphs from the University of Florida Sparse Matrix Collection [] which were regrouped in 3 categories

Say more about the graphs. Range of similar average degree. Variation in max-degree and size. Similar number of colors, but size impacts execution time significantly due to polynomial algorithm. etc. etc.

Sequential First Fit and SDO & LDO were run on each of the above graphs to get a color and timing baseline.

While the colors are accurate, the timing information for the sequential SDO & LDO could have been improved. The latter has is from a naive implementation which is quite inefficient. Nevertheless, the sequential SDO & LDO has been reported by its authors [...] to be O (N^2) and despite yielding a good coloring is quite slow. On the other hand, First Fit which is fast yields a poor coloration! Ideally what we would really want is a fast algorithm that generates as few colors as possible.

5.1 Algorithms Compared

To investigate the impact of parallel graph coloring algorithms on both performance and number of colors obtained, the number of threads is linearly incremented and the timing and color obtained is compared.

Number of threads Given that the graphs have very different sizes, it does not make sense to use the same number of threads for each. For example, for a graph of 50 000 nodes, if 1024 threads are used, each thread handles about 48 vertices while for a graph of 500 000 nodes, each thread will operate on around 480 vertices. Since GPU cores are much slower than state-of-the-art processors speedup gains are the result of using as many cores as possible where each is operating on just a few vertices.

To determine subgraph sizes that tend to give fewer colors, we linearly decreased the size of the subgraph by varying the number of threads for each of the 9 real graphs that we have. The results is shown in Figure **??** and Table **??**.

5.2 Multi Passes

In the second set of experiments, we allow the program to determine the number of GPU passes it requires. We have seen that if the CPU has fewer than 200 conflicts to solve it is very fast(It is better to use a percentage, like percentage of the total number of vertices, but the stop criterion dose not have math). (How did you arrive at this number of 200?) - can we instead say that we move over to CPU when the number of remaining conflicts is negligible So after each pass we check to see the number of conflicts and if it is approximately less than 200, we terminate the GPU Passes and send the remaining conflicts to the CPU for solving.

Refer to the figures. Need better labels (graph names rather than a,b,c). SDO&LDO is approximately twice as fast as compared to MAX or MIN. However, the number of colors for MAX and MIN is usually better than SDO&LDO, with MAX nearly consistently yielding fewer colors. For this set of graphs, the MAX heuristic also yields fewer colors that the sequential SDO&LDO algorithm and the sequential First Fit (FF: 42, 57, 57 - SDO: 42, 51, 51 - Best form other Paper: 44, 57,57 removed) Also should talk about relationship between number of threads and colors, and number of threads and performance. Where in these graphs does the performance associated with 8 vertices per subgraph fall? Need to make a connection back to that.

For this second set of tests, we see that though we easily beat the sequential First Fit in terms of color allocated, for ct20stif and nasasrb, we do not get consistently get fewer or match the colors of sequential SDO&LDO while for the pwtk graph, we are well below the 48 colors obtained from running sequential SDO. Do you have a sense why you are using fewer colors?

For ct20stif, in our 10 test runs we sometimes get 47 colors which beat the sequential SDO implementation but on average we get more colors and the best performances are obtained with parallel First Fit. Where is non-determinism coming from? The conflict resolution? You should probably elaborate on that. (FF: 50, 47, 52 - SDO: 48, 36, 48 - Best from other Paper: 47,38,41)

5.3 Using METIS

METIS is a set of algorithms that can be used to partition a graph into subgraphs. Many parallel graph algorithms advise the use of METIS to have better partitioning. However these algorithms run on parallel machines and they do not divide graphs in as many partitions as we do. Consequently when we tried to run METIS to partition the graph as we need, the resulting partitions were very unbalanced. The figure below shows the subgraph sizes from metis.

5.4 Conclusion

5.5 Reference

A. Appendix Title

This is the text of the appendix, if you need one.

Acknowledgments

Acknowledgments, if needed.

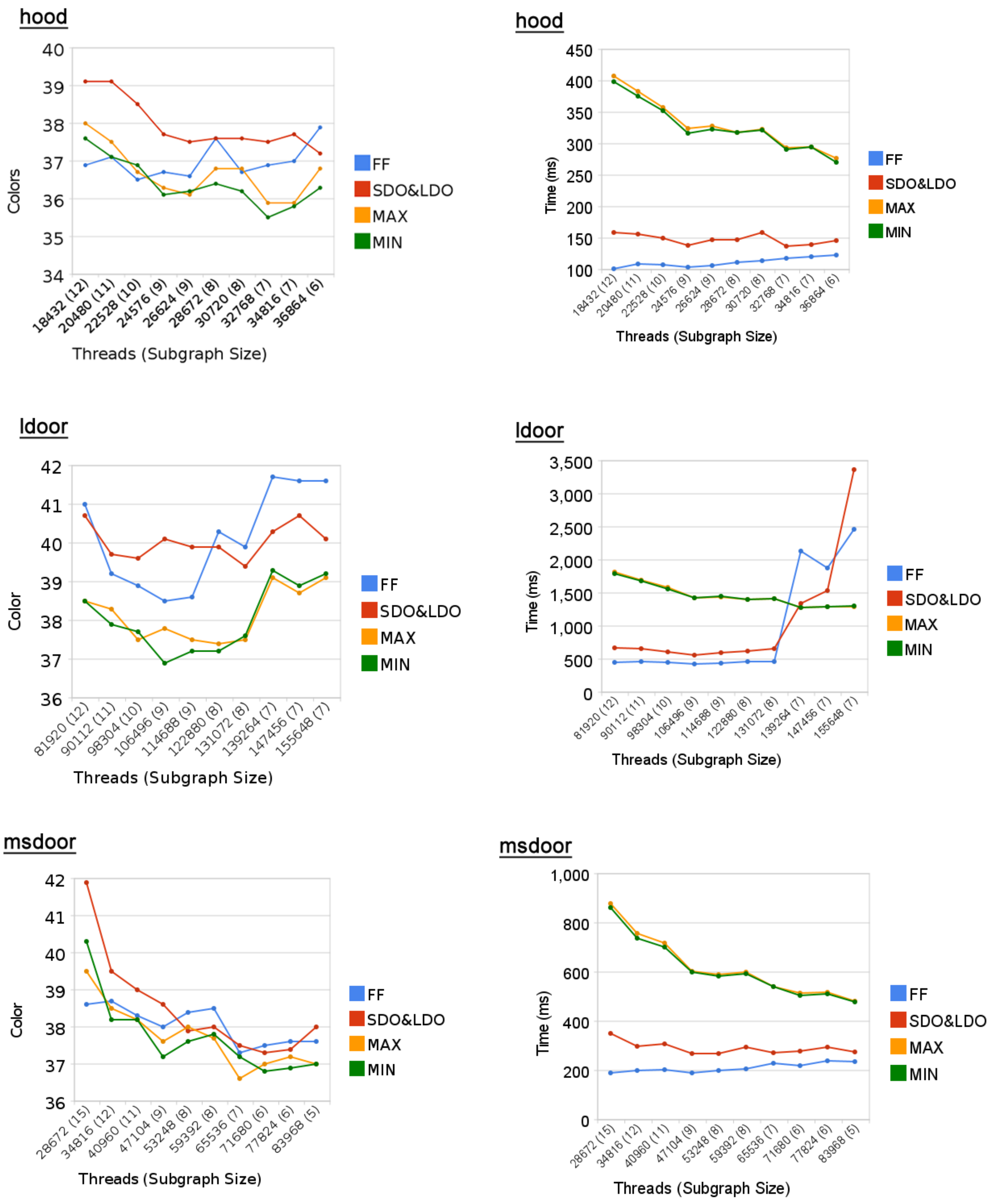

Figure 2. Automotive

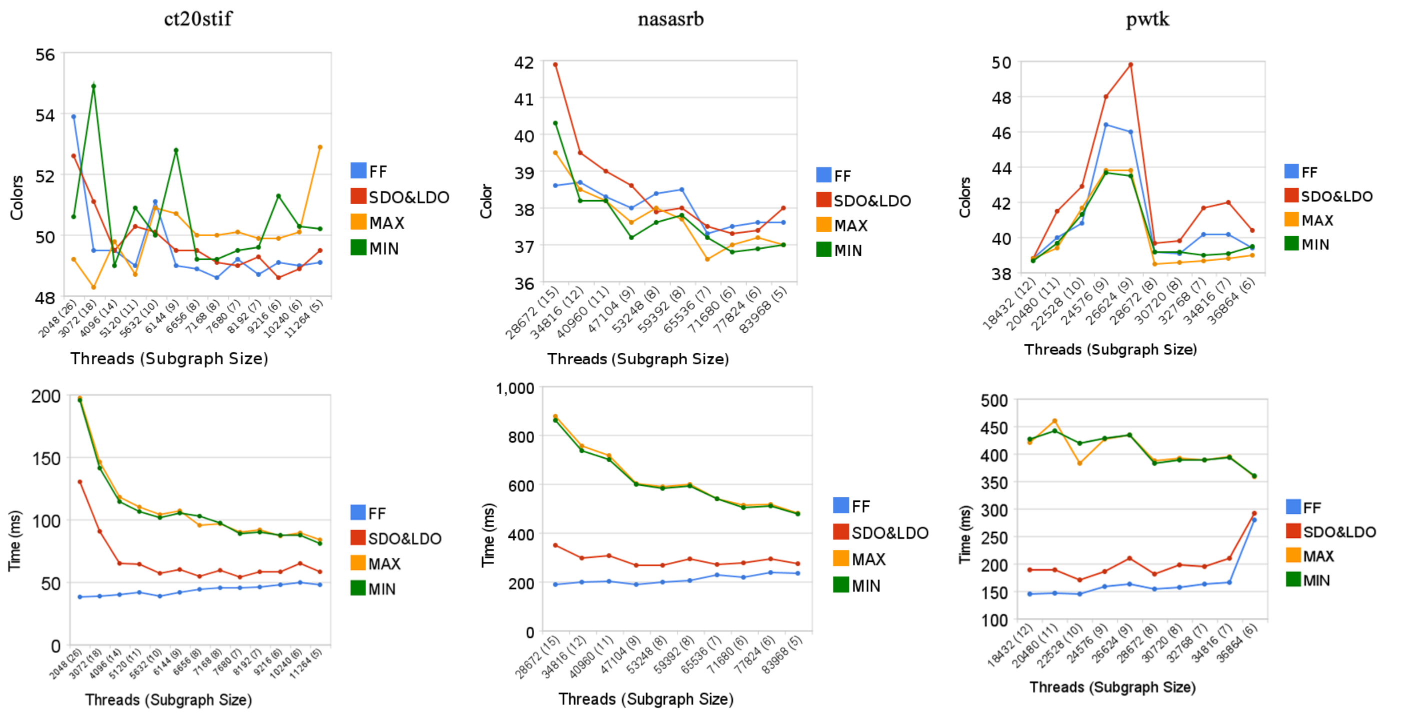

Figure 3. Structural

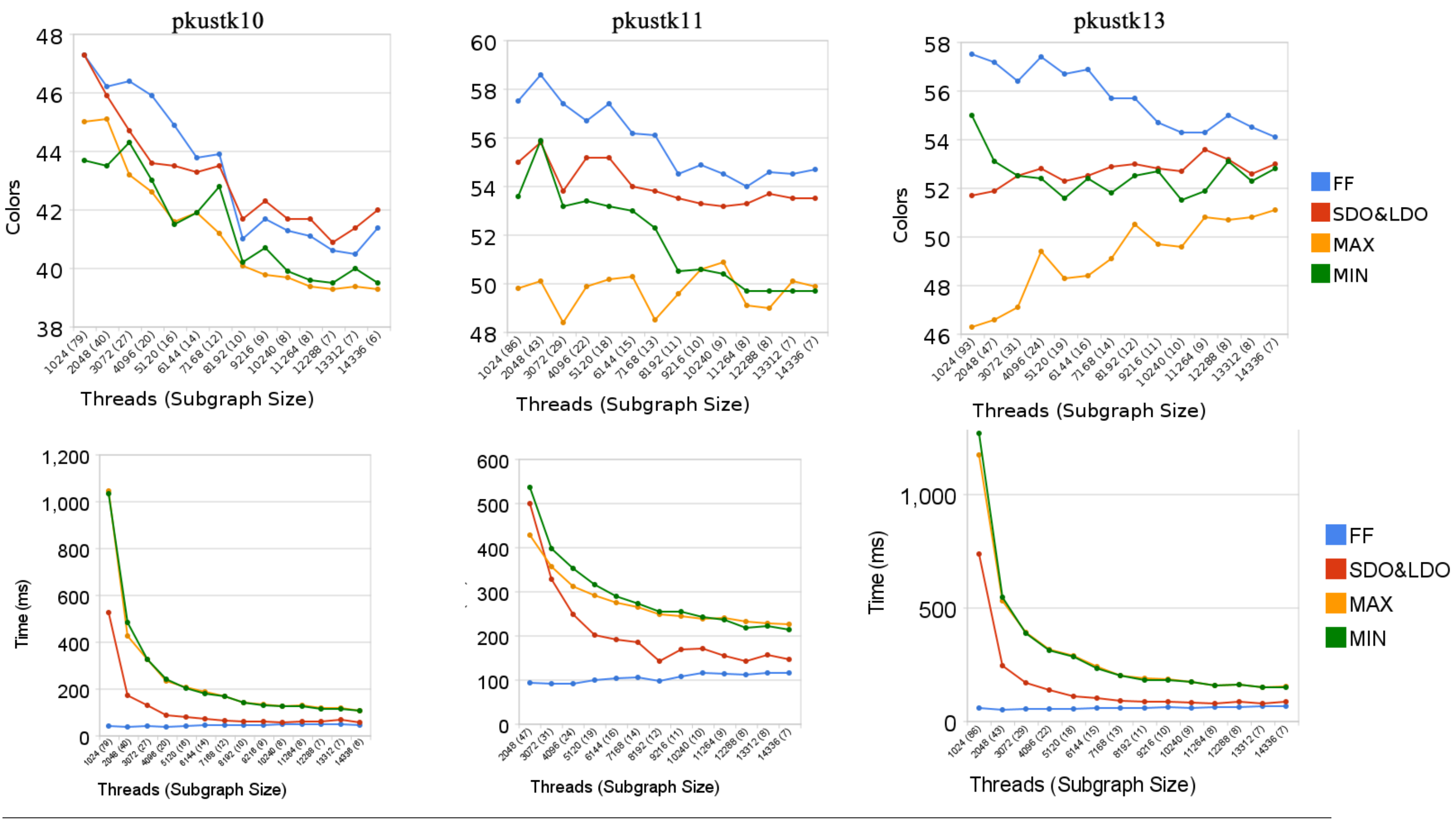

Figure 4. Civil

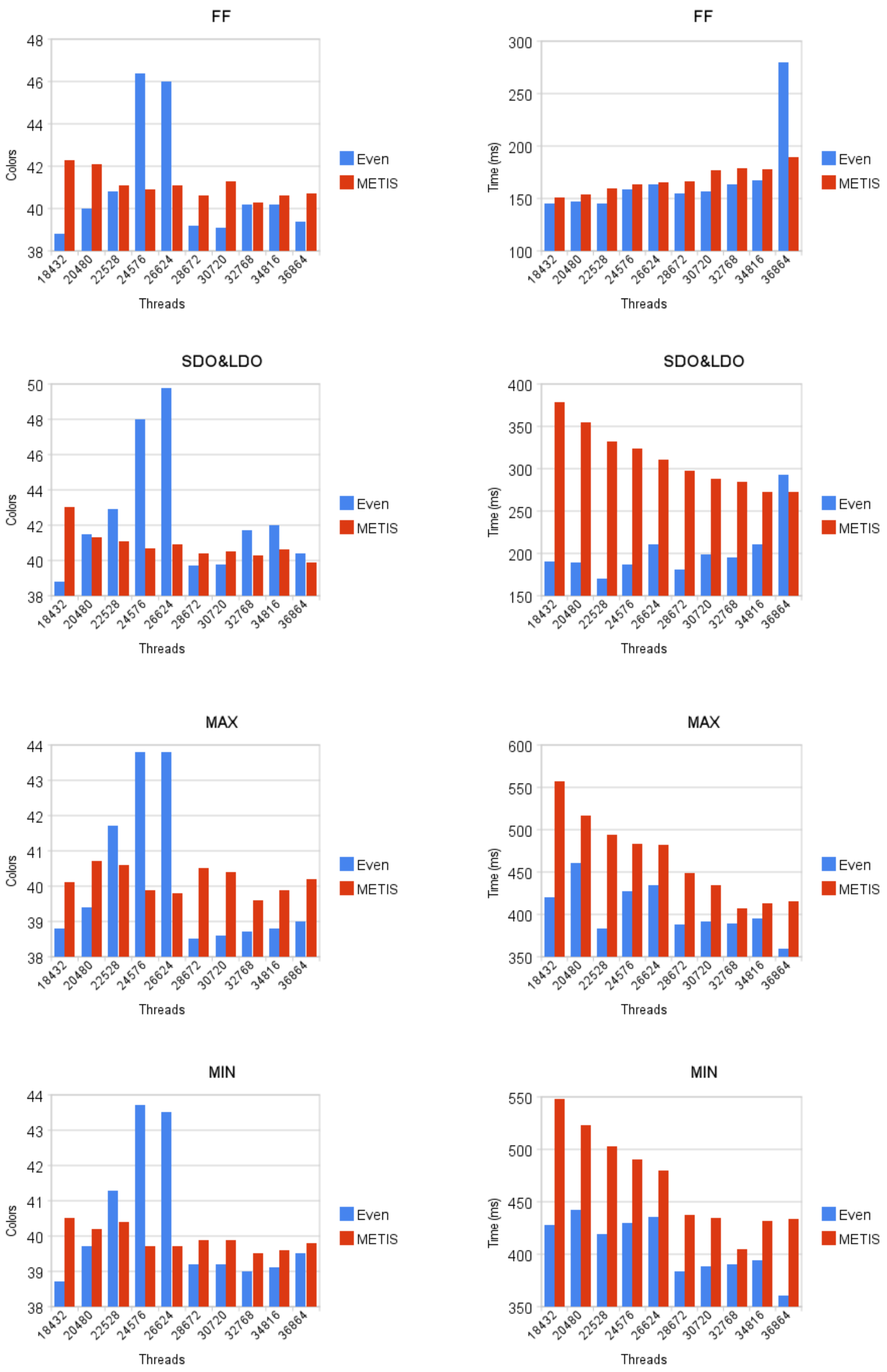

Figure 5. Pwtk Metis

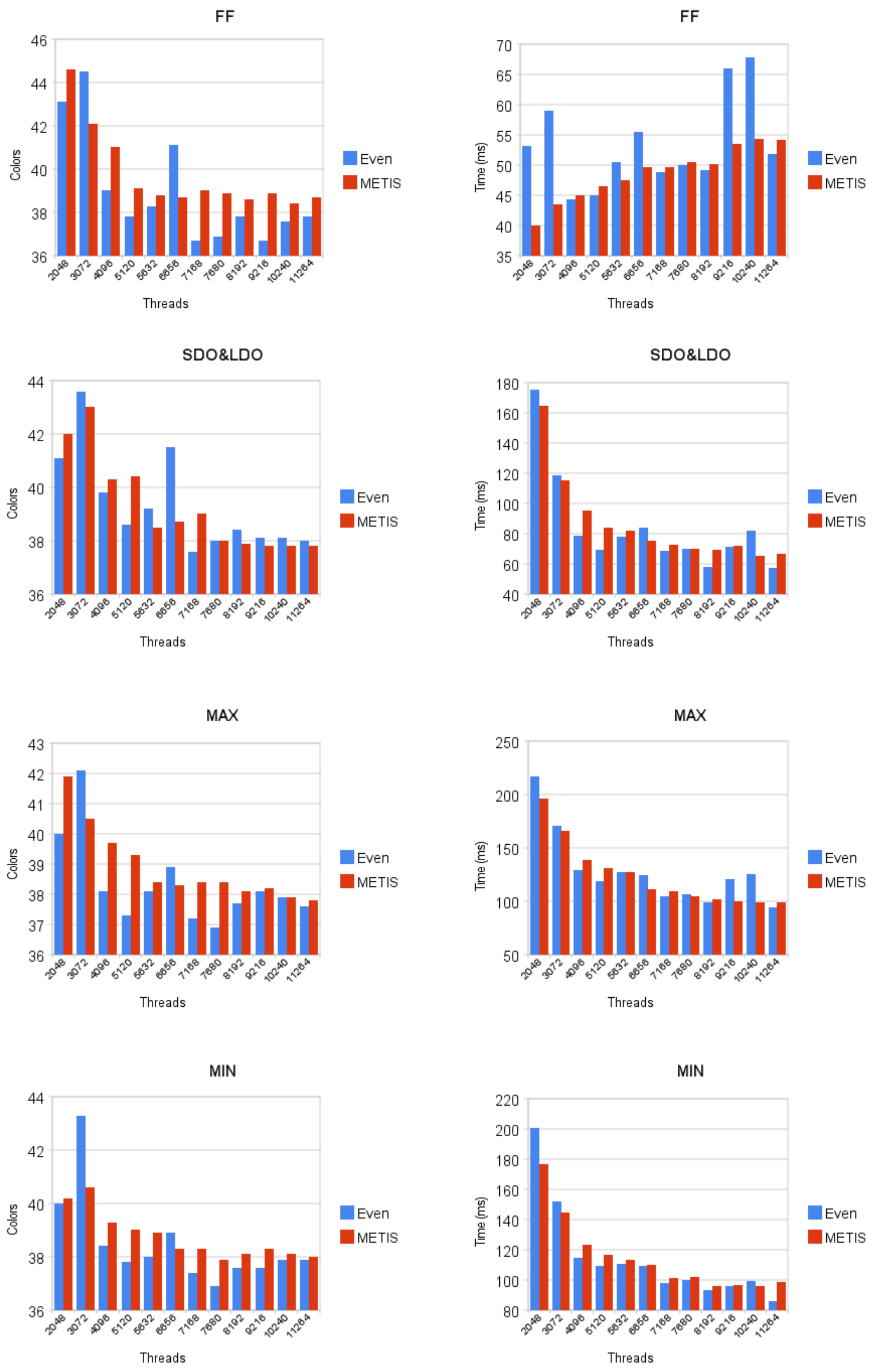

Figure 6. Nasasrb Metis