High Performance Graphics (2016)
Ulf Assarsson and Warren Hunt (Editors)

A Fast, Massively Parallel Solver for Large, Irregular Pairwise Markov Random Fields

D. Thuerck[1,2], M. Waechter[1], S. Widmer[1,2], M. von Buelow[1], P. Seemann[1], M. E. Pfetsch[1,2] and M. Goesele[1,2]

[1]TU Darmstadt [2]Graduate School of Computational Engineering; TU Darmstadt

Abstract

Given the increasing availability of high-resolution input data, today's computer vision problems tend to grow beyond what has been considered tractable in the past. This is especially true for Markov Random Fields (MRFs), which have expanded beyond millions of variables with thousands of labels. Such MRFs pose new challenges for inference, requiring massively parallel solvers that can cope with large-scale problems and support general, irregular input graphs. We propose a block coordinate descent based solver for large MRFs designed to exploit many-core hardware such as recent GPUs. We identify tree-shaped subgraphs as a block coordinate scheme for irregular topologies and optimize them efficiently using dynamic programming. The resulting solver supports arbitrary MRF topologies efficiently and can handle arbitrary, dense or sparse label sets as well as label cost functions. Together with two additional heuristics for further acceleration, our solver performs favorably even compared to modern specialized solvers in terms of speed and solution quality, especially when solving very large MRFs.

Categories and Subject Descriptors (according to ACM CCS): I.4.m [Image Processing and Computer Vision]: Miscellaneous—Probabilistic Models

1. Introduction

In the last two decades the framework of Markov random fields (MRFs) and global inference proved to be a versatile tool for probabilistic models [Sze11]. Pairwise MRFs in particular have become ubiquitous in computer vision. The MRFs in current computer vision problems such as segmentation, deblurring, inpainting, global structure from motion [COSH13], or as part of benchmarks [SZS*08,KAH*15,SHK*14] present new, tough challenges: millions of nodes, hundreds of labels, arbitrary cost functions, or even new classes such as label costs. A recent example is texturing 3D reconstructions from real-world images [WMG14] (Figure 1 shows an example), where MRFs have irregular topology, more than 10^7 variables, and more than 500 labels. While restricting cost functions or topologies allows using specialized algorithms that can even solve very large MRFs efficiently, the general case—*i.e.* only weak or no assumptions on costs and the topology—strongly encourages research toward *general* massively parallel solvers.

Solving the MAP inference problem on such MRFs with arbitrary structure in a massively parallel way suitable for GPUs is hard due to the lack of regularity and readily available parallelism. Therefore, Chen and Koltun [CK14] recently focused on MRFs with grid topology and proposed a block coordinate descent (BCD) solution. We present the first efficient massively parallel approach and implementation for solving irregular MRFs, that is able to handle even large MRFs with tens of millions of nodes and hundreds of labels per node in less than a minute on a modern GPU.

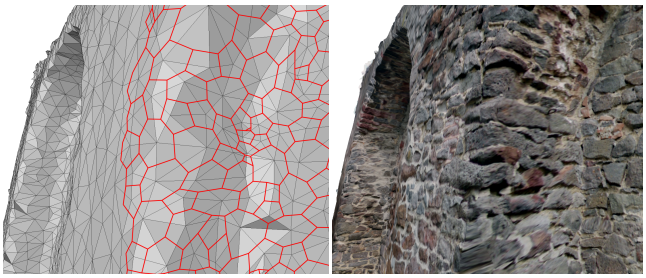

Figure 1: Left: Citywall triangle mesh with a portion of the resulting MRF (edges) overlaid in red. Right: Textured result. This dataset's size and structure stresses the need for an efficient, parallel solver without a restriction to regular grid-based topologies.

One key motivation of our paper is that even though many problems initially have a regular structure (*e.g.*, pixel-based MRFs from computer vision problems), efficient solution strategies often remove this regularity: As Kim et al. [KNKY11] show, grouping variables into clusters in a cost-sensitive manner to restrict the solution space, can be very effective. But only a solver that finds parallelism beyond a regular topology can benefit from such insights.

DOI: 10.2312/hpg.20161203

While some of our solver's underlying concepts have been introduced before, we extend and combine them in a way that is centered on massively parallel computation suitable for modern GPUs. The key contributions of this paper are as follows:

- We introduce the first efficient, massively parallel solver for MRF MAP problems with minimal assumptions on their topology, smoothness costs and label costs that excels for large problems. We only require the topology to be finite and pairwise.
- To this end we present a block coordinate selection scheme for irregular topologies and its massively parallel implementation,
- an approach to integrate label costs in dynamic programming-based optimization methods (such as belief propagation), and
- a description of how the mentioned components and two well-known, adapted heuristics for rapid energy descent can be integrated into an efficient, massively parallel framework.

Our solver implements a general-purpose method that handles cases not efficiently covered by other solvers: Arbitrary (even non-metric) smoothness costs, label costs, arbitrary topology (in the following, we assume a finite, pairwise graph when using this term), and sparse cost tables (which avoids numerical instabilities compared to solvers that handle sparse costs by assigning large costs to infeasible labels). Further, it does not rely on expert knowledge regarding, *e.g.*, problem decompositions (in contrast to Komodakis [Kom10]), yet solves large instances out of the box. For small and medium-sized problems the resulting energies are close to the state of the art, which is often considered to suffice for practical use [MYW05], while for very large datasets (*e.g.* in our datasets with up to $1.5 \cdot 10^7$ nodes, $7 \cdot 10^7$ edges, and >500 labels) we excel in terms of runtime—in several cases by 1–2 orders of magnitude—and memory consumption. As our solver operates in the primal domain, it can be terminated at any time with a valid solution, which is especially useful for very large datasets.

2. Related Work

There is a large body of work on MRF optimization. We refer to Kappes *et al.* [KAH*15] for a comprehensive review and focus on the most important techniques, with a special emphasis on parallel algorithms for large problems. Compared to early approaches such as Iterated Conditional Modes (ICM) [Bes86], today's MRF solvers have drastically improved in terms of solution quality and speed. Currently, there are two main classes: *primal* and *dual* methods.

Primal methods begin with a feasible solution and generate a labeling sequence with decreasing energy until a local minimum is reached. We observe two approaches for calculating the next solution: label space reduction or simplification of the MRF topology.

A popular representative of the former is α-expansion [BVZ01], which reduces multi-label MRFs onto sequences of min-cut/max-flow computations with two labels. Its minima are optimal within one α-expansion move neighborhood. It has been improved, *e.g.*, by guided selection of labels per iteration [BK11] or warmstarting [JB06]. The original graph cut method restricts the binary costs to (semi-)metrics to guarantee decreasing energy. Lempitsky *et al.* [LRRB10] circumvent this by combining different independent expansion moves for a lower energy using quadratic pseudo-boolean cuts (QPBO). Label costs are supported in a modified α-expansion via auxiliary nodes in the flow network [DGVB12].

The second class of primal methods builds on the idea of keeping a subset of the variables fixed and optimizing only the remainder of the MRF while taking the fixed variables into account. The remaining topology is selected to be tractable for exact optimization. This is known as *block coordinate descent* (BCD). ICM is the simplest representative of this class, with its generalization Block-ICM [KMMH06] being a template for other algorithms. Chen and Koltun [CK14] use alternating scanlines on grids and optimize them using dynamic programming, whereas Andres *et al.* [AKB*12] use a bounded depth-first search with candidate elimination on larger, connected windows of the graph. Our method falls into this class of purely primal methods.

For specific cost function classes such as the Potts model special methods have been presented [Vek15]. When applicable, these methods can speed up the solution process by orders of magnitude but since they are highly problem-dependent, they are out of our work's scope. Additionally, some techniques used in two-view stereo could in principle be applied to the general MRF labeling problem; for an overview we refer to Scharstein *et al.* [SHK*14].

Dual methods maximize a lower bound on the optimal energy and use rounding and other heuristics to generate feasible solutions. Solvers are often based on the linear programming relaxation over the local polytope, *e.g.* FastPD [KT07], or the Lagrangian relaxation [Kol06]. Apart from this, especially the Lagrangian dual can be regarded as a framework for decomposing the MRF into smaller parts solved by dual decomposition [KPT07]. For some problems these relaxations are quite loose, *i.e.*, there exists a so called integrality gap. Additionally, depending on the costs even the relaxation solvers are sometimes unable to close the duality gap, providing inferior feasible solutions. Therefore, a common approach is tightening these relaxations by cutting-planes such as cycle-repairing [KP08], to an extent where MRF solvers more and more resemble conventional branch-and-cut solvers.

If dual methods operate both in the primal and dual domain (*e.g.*, for generating feasible labelings), the memory requirements for the additional dual variables can have a negative impact on performance. Nevertheless, they are useful to obtain lower energy bounds to estimate the quality or prove the optimality of primal solutions. Most dual methods support arbitrary energy functions.

2.1. Parallel Algorithms

Efforts to parallelize MRF optimization have so far concentrated on parallelizing state-of-the-art sequential algorithms. The majority of implementations with competitive performance is however limited to specific MRF topologies. Due to its frequent use in stereo and image processing, there are several efficient algorithms tailored to pixel grids: On the graph cut side, Delong *et al.* [DB08] replaced the underlying max-flow algorithm by a push-relabel-based method that splits the graph into several regions and processes each in a separate thread. The grid topology can be exploited to make the max-flow algorithm more cache-efficient [JSH12]. Shekhovtsov and Hlaváč [SH13] present a hybrid max-flow algorithm that allows arbitrary graph partitions, but do not investigate how to best

split up irregular MRF graphs. Fishbain *et al.* [FHM13] note, however, that a great deal of the reported performance increase stems from serializing heuristics such as global relabeling that forms a bottleneck for large graphs. This problem can especially be seen in GPU solvers [VN08, STT10], unless the optimality criterion on the flow is relaxed. In addition to leveraging fine-grained parallelism in max-flow computations, Veksler [Vek15] uses parallelism on an algorithmic level, decomposing α-expansion into a hierarchy of independent flow problems, solving each on a different CPU core.

Chen and Koltun [CK14] propose a BCD scheme for regular grids. In each round, they use a subset of the horizontal or the vertical lines that yield 1D MRFs, and optimize them with vectorized dynamic programming. The resulting solver is often an order of magnitude faster than general-purpose solvers. Their results' quality is however 5–10 % worse than previous work, their approach is limited to submodular functions, and the scanline concept cannot be directly transferred to non-grid topologies.

Nonetheless, given dual methods' memory requirements and flow algorithms' limited scalability, BCD seems suitable for very large MRFs. Chen and Koltun [CK14] and Fix *et al.* [FCBZ12] note that generalizing BCD to arbitrarily shaped MRFs could be promising. Efficient selection of subgraphs is, however, still an open question. In this paper we give a possible answer to this problem in its most general form: we pay special regard to working on irregular input even on hardware built for regular inputs, such as GPUs.

3. Goal of this Paper

A pairwise MRF is an undirected graph $\mathcal{M} = (\mathcal{P}, \mathcal{N})$ where $\mathcal{P} = \{0, \dots, n\}$ are the IDs of nodes corresponding to random variables. In the following we use "node" and "variable" interchangeably. The neighborhood relation $\mathcal{N} \subseteq \{\{i, j\} : i, j \in \mathcal{P}, i \neq j\}$ models pairwise dependencies between variables. With a set of labels $\mathcal{L} \subset \mathbb{N}$, the energy function for a labeling (or assignment) $f : \mathcal{P} \to \mathcal{L}$ is

$$E(f) = \sum_{i \in \mathcal{P}} D_i(f(i)) + \sum_{\{i,j\} \in \mathcal{N}} V_{\{i,j\}}(f(i), f(j)) + \sum_{\ell \in \mathcal{L}} \delta_\ell(f) M_\ell. \quad (1)$$

For each $i \in \mathcal{P}$, $D_i : \mathcal{L} \to \mathbb{R}_+$ are the unary (data) costs for assigning a specific label to a given node. Moreover, $V_{\{i,j\}} : \mathcal{L}^2 \to \mathbb{R}_+$ for $\{i, j\} \in \mathcal{N}$ are the binary (smoothness) costs defined on pairs of neighboring nodes. Finally, $M_\ell \in \mathbb{R}_+$ is the fixed cost of label ℓ, which occurs as soon as ℓ is assigned to at least one node in the graph, i.e., $\delta_\ell(f) = 1$ iff there exists an $i \in \mathcal{P}$ with $f(i) = \ell$. Further, we optionally allow a different set of feasible labels $\mathcal{L}_i$ for each node $i \in \mathcal{P}$. In this case we define $\mathcal{L} = \bigcup_{i \in \mathcal{P}} \mathcal{L}_i$.

Our goal is to minimize $E(f)$ in Equation (1) on arbitrary topologies and with arbitrary costs D_i, $V_{\{i,j\}}$, and M_ℓ, which is $\mathcal{NP}$-hard [Shi94, BVZ01]. This problem is generally called the *Maximum A-Posteriori probability* problem (*MAP* in short). On special topologies such as trees, it is solvable in polynomial time via belief propagation [FH06] or dynamic programming [Vek05, Sze11]—at least if label costs are all zero. Otherwise it is $\mathcal{NP}$-hard even on trees, as shown by Delong *et al.* [DGVB12] by reduction of the *uncapacitated facility location problem*. In line with recent hardware development, we focus on an algorithm—and explain its implementation in detail—which efficiently leverages modern, massively parallel hardware to tackle this general class of problems.

4. Algorithm

BCD-based algorithms iteratively select a subset of nodes, keep the labels of the remaining nodes fixed, and optimize the selected subset. Hence, the key ingredient for optimization quality and performance is the choice of coordinates for descent in each iteration. When selecting the coordinates, one usually forces the resulting subproblem to have a specific structure that makes solving it easier. In that sense, the coordinate selection scheme (BCD scheme) sets the stage for leveraging parallelism in solving the subproblems. As we aim to keep our solver general-purpose, its BCD-scheme must be designed with the requirements of massively parallel hardware in mind. In the past, massively parallel MRF solvers were restricted to regular structures such as grids [CK14] with BCD schemes that are not trivially applicable to irregular structures. This section proposes a BCD scheme coping with the latter and describes how the resulting subproblems can be solved by dynamic programming—and approximated when including label costs.

Applying a BCD scheme partitions the variable set $\mathcal{P}$ into two disjoint subsets: The *selected coordinates* $\mathcal{C}$ (from here on called *coordinates*) and the fixed variables $\mathcal{F} = \mathcal{P} \setminus \mathcal{C}$. While the subproblem given by $\mathcal{C}$ is solved, fixed variables do not change their label, but keep the label assigned to them in the previous BCD iteration. The partition into $\mathcal{C}$ and $\mathcal{F}$ gives rise to three types of edges:

- edges $\mathcal{N}[\mathcal{C}]$ between two coordinates, which we call *links*,
- edges $\mathcal{N}[\mathcal{C}, \mathcal{F}]$ between coordinates and fixed variables, which we call *dependencies*, and
- edges $\mathcal{N}_\mathcal{F} = \mathcal{N}[\mathcal{F}]$ between two fixed variables.

A subproblem is specified by a subgraph $(\mathcal{C}, \mathcal{N}_\mathcal{C})$ with $\mathcal{N}_\mathcal{C} \subseteq \mathcal{N}[\mathcal{C}]$ and $\mathcal{N}_\Delta \subseteq \mathcal{N}[\mathcal{C}, \mathcal{F}]$. Given an assignment f with subset $f_\mathcal{F}$ for the fixed variables, subtracting terms from Equation (1) that only include fixed variables yields the objective:

$$E_{\text{sub}}(g) = \sum_{i \in \mathcal{C}} D_i(g(i)) + \sum_{\{i,j\} \in \mathcal{N}_\mathcal{C}} V_{\{i,j\}}(g(i), g(j)) + \sum_{\{i,j\} \in \mathcal{N}_\Delta} V_{\{i,j\}}(g(i), f_\mathcal{F}(j)) + \sum_{\ell \in \mathcal{L}} \delta_\ell(g)(1 - \delta_\ell(f_\mathcal{F})) M_\ell. \quad (2)$$

An assignment $g : \mathcal{C} \to \mathcal{L}$ that minimizes Equation (2) yields a new assignment f' via $f'(i) = g(i)$ if $i \in \mathcal{C}$ and $f'(i) = f(i)$ otherwise, for which $E(f') \leq E(f)$ holds—if the subproblem is *compatible* with the original MRF. Compatibility means that $\mathcal{N}_\mathcal{C} = \mathcal{N}[\mathcal{C}]$ and $\mathcal{N}_\Delta = \mathcal{N}[\mathcal{C}, \mathcal{F}]$. Otherwise, the second and third term of Equation (2) do not sum over all necessary binary costs. Therefore, $\mathcal{N} = \mathcal{N}_\mathcal{C} \cup \mathcal{N}_\Delta \cup \mathcal{N}_\mathcal{F}$ must hold, which we call *edge partition requirement*. Our solver respects it, whereas Veksler's spanning trees [Vek05] do not and therefore have no energy decrease guarantee. The supplemental material elaborates on this in more detail (Section B.3.1) and proves the energy monotonicity (Section B.3).

4.1. Coordinate Selection

Before discussing our coordinate selection scheme in detail, let us review some requirements: First, it is desirable to add as many variables to $\mathcal{C}$ as possible to enlarge the resulting subproblem's search space, thereby allowing for more substantial energy decrease. Second, the subproblem should remain tractable. Both goals are contradictory. With respect to generality, the selection scheme should

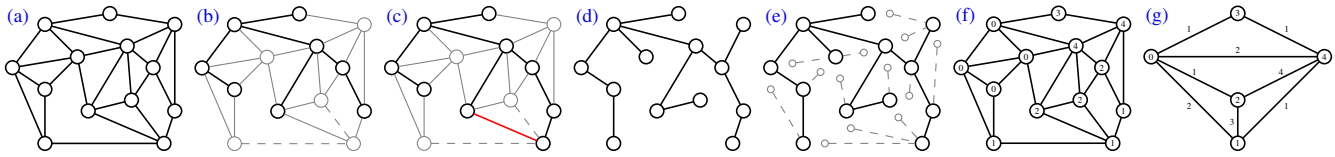

Figure 2: (a) MRF, (b) maximal acyclic coordinate set (black: coordinates and links; solid gray: fixed variables and dependencies; dashed gray: edges between fixed variables), (c) the same as (b) but with one node too many inserted which creates a conflict (red), (d) spanning tree, (e) corresponding modified spanning tree with small, gray node copies and dashed dependencies for all removed links, (f) original MRF with an initial solution f, and (g) corresponding region graph for a piecewise constant prior.

also not assume anything about the MRF topology. In the following, whenever we talk of *coordinate sets*, this means the set $\mathcal{C}$ as well as the topology of the subproblem resulting from adding (a subset of) links and dependencies.

We trade both goals off as follows: To fulfill tractability we choose coordinates such that the graph $(\mathcal{C},\mathcal{N}_\mathcal{C})$ is a *tree*, *i.e.*, connected and acyclic. Veksler [Vek05] and Szeliski [Sze11] proved that trees can be optimized in polynomial time with dynamic programming. We thus restrict ourselves to *tree coordinate sets*. To fulfill maximality we select compatible trees that are maximal in the sense of set-inclusion, *i.e.*, no variable can be added to $\mathcal{C}$ without introducing cycles. Figure 2b shows such a maximal coordinate set. Any two coordinates not sharing a link are separated by at least one fixed variable. In Figure 2c we added one more variable to $\mathcal{C}$, which creates a conflict marked in red: The edge partition requirement forces the red edge to be in $\mathcal{N}_\mathcal{C}$, but this violates acylicity. Thus, the tree in Figure 2b is already maximal. This "separated by at least one fixed variable"-insight will be used in Section 6.1 where we describe our parallel tree sampling implementation.

Next, we show how such a subproblem—a tree MRF with dependencies—can be solved to optimality if no label costs M_ℓ are used. We deal with label costs in a separate section afterwards.

4.2. Solving Tree MRFs with Dependencies via Dynamic Programming

Our solution of Equation (1) follows Veksler [Vek05]. We assume rooted, directed trees as input. Let i be a tree node, p_i its parent, $\mathfrak{C}_i$ its children, and $\mathfrak{D}_i$ the nodes connected to it via dependencies. For a leaf i, the energy of its optimal labeling $f(i)$ can be tabulated for all feasible parent labels $\ell_p \in \mathcal{L}$:

$$E(f(i);\ell_p) = \min_{\ell\in\mathcal{L}} \left(D_i(\ell) + V_{\{p_i,i\}}(\ell_p,\ell) + \sum_{d\in\mathfrak{D}_i} V_{\{i,d\}}(\ell,f(d)) \right). \tag{3}$$

We now proceed up in the tree with the following recurrence. Node i is an inner node and $f(i)$ is the labeling of the subtree rooted at i.

$$E(f(i);\ell_p) = \min_{\ell\in\mathcal{L}} \Big(D_i(\ell) + V_{\{p_i,i\}}(\ell_p,\ell) + \sum_{d\in\mathfrak{D}_i} V_{\{i,d\}}(\ell,f(d)) + \sum_{c\in\mathfrak{C}_i} E(f(c);\ell) \Big). \tag{4}$$

Hence, an inner node can be processed and tabulated for all feasible ℓ_p if all its children's $E(f(c);\ell)$ are tabulated. The root is treated like an inner node but without the parent term $V_{\{p_i,i\}}$. After computing its minimal energy, we obtain the tree MRF's optimal labeling by traversing the tree top-down with lookups in the E table. This dynamic programming instantiation has no cost function restrictions. Its complexity is $\mathcal{O}(|\mathcal{L}|^2 D_{\max})$ ($D_{\max}$ is the maximal degree of nodes in $\mathcal{M}$). For some cost function classes one could further reduce the complexity to $\mathcal{O}(|\mathcal{L}|\,D_{\max})$ for tree MRFs using fast distance transformations as in Felzenswalb and Huttenlocher [FH06].

Algorithm 1 Local search scheme (one sweep) for tree MRFs with label costs with initial set $\mathcal{L}'$.

```
e ← ∞
△ ← L'
for ℓ ∈ L do
    △ ← △ ∪ {l}
    f_△ ← labeling by DP restricted to △
    reduce labels not used in f_△ from △
    if E(f_△) < e then
        e ← E(f_△)
    else
        △ ← △ \ {ℓ}
    end if
end for
return △
```

4.3. Label Costs

Equations (3) and (4) are based on the locality of unary and binary costs for subtrees. Label costs, on the contrary, are global decisions in the sense that using a label in one node during optimization influences all other nodes. As Delong *et al.* [DGVB12] show, introducing label costs makes even tree-shaped MRFs $\mathcal{NP}$-hard.

Including label costs into the dynamic programming process in a way that loses the optimality guarantee poses certain problems: When considering the leaves in Equation (3), label costs can dominate the unary and binary costs, thus forcing the solution to the least-cost label, ultimately ignoring the localized costs, leading to low-quality solutions. Instead, we strive to avoid these localized decisions by borrowing an idea from α-expansion: Restricting the set of feasible labels in subproblems. The rationale behind this can be explained by the following idea: Let f^* be the optimal solution of the subproblem in Equation (2). Then there is a corresponding optimal label set $\triangle^*_{f^*}$. If we had this set at hand, setting $\mathcal{L} := \triangle^*_{f^*}$ and using dynamic programming as usual—without label costs—delivers the optimal solution including label costs. Intuitively, restricting the label set means the costs have already been paid for all labels of the solution, hence using a label comes for 'free'.

To approximate $\triangle^*_{f^*}$ we propose a greedy local search algorithm (see Algorithm 1 for pseudocode): We start with an initial label set

$\triangle$, *e.g.* of size 1, and iterate over $\mathcal{L}$. Adding each label, we determine the optimal solution of the new subproblem. If the resulting solution has lower costs according to Equation (1) than the current solution, the label is permanently added to $\triangle$. These *sweeps* over $\mathcal{L}$ are repeated until no further cost reduction is possible.

5. Heuristics

In addition to the presented, generic BCD scheme we include two heuristics into our solver to improve solution quality and speed. Both yield tree subproblems that are solved as shown above.

5.1. Spanning Trees

Our BCD scheme respects the edge partition requirement to guarantee monotonous energy decrease. Given a maximal acyclic coordinate set, every extension of it leads to cycles in the subproblem and we loose tractability—if we respect the partition requirement. To allow for steeper energy descent we propose to drop the partition requirement by removing some links from the subproblem while adding the maximal amount of coordinates, *i.e.* $\mathcal{C} = \mathcal{P}$.

Removing only the minimum number of links necessary for acyclicity yields a spanning tree on the original MRF as in Veksler's approach [Vek05]. Her approach is, however, non-iterative and does not include information from a previous assignment—the previous BCD iteration in our case. Thus, to iteratively improve a prior labeling f, we propose to replace removed links by dependencies on auxiliary variables. For each of the removed links $\{i, j\}$ with $i < j$, we modify the subproblem as follows:

1. Add auxiliary variable i' (called copy of i) to X.
2. Set the assignment $f(i')$ to $f(i)$.
3. Add a dependency $\{i', j\}$ to $\mathcal{N}_\Delta$.

Figures 2d and e outline the result. The copies force the coordinates to respect the prior labeling f via dependencies. Unless f was a local optimum of $\mathcal{M}$, the optimization will produce label disagreements between coordinates and their copies. Thus, the subproblem's objective is incompatible with the original MRF's objective, leaving us with no formal guarantees. This implies that its effectiveness can only be determined by experimental evaluation.

5.2. Region Graphs

The basic BCD scheme and spanning trees both operate on single-node granularity, *i.e.*, they calculate dynamic programming tables per node. Many applications such as stereo or texturing favor, however, piecewise constant or smooth solutions where large MRF *regions* have equal or mostly equal labels. In this case we can speed up the optimization by grouping nodes, thereby changing the MRF graph and adapting it to the inherent regularity of the problem.

This approach is often referred to as *variable grouping* (*e.g.*, Kim *et al.* [KNKY11]) and is a standard approach in computer vision optimization. It is particularly often used in multigrid frameworks [FH06, BG12] that employ a whole hierarchy of variable groups, propagating information about possible solutions through the layers. While the concept of grouping variables and optimizing them as if they were one variable is intuitively clear, the method for selecting the groups-to-be is of vital importance. Mostly, variables are grouped upfront based on topology [FH06], cost functions [KNKY11], or based on a given assignment.

In our implementation, we opt for the latter: To facilitate the formation of large, homogeneous label clusters, we group adjacent variables that have the same or a similar label (depending on the prior). In the resulting region graph we represent such clusters with a single node and add edges between these region graph nodes such that the original topology is reflected: If two adjacent variables end up in different clusters, we connect their region graph nodes by an edge. Unary and binary costs are added up accordingly.

For optimizing the region graph we then execute either a BCD or a spanning tree step and propagate the resulting labels to the corresponding variables in the full MRF. Iterating this successively fuses smaller regions, yielding large homogeneous regions.

6. Parallel Implementation

Now that we have all four algorithmic building blocks (coordinate set selection, dynamic programming on trees, spanning trees, and region graphs) at hand, we give a short overview of how these are combined (see Figure 3): To quickly reach the neighborhood of high-quality solutions, we initialize the MRF's labeling with a single iteration of our spanning tree heuristic without dependencies (since there is no prior labeling). With the result we then optimize region graphs until they fail to decrease the energy. Since region graphs cannot be applied for priors that do not prefer homogeneous regions, our implementation detects this case and automatically skips region graph optimization if necessary. The solution now exhibits large homogeneous regions, and further energy decrease can only be achieved by exchanging nodes between regions. Therefore, we alternate between five iterations of spanning trees with dependencies to disturb regions and one region graph iteration to combine new regions. When these methods do not yield further improvements, inclusion-maximal trees are used to refine the solution. In terms of the solution space, using spanning trees can be thought of a step that could potentially worsen the solution, but escapes an otherwise enclosed neighborhood; whereas the inclusion-maximal trees represent classical coordinate descent.

All four building blocks can be efficiently implemented on many-core architectures. In the following we describe our implementation, not focusing on a particular architecture but assuming a generic, SIMD-structured machine with the following features: The available processing elements (PE) can be divided into groups. PEs within a group can share local, fast memory. All groups share the main memory and can perform simple atomic operations on it. Many popular platforms such as GPUs, FPGAs, Xeon Phi or multi-core CPUs satisfy these requirements. We implemented our algorithm for CPUs using Intel's TBB runtime and GPUs using CUDA.

As all following algorithms operate purely on graphs—the MRF's topology or subsets thereof—we mostly use the terms graph, tree, node and edge instead of optimization-specific terms. To represent the MRF's topology, we use an adjacency list structure packed as structure-of-arrays in device memory.

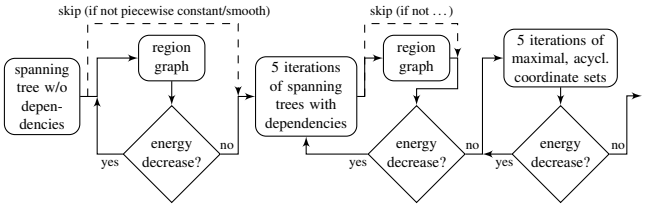

Figure 3: Overview over our algorithm's control flow.

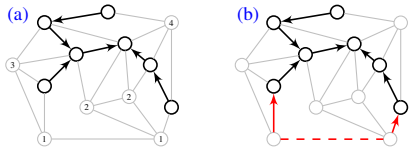

Figure 4: Sampling maximal tree coordinate sets: (a) Markers on a non-maximal coordinate set. (b) Concurrently adding the two red nodes causes an edge partition requirement violation (dashed red).

6.1. Sampling Maximal Acyclic Coordinate Sets in Parallel

We now describe our parallel coordinate set selection algorithm that runs in $\mathcal{O}(|\mathcal{P}|)$ steps and later show how spanning tree selection is derived from this. We start with the full MRF graph as input and then select nodes as coordinates such that the graph formed by the coordinates and links is a maximal, rooted, directed tree that obeys the edge partition requirement. We start with an empty coordinate set $\mathcal{C}_0$ and iteratively add nodes that violate neither acyclicity nor the edge partition requirement (recall Section 4.1 and Figures 2b and 2c about both requirements). When adding a node to $\mathcal{C}$ both requirements can be formalized as a necessary condition:

Invariant 1 Let $\mathcal{C}$ be a (not necessarily maximal) acyclic coordinate set. A node i is a candidate for inclusion into the coordinate set if and only if it is a neighbor of at most one node in the current tree.

If Invariant 1 does not hold for a node (such as the bottom right node in Figure 2c), it is adjacent to at least two nodes in the current tree and its inclusion would violate either acyclicity or the edge partition requirement. For serial execution, the invariant is necessary *and* sufficient. We add node after node until there is none left satisfying the invariant. Note that the resulting tree is not unique and depends on the node adding order. We make use of this to avoid getting trapped in local minima by sampling coordinate sets uniformly at random, thereby optimizing on varying subproblems.

To obey Invariant 1 during serial execution we can simply use counters for each non-coordinate counting the number of adjacent coordinates. However, for parallel execution the invariant is not sufficient and using the counters fails when nodes are added simultaneously, as illustrated in Figure 4: In Figure 4a, the black nodes and links form a non-maximal coordinate set. All nodes not in the tree store a counter for adjacent tree nodes. All nodes having a counter of zero or one are candidates for inclusion into the tree. However, if both nodes at the bottom with a counter of one are added at the same time, the dashed red edge in Figure 4b connecting both nodes leads to a cycle in the resulting subproblem. Thus, we need to add a second concept on top of the counters. The first idea coming to mind is *locking* certain regions around nodes. Since we explicitly support irregular inputs, locking a minimal number of nodes in each step is not straightforward. Therefore, we propose an alternative based on trial and error. We describe this algorithm in textual form, referencing the pseudocode in Algorithm 2 by line numbers for clarity.

Algorithm 2 Parallel algorithm for sampling maximal, acyclic coordinate sets (code for one SIMD unit identified by its `id`, working on queues $w_{\texttt{in}}, w_{\texttt{out}}$).

```
1:  i ← w_in(id)
2:  Phase I: try growing a new branch
3:  if i successfully locked then
4:      select random adjacency table entry e_r where r ∈ [0, d_f(i)) ∩ ℕ
5:      j ← neighbor of i via e_r
6:      if j successfully locked and j not in tree and m(j) < 2 then
7:          p(j) ← i
8:          put j into w_out if d_f(j) > 1
9:          release j's lock
10:     end if
11:     if d_f(i) > 1 then
12:         swap e_r with adjacency table entry at d_f(i) − 1
13:         d_f(i) ← d_f(i) − 1
14:         put i into w_out
15:     end if
16:     release i's lock
17: end if
18: Phase II: update markers and detect collisions
19: for n ∈ N(j) do
20:     if n in tree then
21:         record conflict (j, n)
22:     end if
23:     m(n) ← m(n) + 1 (atomic)
24: end for
25: Phase III: resolve conflicts
26: (c_1, c_2) ← conflict for id
27: c* ← random{c_1, c_2}
28: if c* in tree then
29:     remove c* from tree (invalidate parent)
30: end if
```

We start with an empty coordinate set and extend it in rounds, growing a tree step by step. To organize the growing process, we keep track of two work queues, w_{in} and w_{out}, which switch their roles in every iteration. The queue w_{in} contains the nodes that are already in the tree and have at least one adjacent non-coordinate node. We call them *gateways*. We parallelize over this queue's content, each PE selecting one node from w_{in} in each round.

Having selected a node, each PE locks its node i using an atomic exchange instruction. If locking was successful, an adjacent inclusion candidate j (*i.e.*, a node with counter < 2) is chosen at random (Algorithm 2 line 4). If the PE then also obtains j's lock, j and the edge $\{i, j\}$ are included into the tree. This constitutes a round's first of three phases: *growing*. At this point the case illustrated in Figure 4b (a violation of Invariant 1) may have happened. In the second phase—*checking*—the markers of nodes adjacent to the newly added nodes are updated (line 23). If we encounter an adjacent node that is in the tree as well, we found the case from Figure 4b and record that node pair as a conflict (line 21). In the final phase—*resolving*—each PE selects one of the reported conflicts and resolves it by removing one node of the conflicted pair from the tree (line 29). Which of the two gets removed is chosen randomly to rule out selection bias. Markers of adjacent nodes do not need to be updated: Nodes once removed from the tree already

have two neighbors in the tree and will keep violating Invariant 1. After all three phases, nodes with neighbors that could be included into the tree, are pushed to w_{out} and both queues change their roles.

One algorithm step can cause major branch divergence and hurt performance especially on SIMT systems such as GPUs: selecting an adjacent node with marker <2. A naïve implementation requires a linear search through each gateway's adjacency list, which causes uneven workload on irregular graphs. To avoid this, we propose modifying the adjacency lists in the growing phase (lines 11–15): We keep all edges to tree inclusion candidates at the list's front and set the node's degree d to the number of inclusion candidates. To achieve this, after growing along edge (i, j) we modify i's adjacency list by swapping j with the element at position $d-1$ and decrease d by one. By packing edges to inclusion candidates to the adjacency list's front we can select a random inclusion candidate in constant time, and each edge between a gateway and an inclusion candidate is considered at most once in the whole algorithm. During execution of the algorithm, invalidities may arise from modifying only one node's adjacency table per PE, but not necessarily its adjacent node(s)' tables. In this case adjacency table entries representing edges to invalid nodes are dangling, but we detect this in a later iteration and remove them on demand. This scheme avoids costly searching for nodes in their respective adjacency tables.

Spanning Trees can be grown similar to coordinate sets, but since they clearly violate Invariant 1 anyway, we drop the checking and resolving phase. Only random gateway selection and growing remains. Since for spanning trees $\mathcal{C} = \mathcal{P}$ holds, we can count nodes not yet in the tree and terminate early when that counter reaches 0 regardless of the queue's content. After termination, we just ignore all edges that were never traversed.

6.2. Creating Region Graphs in Parallel

Our region graph heuristic groups nodes in a cost-sensitive manner, hence conditions for grouping nodes into supernodes depend on the input energy. For smoothness-enforcing priors grouping adjacent nodes that currently have the same label is the natural choice. We now describe a massively parallel procedure for this case. Others can be constructed accordingly.

The key step in region graph construction is assigning a common ID to all nodes that are to be fused. Initially, we assign a unique ID to each node. We then proceed iteratively: In each iteration, each MRF edge is assigned to one PE which checks whether the incident nodes should be fused according to the fusion criterion. If so, their current IDs are set to the minimum of both nodes' previous ID. Iterating this until no more changes occur leads to all nodes in a future supernode having the same ID. Next we collect all these IDs and map them to a contiguous sequence of supernode IDs: All PEs write their current ID into a list and we sort this list. In this sorted list L_1 we check each position i with a separate PE, create a new list L_2 of the same size, and set $L_2(i)$ to 1 if $L_1(i) \neq L_1(i+1)$ and 0 otherwise. Using the parallel primitive of exclusive scan on L_2, we finally obtain the contiguous list of supernode IDs. Given each node's ID from after the iteration stopped (which we save in device memory while processing L_1 and L_2), we use the resulting mapping from L_1 to L_2 to their respective new supernode IDs.

Algorithm 3 Organization for bottom-up tree dynamic programming for a group of PEs.

```
1:  i ← id for current node
2:  p ← id of i's parent
3:  optimize(i, p):
4:  L_p ← p's set of feasible labels
5:  if |L_p| > |L_i| then
6:      get spill-over memory from stash for DP(i, ·)
7:  end if
8:  for ℓ_p ∈ L_p do
9:      run Algorithm 4 for (i, ℓ_p)
10: end for
11: d(p) ← d(p) − 1
12: if d(p) == 0 then
13:     i ← p
14:     p ← id of i's parent
15:     goto optimize(i, p)
16: end if
```

After obtaining supernode IDs for every node we can construct the graph of supernodes and sum up the supernodes' unary costs. To stay compatible with the original energy function, we add the binary costs of edges inside a supernode to the supernode's unary costs. Also, we sum up binary costs of multiple edges between supernodes and assign them to the corresponding superedge.

6.3. Solving Tree MRFs with Dependencies in Parallel

After sampling a coordinate set, we use dynamic programming to solve the resulting subproblem. Our implementation follows relatively straightforward from Equations (3) and (4). In general we follow Veksler [Vek05] and restrict this section to presenting the technical details of GPU parallelization.

As Equations (3) and (4) suggest, we can compute all $E(f(i); \ell_p)$ in parallel over the ℓ_ps and handle different branches of a subtree in parallel up to the subtree root. We do this as follows (see Algorithm 3 for pseudocode): On the leaf level each node i is optimized by one PE group. Within a group, each PE performs the optimization of one feasible parent label ℓ_{p_i} (Algorithm 3 line 9). For each non-leaf we maintain a counter of unprocessed children which a PE group decrements atomically when it finishes computation on a child (l. 11). The PE group setting this counter to 0 adopts the node for optimization (ll. 12ff.). This communication-less scheme offers enough independent tasks as long as the tree width and $|\mathcal{L}|$ are large enough. If $|\mathcal{L}|$ is particularly large, we assign multiple labels to one PE. By sharing unary cost tables and feasible label sets among a group's PEs we benefit from high-bandwidth local memory.

After filling the dynamic programming table DP and saving the indices of minima in a separate index table, we execute a top-down traversal on the tree to find the optimal solution's corresponding labeling. Algorithm 4 implements the processing of a leaf or inner node in the tree—according to both Equations (3) and (4)—in a straightforward manner. Again, the tree's width bounds the number of independent task. In most applications, the first stage of filling the DP and index tables offers enough work to saturate the GPU. In contrast, the second stage of climbing down the tree and assigning labels according to the index tables suffers from its inferior ratio of overhead to computation.

Algorithm 4 Parallel dynamic programming for a node i and table DP(i, ℓ), $i \in \mathcal{P}, \ell \in \mathcal{L}_i$; PE with parent's label ℓ_p.

$e_{\min} \leftarrow \infty$
for $\ell \in \mathcal{L}_i$ **do**
  $e \leftarrow D_i(\ell)$
  $e \leftarrow e + V_{\{p,i\}}(\ell, \ell_p)$
  for $c \in \mathfrak{C}_i$ **do**
    $e \leftarrow e + \mathrm{DP}(c, \ell)$
  end for
  for $d \in \mathfrak{D}_i$ **do**
    $e \leftarrow e + V_{\{i,d\}}(\ell_d, \ell)$
  end for
  $e_{\min} \leftarrow \min\{e, e_{\min}\}$
end for
$\mathrm{DP}(i, \ell_p) \leftarrow e_{\min}$

The proposed approach is seemingly easy and work-balanced if all (or most) nodes share the same label set. If they do not, per-node label sets introduce a need for memory management, as DP table sizes vary between nodes. To avoid global synchronization when using the GPU's heap, we reserve a contiguous chunk of device memory as *spill-over* for local DP tables. Additionally, we allow each parent to overwrite its children's DP cost tables. Whenever a parent's DP table needs more space than the children's tables combined, we provide a chunk from this spill-over, which in turn is organized as a ring-buffer. A small spill-over area usually suffices and we can detect overwrites; in this case we repeat the computation with a larger amount of spill-over for an exact solution.

Finally, in order to avoid bottlenecks near tree levels with low tree width, we increase the number of independent tasks by sampling *forests* instead of trees in the algorithm of Section 6.1. To ensure that the forest still maintains acyclicity and the edge partition requirement, we start with a random *set of roots* that obeys the edge partition requirement, store their IDs in the initial queue w_{in}, and continue the algorithm of Section 6.1 as usual.

7. Experimental Evaluation

We evaluate a GPU (using CUDA) and a CPU implementation (using Intel's TBB and manual SSE vectorizations) of our algorithm on a dual Xeon E5-2650 (256 GB RAM) and a GeForce Titan X (12 GB VRAM, 3072 CUDA cores). We compare against some of the computer vision community's most popular solvers: GCO [BVZ01], FASTPD [KT07], Belief Propagation (BP) and TRW-S [Kol06], Chen and Koltun's BCD [CK14] (CK-BCD), and α-expansion with two different parallel max-flow algorithms: DGCO [SH13] and GRIDGCO [JSH12]. We omit parallel BP and TRW-S versions since Kolmogorov [Kol06] noted they generally perform worse than their sequential counterparts. For all competing solvers we wrapped the respective authors' downloadable code.

Since we propose a solver applicable to the vast majority of large datasets, we supply costs in their most general form: as sparse tables in memory. For a fair comparison we do this for all solvers. Using costs modeled by a function limits a solver's applicability (*e.g.*, if costs were learnt). In this paper we concentrate on general solvers, but note that Kappes *et al.* [KAH*15] observed that specialized solvers can be several orders of magnitude faster.

We evaluate on a variety of datasets. Not all solvers are applicable to all datasets due to restrictions on cost type (*e.g.*, (semi-)metrics for GCO and FASTPD), restrictions on topology type (grid for GridCut and CK-BCD), or memory consumption beyond our test systems' capabilities. We omit solvers from the plots that fail to find a feasible solution within reasonable time. Since we zoomed in to focus on important plot parts, some solvers' results may not be visible, which we mark with '$*$' in the graph legends in Figure 5.

Quantitative performance evaluation requires datasets with different properties. Since we especially focus on large-scale optimization, we prefer large datasets. To quantify dataset size we use the average number of feasible labels per node times the number of nodes. We use some of the largest benchmark [KAH*15, SZS*08] datasets: Teddy, Tsukuba, Venus, Brain_9mm, Brain_5mm, and Brain_3mm. Since they all have a grid topology and are two orders of magnitude smaller than some datasets arising from real-world applications, we add further datasets, which we describe shortly with respect to their challenges in the following. The supplemental material gives key figures about their size etc. (Table A.1) and descriptions of the applications they arise from (Section A.1).

Our *sparse plane sweep* datasets have grid topology, sparse unary costs (*i.e.*, not all labels are valid for all nodes) and the two larger instances are much larger than all of Kappes' datasets that we tested. Our *mesh segmentation* datasets have irregular topology, their sizes range from small ($4 \cdot 10^3$) to medium ($4 \cdot 10^6$), they have small label sets which is challenging on the GPU, and one of them (Dragon with label costs) uses label costs. Our *texturing* datasets have irregular topology (see Figure 1), their sizes exceed Kappes' datasets by far ($3.5 \cdot 10^7$ to $7 \cdot 10^8$), their label set is large and unary costs are sparse. Our *graph coloring* datasets have irregular topology, are also much larger than Kappes' datasets (up to $2.4 \cdot 10^8$), have anti-metric binary costs, and one instance employs label costs. Note that the combination of anti-metric binary costs with label costs is not supported by any of the competing solvers.

We now discuss our GPU and CPU results with a focus on the GPU. All plots in Figure 5 have logarithmic time on the x-axis and relative energy offsets to the best solver on the y-axis. Due to space considerations we only show a subset of all datasets. The supplemental material contains larger plots for all datasets (Figures A.1, A.2, and A.3), absolute final energies (Table A.1) to avoid bias in the plots from adding large constant offsets as well as a more in-detail evaluation for each class of datasets (Section A.2).

On small datasets such as Brain_5mm (Figure 5b) or Dolphin (5e) our solver is slower than the best competitors since the low number of nodes or labels does not offer enough parallelism to fully utilize the GPU's processing groups. Also, for many datasets (Venus (5a), Brain_3mm (5c), Planesweep_1280_1022_96 (5d), Dolphin (5e), Dragon (5f), and Citywall-20 (5g)) our solver's final energy is higher than the best competitor's. This difference is, however, usually very small. Especially for plane sweeping we refer to Meltzer *et al.* [MYW05], who argued that in stereo instead of the "last percent" in energy one should rather improve the stereo model to obtain results closer to ground truth. A visual comparison in the supplemental material's Section A.3 confirms this observation. In some problem classes, *e.g.* stereo, dual solvers such as TRW-S suffer from loose LP relaxations; it is especially here where primal solvers such as ours can excel (see also Kappes *et al.* [KAH*15]).

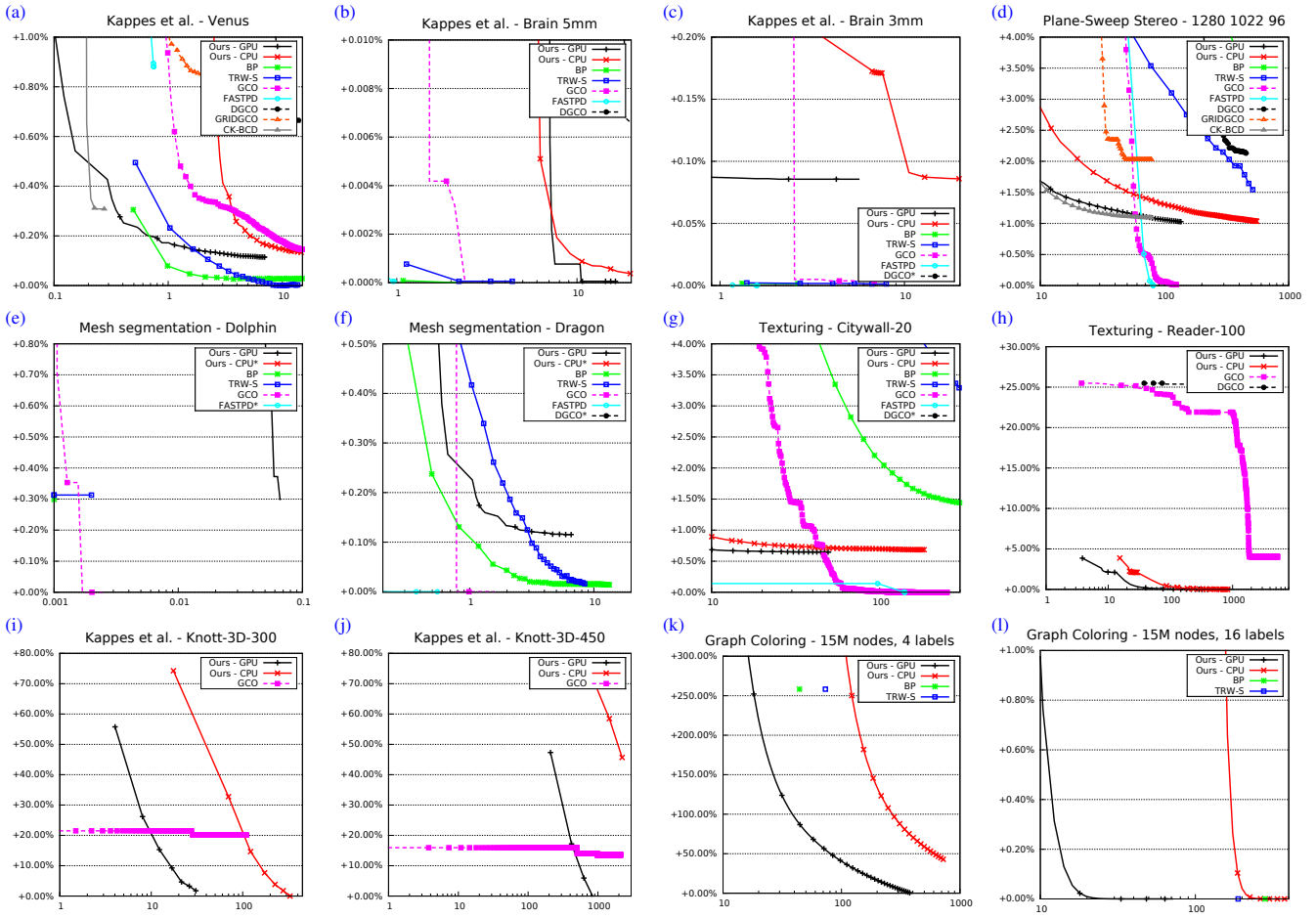

Figure 5: Relative energy difference to best solver's final solution over logarithmic time (in seconds) for different datasets. Figures A.1, A.2, and A.3 in the supplemental material additionally show datasets that we had to exclude here for brevity.

On very large datasets our solver outperforms the others by far: For Reader-100 (Figure 5h) it provides a better energy after 4 s than GCO (one of only two other solvers handling datasets of this size) does after 1.4 h. For 3D segmentation problems (5i and 5j)—both symmetric, purely smoothness cost based problems with extremely high label count of 4,000 resp. 16,000—we dominate the only competitor GCO in terms of solution quality. In both cases however, due to the size of the DP tables, the GPU reverts to solely global memory based computation. For Graph Coloring (5k) both competitors only found a solution that is 250 % worse than our solution.

We want to stress that in contrast to other solvers there is no dataset where our solver failed completely in terms of runtime or energy. Further, there is a range of datasets that many competitors cannot handle due to cost type, topology, or size. Notably, no competitor handles the combination of label costs and non-metric binary costs in Figure 7c. Especially among the parallel solvers (CK-BCD, DGCO, and GRIDGCO) our solver's generality stands out.

7.1. Influence of Heuristics and Randomization

In addition to coordinate sets our solver uses the spanning tree and region graph heuristics. This raises the question of how much

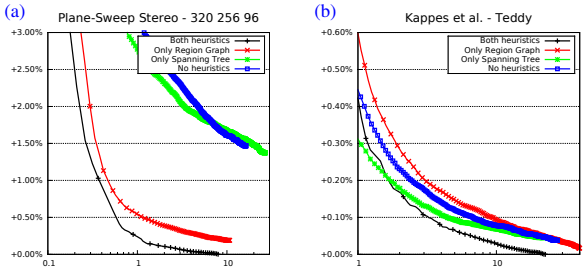

Figure 6: Heuristics' influence on smallest plane sweep and Teddy.

the performance can be attributed to them, especially compared to Chen and Koltun's [CK14] scanlines. Since the success of heuristics is problem-dependent, the answer varies between datasets. In plane sweeping (Figure 6a) there is 1.5 % difference between the final results with both and with no heuristics, which is mainly due to the region graph enforcing large homogeneous regions. The spanning tree heuristic only leads to a faster descent, not to a significantly lower final energy. The green and the red line in Figure 6a show the results when only spanning trees or region graphs are used before continuing with coordinate sets. For the Teddy (Figure 6b) the heuristics do not influence the result much.

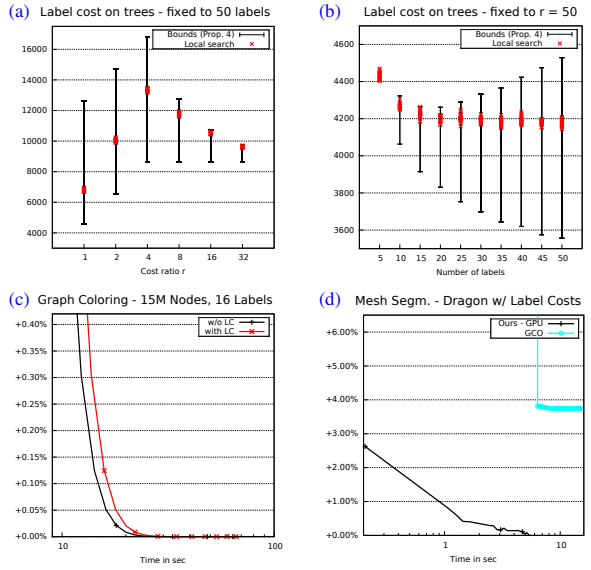

Figure 7: Label cost approximation performance: (a) and (b) are randomized tests (y-axis: absolute energy), (c) and (d) are real-world datasets (y-axis: energy relative to best solution).

These two datasets reflect the heuristics' properties quite well: In our experience, the region graph leads to a lower final energy, while the spanning tree heuristic only contributes to speed. Especially in graph coloring region graphs are not applied at all and yet we outperform TRW-S and BP. As our goal was rapid descent towards a competitive solution, we use the heuristics to speed up our BCD scheme. Without them, performance depends strongly on the dataset. We conclude that the combination of all three approaches is the key to good performance on different kinds of datasets.

As described in Section 6 each iteration of our solver samples a new coordinate set. This circumvents the sensitivity to local minima usually associated with tree MRFs. As mentioned, we initialize the process with a randomly sampled spanning tree without dependencies. Further experiments show that the initialization has only a negligible influence on the final results, independent of the dataset.

7.2. Evaluation of Label Costs

In Section 4.2 we described an algorithm that handles label costs on trees with local search on candidate label sets. We now examine its behavior by executing a range of experiments on random trees and trees from real-world problems. To examine the quality of approximate solutions, we first introduce upper and lower energy bounds on the optimal solution: Removing label costs from Equation (1)

$$\tilde{E}(f) = \sum_{i \in \mathcal{P}} D_i(f(i)) + \sum_{\{i,j\} \in \mathcal{N}} V_{\{i,j\}}(f(i), f(j)), \tag{5}$$

we obtain the bounds of Proposition 1:

Proposition 1 Let f be an optimum of Equation (1) and $\tilde{f}$ an optimum of Equation (5). The following holds for non-negative unary, binary and label costs (proof in supplemental material Section B.4):

$$\tilde{E}(\tilde{f}) \leq E(f) \leq E(\tilde{f}) \leq \tilde{E}(\tilde{f}) + \sum_{\ell \in \mathcal{L}} M_\ell. \tag{6}$$

An acceptable algorithm should at least be better than the upper bound. Both bounds can be calculated explicitly: For the lower bound we optimize Equation (5) without label costs and for the upper bound we add the costs of all labels. The bounds are closer to the global optimum if the ratio between label costs and unary/binary costs is small. Furthermore, if label costs (which are global) are much higher than most unary/binary costs (which are local), decisions done in early stages of the local search may turn out to be suboptimal in later stages. In order to quantify the behavior of our label cost approximation, we introduce the ratio $r = \sum_{i \in \mathcal{P}} \sum_{\ell \in \mathcal{L}} D_i(\ell) \,/\, \sum_{\ell \in \mathcal{L}} M_\ell$ between unary and label costs as key property of MRFs. Another key property is the number of labels.

In Figures 7a and b we show experiments where we analyzed these properties' influence. Each data point was generated by running 100 independent runs on random trees with 10^5 nodes. In Figure 7a we fixed the label count to 50 and varied r (x-axis). With growing r the label cost influence declines and the gap between upper and lower bound closes. The local search shows constant performance independent of r: It is always better than the upper bound, in many cases strongly superior. In Figure 7b we fixed r to 50 and varied the label count (x-axis). With more labels the gap between the bounds widens but local search shows constant performance: It is always better than the upper bound. Quality-wise, we conclude that the local search is useful for handling label costs on trees.

Beside these synthetic experiments we include two real-world datasets with label costs in Figures 7c and d. The graph coloring problem in 7c has a very small r. Our label cost algorithm finds a conflict-free labeling with only 9 labels (instead of 16 as determined by the algorithm without label costs). In the mesh segmentation problem in Figure 7d r is of medium value. There are only 5 labels, hence the multiple optimization runs of the local search scheme have no negative influence on the result. Thus, we outperform GCO in terms of quality and speed.

7.3. Discussion

Creating an efficient massively parallel software system poses many challenges. For the case of our MRF solver, we identified two key aspects for high performance: availability of a sufficient number of SIMD tasks to avoid under-utilization and efficient strategies to cope with frequent random memory access. Our GPU implementation addresses both: Since we parallelize over the feasible labels per node in the dynamic programming, the number of feasible labels determines how many SIMD tasks can be distributed on the GPU's PE groups. Given a sufficient ratio between tasks and PEs, the hardware can hide memory latency with computation. This is crucial for the performance of the memory-intensive dynamic programming procedure. Fast on-chip memory can furthermore be used for efficient data sharing among SIMD tasks. We found this to greatly reduce accesses to device memory. In combination with modern hardware that is able to coalesce parallel memory accesses, this helps to reduce the memory bandwidth bottleneck.

8. Conclusion

MRFs are an important and versatile computer vision tool, that can now be efficiently applied to even larger problems using our solver.

Our goal was to create an algorithm leveraging modern massively parallel hardware to quickly solve large irregular MRFs. With this in mind, our evaluation yields two main insights: First, for large graphs and label spaces our solver is highly effective in terms of solution quality and speed. Especially the texturing datasets (with sizes $> 10^7$) are very large real-world problems and yet our solver solves them in under a minute with energies better than state of the art solvers. Second, on small MRFs we achieve a solution quality and runtime that is in the worst case only slightly inferior to state of the art solvers. In fact (and in contrast to other solvers) there is no dataset where our solver failed completely. It instead fails gracefully on all tested datasets for which it was not designed.

We combine several interesting features in a single approach which enables our solver to handle arbitrary (including irregular) topologies, sparse cost matrices, arbitrary (including anti-metric) costs and label costs. This makes it applicable even to problems, where most other solvers cannot be used. Note that dual solvers are not always an option here, since they tend to be too memory intensive for large datasets or suffer from loose LP relaxations. Our label cost approximation on tree-shaped MRFs proved to be efficient and can even be applied to other dynamic programming based techniques such as belief propagation. In future work we plan to improve the coordinate set sampling, which is currently performed at random and to develop heuristics that guide the sampling towards the most effective coordinate sets in terms of energy decrease.

Acknowledgements. D. Thuerck and S. Widmer are supported by the 'Excellence Initiative' of the German Federal and State Governments and the Graduate School of Computational Engineering at TU Darmstadt. M. Waechter is grateful for support by the Intel Visual Computing Institute through the project 'RealityScan'.

Source Code. The source code for this paper is available at `www.gcc.tu-darmstadt.de/home/proj/mapmap`.

References

[AKB*12] ANDRES B., KAPPES J. H., BEIER T., KÖTHE U., HAMPRECHT F. A.: The Lazy Flipper: Efficient depth-limited exhaustive search in discrete graphical models. In *ECCV* (2012). 2

[Bes86] BESAG J.: On the statistical analysis of dirty pictures. *Journal of the Royal Statistical Society. Series B (Methodological)* (1986). 2

[BG12] BAGON S., GALUN M.: A multiscale framework for challenging discrete optimization. In *NIPS* (2012). 5

[BK11] BATRA D., KOHLI P.: Making the right moves: Guiding alpha-expansion using local primal-dual gaps. In *CVPR* (2011). 2

[BVZ01] BOYKOV Y., VEKSLER O., ZABIH R.: Fast approximate energy minimization via graph cuts. *PAMI* (2001). 2, 3, 8

[CK14] CHEN Q., KOLTUN V.: Fast MRF optimization with application to depth reconstruction. In *CVPR* (2014). 1, 2, 3, 8, 9

[COSH13] CRANDALL D., OWENS A., SNAVELY N., HUTTENLOCHER D.: SfM with MRFs: Discrete-continuous optimization for large-scale structure from motion. *PAMI* (2013). 1

[DB08] DELONG A., BOYKOV Y.: A scalable graph-cut algorithm for N-D grids. In *CVPR* (2008). 2

[DGVB12] DELONG A., GORELICK L., VEKSLER O., BOYKOV Y.: Minimizing energies with hierarchical costs. *IJCV* (2012). 2, 3, 4

[FCBZ12] FIX A., CHEN J., BOROS E., ZABIH R.: Approximate MRF inference using bounded treewidth subgraphs. In *ECCV* (2012). 3

[FH06] FELZENSZWALB P., HUTTENLOCHER D.: Efficient belief propagation for early vision. *IJCV* (2006). 3, 4, 5

[FHM13] FISHBAIN B., HOCHBAUM D. S., MUELLER S.: A competitive study of the pseudoflow algorithm for the minimum s-t cut problem in vision applications. *JRTIP* (2013). 3

[JB06] JUAN O., BOYKOV Y.: Active graph cuts. In *CVPR* (2006). 2

[JSH12] JAMRISKA O., SYKORA D., HORNUNG A.: Cache-efficient graph cuts on structured grids. In *CVPR* (2012). 2, 8

[KAH*15] KAPPES J. H., ANDRES B., HAMPRECHT F. A., SCHNÖRR C., NOWOZIN S., BATRA D., KIM S., KAUSLER B. X., KRÖGER T., LELLMANN J., KOMODAKIS N., SAVCHYNSKYY B., ROTHER C.: A comparative study of modern inference techniques for structured discrete energy minimization problems. *IJCV* (2015). 1, 2, 8

[KMMH06] KELM M., MUELLER N., MENZE B., HAMPRECHT F.: Bayesian estimation of smooth parameter maps for dynamic contrast-enhanced MR images with block-ICM. In *CVPR* (2006). 2

[KNKY11] KIM T., NOWOZIN S., KOHLI P., YOO C. D.: Variable grouping for energy minimization. In *CVPR* (2011). 1, 5

[Kol06] KOLMOGOROV V.: Convergent tree-reweighted message passing for energy minimization. *PAMI* (2006). 2, 8

[Kom10] KOMODAKIS N.: Towards more efficient and effective LP-based algorithms for MRF optimization. In *ECCV* (2010). 2

[KP08] KOMODAKIS N., PARAGIOS N.: Beyond loose LP-relaxations: Optimizing MRFs by repairing cycles. In *ECCV* (2008). 2

[KPT07] KOMODAKIS N., PARAGIOS N., TZIRITAS G.: MRF optimization via dual decomposition: Message-passing revisited. In *ICCV* (2007). 2

[KT07] KOMODAKIS N., TZIRITAS G.: Approximate labeling via graph cuts based on linear programming. *PAMI* (2007). 2, 8

[LRRB10] LEMPITSKY V., ROTHER C., ROTH S., BLAKE A.: Fusion moves for Markov random field optimization. *PAMI* (2010). 2

[MYW05] MELTZER T., YANOVER C., WEISS Y.: Globally optimal solutions for energy minimization in stereo vision using reweighted belief propagation. In *ICCV* (2005). 2, 8

[SH13] SHEKHOVTSOV A., HLAVÁČ V.: A distributed mincut/maxflow algorithm combining path augmentation and push-relabel. *IJCV* (2013). 2, 8

[Shi94] SHIMONY S. E.: Finding MAPs for belief networks is NP-hard. *Artificial Intelligence* (1994). 3

[SHK*14] SCHARSTEIN D., HIRSCHMÜLLER H., KITAJIMA Y., KRATHWOHL G., NEŠIĆ N., WANG X., WESTLING P.: High-resolution stereo datasets with subpixel-accurate ground truth. In *GCPR* (2014). 1, 2

[STT10] SOLOMON S., THULASIRAMAN P., THULASIRAM R. K.: Exploiting parallelism in iterative irregular maxflow computations on GPU accelerators. In *HPCC* (2010). 3

[Sze11] SZELISKI R.: *Computer Vision – Algorithms and Applications*. Springer London, 2011. 1, 3, 4

[SZS*08] SZELISKI R., ZABIH R., SCHARSTEIN D., VEKSLER O., KOLMOGOROV V., AGARWALA A., TAPPEN M., ROTHER C.: A comparative study of energy minimization methods for Markov random fields with smoothness-based priors. *PAMI* (2008). 1, 8

[Vek05] VEKSLER O.: Stereo correspondence by dynamic programming on a tree. In *CVPR* (2005). 3, 4, 5, 7

[Vek15] VEKSLER O.: Efficient parallel optimization for Potts energy with hierarchical fusion. In *CVPR* (2015). 2, 3

[VN08] VINEET V., NARAYANAN P.: CUDA cuts: Fast graph cuts on the GPU. In *CVPR Workshops* (2008). 3

[WMG14] WAECHTER M., MOEHRLE N., GOESELE M.: Let there be color! Large-scale texturing of 3D reconstructions. In *ECCV* (2014). 1