Lock-Free Parallel Feedback Vertex Set Selection

Daniel Thuerck and Michael Goesele

Graduate School of Computational Engineering, TU Darmstadt

May 23, 2017

Abstract

For large, general MRF MAP problems, the solver mapMAP [1] has been shown to effectively use parallel hardware for generating good solutions in less time than the prior state-of-the-art. With ever-broader SIMD registers and more and more compute cores, its core computational block of *dynamic programming* stands to profit – though at some point, the tree selection procedure can become a bottleneck for scalability. In this short report, we describe an alternative tree sampling procedure that proves to be faster, resulting in better run-time behavior for medium and large datasets.

1 Introduction

Given an

- undirected graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$,
- a *label set* $\mathcal{L}$ and
- *cost* functions $D_v : \mathcal{L} \to \mathbb{R}_0^+$ for all $v \in \mathcal{V}$ as well as $V_e : \mathcal{L}^2 \to \mathbb{R}_0^+$ for all $e \in \mathcal{E}$

the *MRF MAP* problem is finding an *assignment* $f : \mathcal{V} \to \mathcal{L}$ minimizing

$$E(f) = \sum_{v \in \mathcal{V}} D_v(f_v) + \sum_{e=\{w,u\} \in \mathcal{E}} V_e(f_w, f_u) \ . \quad (1)$$

The general case (arbitrary graphs $\mathcal{G}$, arbitrary cost functions) is $\mathcal{NP}$-hard. There are many heuristics and even more implementations that attempt to find good solutions to posed problems, especially for solving labelling problems in the computer vision domain. One solver is our package mapMAP [2], released under the liberal 3BSD license as open source [1]. This tech report mainly focuses on improving one of its building blocks' performance. Thus, the following section gives a short overview over mapMAP's core computational steps and identifies a main bottleneck tackled in this report.

2 mapMAP and its Scalability

Compared to mapMAP's release paper [2], some names and definitions are changed, adopting expressions more commonly used in the graph algorithms community. Ignoring the multilevel and spanning tree heuristics previously described, the two core steps in each iteration are the following:

1. Sample an *acyclic coordinate set* on the graph $\mathcal{G}$, resulting in a forest of independent trees.
2. Execute a step of block-coordinate descent by using dynamic programming to optimize the cost function (1) on each subtree, updating the current assignment.

First step. From a bird's eye perspective, mapMAP is a block coordinate descent (BCD) scheme optimized for (massively-) parallel systems. Putting aside the motivation for the coordinate selection procedure in step 1, we point out that the tree selection can easier

be formulated by focusing on its inverse: Instead of looking at the nodes it puts into tree, we regard the nodes it *leaves out* of the trees. Indeed, these form a so-called *feedback vertex set*.

Definition 1. *A* feedback vertex set *for $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ is a set $\mathcal{V}' \subseteq \mathcal{V}$ such that $\mathcal{G}[\mathcal{V} - \mathcal{V}']$ is acyclic.*

Put differently, a feedback vertex set is a set including at least one vertex of each cycle in the *simple* graph (ignoring those of length 2). It is straightforward to see that $\mathcal{G}[\mathcal{V} - \mathcal{V}']$ is exactly the tree decomposition we desire. Thus, each maximal acyclic coordinate set can be derived from a minimal feedback vertex set (and vice versa).

Second step. After sampling such a forest, we proceed with optimizing the subproblems by dynamic programming on the individual trees. Per tree $\mathcal{T}$, this yields an $\mathcal{O}(|\mathcal{T}||\mathcal{L}|^2)$ algorithm, which traverses the tree in two passes: bottom-up and top-down. For parallelization, we process each pair of nodes in different paths from the tree's root node in parallel, given that all its children have already been processed. By the nature of dynamic programming, this algorithm benefits tremendously from vectorization; consequently, our code selects the maximum possible vectorsize on the build system, exploiting instruction sets such as SSE or AVX1/2. For specifics on the algorithm, please refer to the previous paper.

Scalability. For large datasets, the optimization step scales almost linearly with the number of CPU cores until a certain point. Given the parallelization mentioned above, the limiting factor is the longest path from the root in each tree. Since this path must be processed serially, it constitutes a *roofline* for scalability. Consequently, the proposed coordinate selection algorithm is biased towards building shallow, but broad trees to avoid lengthy paths in each tree.

As long as the number of cores used does not cross that barrier, only the coordinate selection remains as a limiting factor for scalability. And indeed, the more cores are used, the more relative time per iteration is spent on coordinate selection: Figure 1 shows a breakdown of time spent in coordinate selection vs. time spent in dynamic programming for the datasets `planesweep_1280_1022_96` and `citywall-100`[1] (representing medium and large datasets), over the number of threads used. Each iterations' total wall clock time is scaled to 100%. Both experiments show a similar thread: With an increasing number of cores used, the percentage of runtime spent in coordinate selection (henceforth named FVS) grows from approx. 5% to almost 30%, confirming its inferior scalability.

From the algorithmic description of the formely proposed FVS algorithm (rechristened *optimistic* FVS), one step becomes the immediate suspect for the runtime behavior: conflict resolution and rollback. Following its randomized nature, we would expect approx. half of the newly added but conflicted nodes in each iteration need to be removed, including rolling back all marker changes. The serial work involved per node equals the work in the second phase (*check*) of the algorithm. With only few nodes to be rolled back and many cores in use, this results in a situation where many cores are left unused yet the wall clock time for conflict resolution equals that of the previous phase. As a result, compute resources are used badly and this phase becomes a bottleneck.

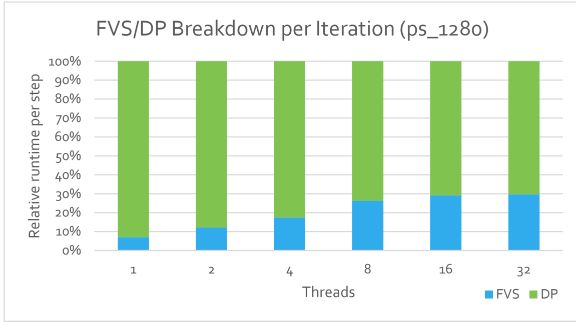
Given that analysis, the immediate point of attack for improving the FVS selection is the conflict resolution and rollback phase.

3 A new Feedback Vertex Set Algorithm

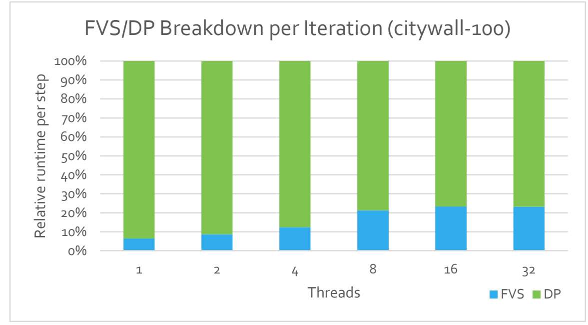
In the remainder of this paper, we describe a way to handle possible conflicts statically, with an acceleration structure built once for each graph. This allows us to decouple conflict handling from FVS selection and delivers an algorithm that is up to 3 times faster than optimistic FVS.

Static conflict resolution. Figure 2 (b) illustrates the issue mentioned in the last section: Assume that optimistic FVS's status is as depicted in (a). 5 nodes

[1]Both datasets and the benchmarking environment are described in detail in section 4.

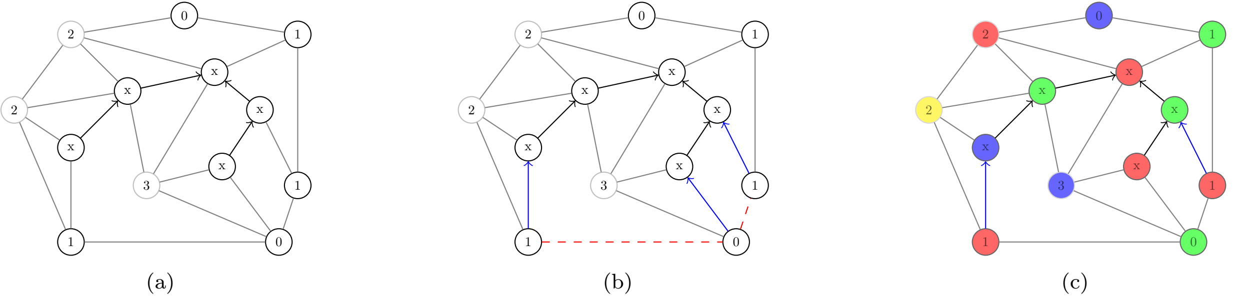


(a) `plane_sweep_1280_1022_96`



(b) `citywall-100`

Figure 1: Relative time per BCD iteration spent in finding the feedback vertex set (FVS) and executing the dynamic programming (DP) steps. With more and more threads involved, the limited parallelism of FVS becomes noticeable.



(a) (b) (c)

Figure 2: Conflict resolution principles: (a) shows the current grown tree with markers and nodes in tree (x), (b) a three-way conflict that has to be handled by optimistic conflict resolution. (c) restricts the nodes added to the tree in this round by restricting them to one color. Newly added edges are colored in blue.

have been included into a tree so far and all markers have been set accordingly. In the next iteration, the lower three nodes can all be added to the tree in parallel following their markers – which, as (b) shows, leads to two conflicts (illustrated by dashed red lines). The minimal conflict resolution would be to roll back the second node, resulting in a rollback phase that uses parallel hardware poorly.

Ignoring all nodes but the three conflicted nodes, we notice that every valid addition to the tree in that step constitutes an *independent (sub-)set* of those nodes.

Definition 2. *A set of nodes* $\mathcal{V}' \subseteq \mathcal{V}$ *is called* independent *in a graph* $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ *iff* $\mathcal{G}[\mathcal{V}']$ *has no edges.*

As it turns out, the proposed conflict resolution phase *is* one way of computing such an independent set. Instead of dynamically computing these sets whenever there is a conflict, we now propose a *static* way of handling conflicts.

First, notice that given a *coloring* of the graph, each color (resp. nodes colored with it) forms an independent set.

Definition 3. *A* coloring *of a graph* $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ *with colors* $\mathcal{C}$ *is a function* $c : \mathcal{V} \to \mathcal{C}$ *such that* $\forall e = \{v_1, v_2\} \in \mathcal{E} : c(v_1) \neq c(v_2)$.

Therefore, whenever new nodes are added to a tree, there can be no conflict as long as all new nodes have the same color. Figure 2 (c) depicts that situation: The three nodes from before are now colored by red and green. If either the green or both red nodes are included, no conflict occurs. Thus, no rollback is necessary. These insights allows us to decouple conflict resolution from the FVS procedure: A graph coloring is computed *once* per graph and can be used in every FVS computation after that.

Algorithm 1 Pseudocode for the colored FVS procedure. The code is for *one thread* `id` and assumes the round color is c_r.

```
1: i ← w_{c_r}(id)
2: Phase I: grow a new branch
3: if maker(i) < 2 then
4:     select random neighbor j that is already in the tree
5:     p(i) ← j
6:     put i into w_new
7: end if
8: Phase II: update markers and fill queue
9: i ← w_new(id)
10: for j ∈ N(i) do
11:     maker(j) ← maker(j) + 1 (atomically)
12:     if j not in tree and in_queue(j) == false then
13:         put j into queue w_{c(j)}
14:         set in_queue(j) ← true (atomically)
15:     end if
16: end for
```

Changes to the FVS algorithm. Implementing that idea requires two major changes to optimistic FVS, resulting in the pseudocode for tread-wise execution of Algorithm 1. Following the idea of using graph colorings, we call the resulting algorithm *colored FVS*.

First, the immediate consequence of using coloring is dropping the third phase of optimistic FVS. Conflicts are avoided by restricting all operations to one color; hence no rollback has to be performed.

Second, we maintain more than just one queue (optimistic FVS had one input queue $w_{\texttt{in}}$), but one queue per color. Optimistic FVS kept nodes trying to become parents in the queue; colored FVS however cannot use that notion: The color of parents does not help to avoid conflicts. Thus, the queues now contain potential *children*; in each round, all nodes in a pure-colored queue find their parent independently. Since each node can have multiple children, this also removes the necessity of locking nodes as in optimistic FVS (as long as each node is only in a queue once, which we check by using atomics). The implementation of multiple queues is based on one atomic counter per queue; their sum is bounded by $|\mathcal{V}|$. Each iteration buffers new nodes in $w_{\texttt{new}}$ and can overwrite the queue of its round's color immediately. With this, only one queue per color has to be maintained.

Dataset	$\lvert\mathcal{V}\rvert$	$\lvert\mathcal{E}\rvert$
`plane_sweep_320_256_96`	81,920	163,264
`plane_sweep_1280_1022_96`	1,308,160	2,614,018
`citywall-100`	8,164,823	12,240,438
`pisa-100`	25,975,386	38,959,807
`vertex_coloring_15M_10D`	15,000,000	71,135,959

Table 1: Datasets used for benchmarking.

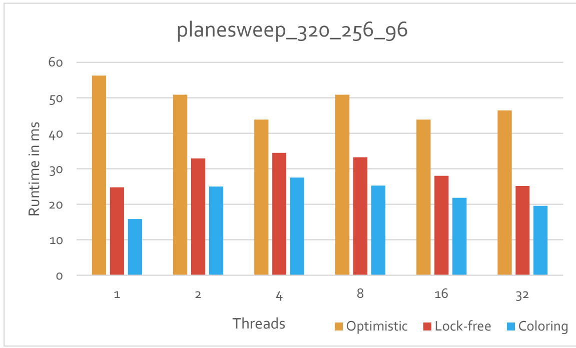
These changes result in a dramatically simplified implementation. No locks or even spinlocks are necessary, never is a decision to add a node reversed.

4 Performance Evaluation

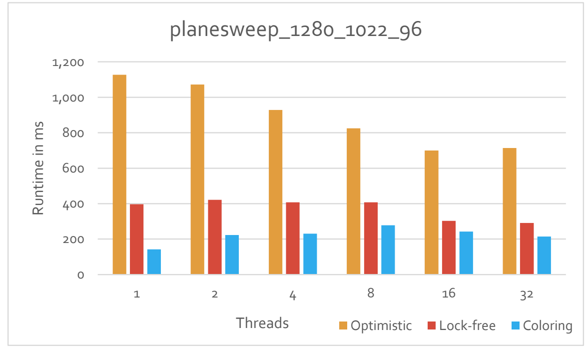
We now compare our implementations of optimistic FVS ([2], version 1.1) against the new colored FVS (same code, branch `lock_free_fvs`) on different datasets (see Table 1) from our original paper [2]. For coloring, a straightforward reimplementation of IPGC [3] is used and included into our codebase. In each iteration of colored FVS, the color was is chosen at random among all colors with nonempty queues. In all benchmarks, the algorithms were executed 5 times after one warmup iteration to alleviate their randomized nature. The times displayed are averaged over all benchmark iterations.

All benchmarks were executed on a dual-socket system with 2x Intel(R) Xeon(R) E5-2687W v3 clocked at 3.10GHz and 128 GB RAM, running Ubuntu 16.04. To exclude the influence of multi-processor synchronization and memory copying, we used `taskset` to restrict our process to the first physical CPU and its 10 physical cores. Hyperthreading was activated at all times. No other major processes were ran at the same time. Times were taken using C++11's `chrono` functionality with `ms` granularity.

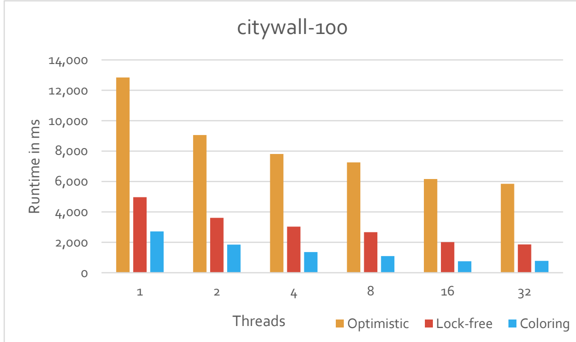
The results – i.e. wallclock time for optimistic FVS, colored FVS as well as the IPGC coloring itself – are depicted per dataset in Figure 3. Note that the time for colored FVS does not include the time for coloring, which is merely a preprocessing step. Clearly, this coloring pays off as an *acceleration structure* if FVS' are sampled more than one time.



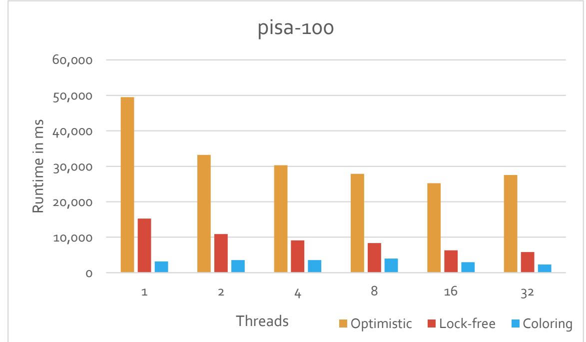
(a) `plane_sweep_320_256_96`



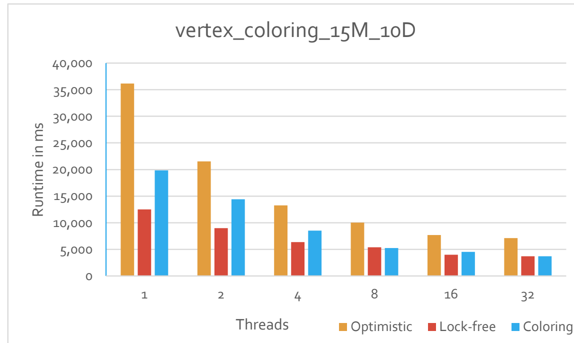
(b) `plane_sweep_1024_1022_96`



(c) `citywall-100`



(d) `pisa-100`



(e) `vertex_coloring_15M_10D`

Figure 3: Performance (execution time) for both FVS algorithms as well as the IPGC graph coloring with different number of threads involved.

All in all, the benchmarks show that – contrary to our hypothesis – we could not improve the scalability much. The development of runtimes with an increasing number of threads mirrors that of optimistic FVS. Deeper inspections show that we are now limited by the number of nodes getting added in each iteration. This number, compared to scheduling overhead for more threads, is essentially the bottleneck for better scalability.

However, all benchmarks on medium and large datasets ((b) - (e)) also show a clear advantage in the order of a 50% up to 350% speedup of colored FVS over optimistic FVS. In (a), the dataset and thus the queue size is just too small to compensate for the overhead of parallel computation. For all other datasets, increasing the number of threads until the physical maximum of 10 (i.e. between 8 and 16) shortens the runtime as expected. The scaling is not linear, but depending on the dataset in the order of the square root of the number of threads.

Especially in low-degree graphs such as (c) and (d), even combining the coloring and a colored FVS yields a 300% speedup over the old code. Compared to dy-

namic programming in Figure 1, the time spent in the FVS selection thus drops down to approx. 15%, with the absolute time per iteration with 16 threads shortened by more than two seconds. In high-degree graphs such as (e), the coloring itself suffers from more conflicts and takes longer; however precomputing it allows the colored FVS to avoid these conflicts and save a lot of rollback operations, as evident by the speedup even in the single-core case.

5 Conclusion

In this report, we presented an approach to static conflict resolution when sampling FVS (resp. its inverse, a forest on the graph). We used graph coloring to restrict the nodes added per iteration and described the resulting changes to mapMAP's FVS selection algorithm. Benchmarks of the old and new code showed a speedup of up to 3 times on medium to large datasets.

While we implemented this algorithm on the CPU, removing the conflict resolution phase results in less memory writes and stalls and therefore should be even more beneficial for the GPU. An integration of a GPU-implementation of this and other parts of mapMAP is left as future work.

The code for our lock-free FVS implementation will shortly be merged into the `master` branch of our GitHub repository [2].

Acknowledgements

The work of Daniel Thuerck is supported by the 'Excellence Initiative' of the German Federal and State Governments and the Graduate School of Computational Engineering at Technische Universität Darmstadt.

References

[1] D. Thuerck, M. Waechter, S. Widmer, M. von Buelow, P. Seemann, M. E. Pfetsch, and M. Goesele, "A fast, massively parallel solver for large, irregular pairwise markov random fields," in *Eurographics/ ACM SIGGRAPH Symposium on High Performance Graphics*, U. Assarsson and W. Hunt, Eds., The Eurographics Association, June 2016.

[2] "mapMAP Code on Github." [Online]. Available: https://github.com/dthuerck/mapmap_cpu

[3] M. Deveci, E. G. Boman, K. D. Devine, and S. Rajamanickam, "Parallel graph coloring for manycore architectures," in *Parallel and Distributed Processing Symposium, 2016 IEEE International*. IEEE, 2016, pp. 892–901.