Fast Parallel and Adaptive Updates for Dual-Decomposition Solvers

Özgür Sümer
osumer@cs.uchicago.edu
Department of Computer Science
University of Chicago, IL

Umut A. Acar[*]
umut@mpi-sws.org
Max-Planck Institute for Software Systems
Germany

Alexander T. Ihler
ihler@ics.uci.edu
Information and Computer Science
University of California–Irvine

Ramgopal R. Mettu[†]
mettu@ecs.umass.edu
Department of Electrical and Computer Engineering
University of Massachusetts Amherst, MA

Abstract

Dual-decomposition (DD) methods are quickly becoming important tools for estimating the minimum energy state of a graphical model. DD methods decompose a complex model into a collection of simpler subproblems that can be solved exactly (such as trees), that in combination provide upper and lower bounds on the exact solution. Subproblem choice can play a major role: larger subproblems tend to improve the bound more per iteration, while smaller subproblems enable highly parallel solvers and can benefit from re-using past solutions when there are few changes between iterations.

We propose an algorithm that can balance many of these aspects to speed up convergence. Our method uses a cluster tree data structure that has been proposed for adaptive exact inference tasks, and we apply it in this paper to dual-decomposition approximate inference. This approach allows us to process large subproblems to improve the bounds at each iteration, while allowing a high degree of parallelizability and taking advantage of subproblems with sparse updates. For both synthetic inputs and a real-world stereo matching problem, we demonstrate that our algorithm is able to achieve significant improvement in convergence time.

1 Introduction

As multi-core computers become commonplace, researchers and practitioners are increasingly looking to parallelization to assist with difficult inference tasks in a variety of applications, including computer vision, automated reasoning, and computational biology. Thus, improving the efficiency of approximate inference algorithms has the potential to advance research in each of these areas. In particular, *dual decomposition* (DD) methods are starting to emerge as a powerful set of tools for parallelizable approximate reasoning in graphical models. At essence, DD methods decompose a complex model into a collection of simpler models (subgraphs) that are forced to agree on their intersections. By relaxing these constraints with Lagrange multipliers, one obtains a bound on the optimal solution that can be iteratively tightened. Since these simpler subproblems can be solved independently in each iteration, this approach is a natural choice for multi-core and parallel architectures.

However, parallelizing a dual decomposition solver can often have a hidden cost in terms of the per-iteration performance of the algorithm. In particular, choosing very small subproblems leads to a highly parallel solver but can also significantly reduce its progress at each iteration, leading to less than the expected improvement in overall speed. On the other hand, collections of larger subproblems often require fewer iterations, but are less parallelizable.

In this paper, we present an approach to parallelizing dual-decomposition solvers using *tree contraction* (Reif and Tate 1994; Acar et al. 2004; Acar, Blelloch, and Vittes 2005), a well-known technique for parallel computation. Tree contraction has been used to parallelize exact inference in graphical models (Pennock 1998; Xia and Prasanna 2008; Namasivayam, Pathak, and Prasanna 2006), has not been applied to approximate inference methods. In particular, we make use of the *cluster tree* data structure for adaptive inference (Acar et al. 2007; 2008; 2009a), and show that it can be used in a way that combines the per-iteration advantages of large subproblems while also enabling a high degree of parallelism. The cluster-tree data structure can be obtained by applying self-adjusting-computation (Acar et al. 2009b; Hammer, Acar, and Chen 2009; Ley-Wild, Fluet, and Acar 2008) to tree contraction and have been previously used to solve dynamic problems on trees (Acar et al. 2004; Acar, Blelloch, and Vittes 2005) in the sequential, single-processor, context. In addition to being highly parallelizable, the adaptive properties of the cluster tree allow minor changes to a model to be incorporated and re-solved in far less time than required to solve from scratch. We show that in DD solvers this adaptivity can be a major benefit, since often only small portions of the subproblems' parameters are modified at each iteration.

We demonstrate that a DD solver using our cluster tree approach can improve its time to convergence significantly over other approaches. For random grid-like graphs we obtain one to two orders of magnitude speedup. We also use our solver for a real-world stereo matching problem, and

[*]U. A. Acar is supported by a gift from Intel and Microsoft Research.

[†]R. R. Mettu is supported by a National Science Foundation CAREER Award (IIS-0643768).

over a number of data sets show a factor of about 2 improvement over other approaches.

2 Background

Graphical models give a formalism for describing factorization structure within a probability distribution or energy function over variables $V = \{x_1, \ldots, x_n\}$. For notational simplicity we consider pairwise models where for a graph $G = (V, E)$, the energy function E_P decomposes into a collection of singleton and pairwise functions:

$$E_P = \sum_{i \in V} \theta_i(x_i) + \sum_{(i,j) \in E} \theta_{ij}(x_i, x_j).$$

which are nonzero over neighboring pairs, $(i, j) \in E$. A common task is to find the optimal assignment to the variables that minimizes E_P; this corresponds to the "most probable explanation" or MPE estimate.

Dual decomposition (DD) methods further split the energy function into an additive set of "subproblems"

$$E_D = \sum_t \Big[\sum_{i \in V} \theta_i^t(x_i^t) + \sum_{(i,j) \in E} \theta_{ij}^t(x_i^t, x_j^t) \Big]$$

where we require that $\theta_i(x_i) = \sum_t \theta_i^t(x_i)$ and $\theta_{ij}(x_i, x_j) = \sum_t \theta_{ij}^t(x_i, x_j)$ for all i, j. The values of θ_{ij}^t are chosen to be non-zero on a strictly simpler graph than G (for example, a tree), so that each subproblem t can be solved easily by dynamic programming (or equivalently variable elimination).

If all copies x_i^t of each variable x_i are forced to be equal, both problems are equivalent. However, by relaxing this constraint and enforcing it with Lagrange multipliers, we obtain a collection of simpler problems that when solved individually, lower bounds the original energy (a Lagrangian dual function). Typical DD solvers maximize this dual lower bound using a projected subgradient update. Suppose that $\{x_i^t\}$ are the copies of variable x_i, with optimal assignments $\{a_i^t\}$ for $t = 1 \ldots T$. Then we modify θ_i^t as

$$\theta_i^t(x) = \theta_i^t(x) - \gamma\Big(\delta(x = a_i^t) - \frac{1}{T}\sum_u \delta(x = a_i^u)\Big) \quad (1)$$

where $\delta(\cdot)$ is the Kronecker delta function and γ is a step-size constant. It is easy to see that this update maintains the constraints on the θ_i^t. If at any point a solution is found in which all variable copies share the same value, this configuration must be the MPE.

Dual decomposition solvers are closely related to LP-based loopy message passing algorithms (Wainwright, Jaakkola, and Willsky 2005; Globerson and Jaakkola 2007), which solve the same dual using a coordinate ascent fixed point update. However, these algorithms can have sub-optimal fixed points, so gradient and "accelerated" gradient methods (Johnson, Malioutov, and Willsky 2007; Jojic, Gould, and Koller 2010) are often preferred. In this paper we focus on the standard projected subgradient method.

DD methods leave the choice of subproblems to the user. All problem collections that include equivalent sets of cliques (for example, any collection of trees that covers all edges of G) can be shown to have the same dual-optimal bound, but the actual choice of problems can significantly affect convergence speed. For example, one simple

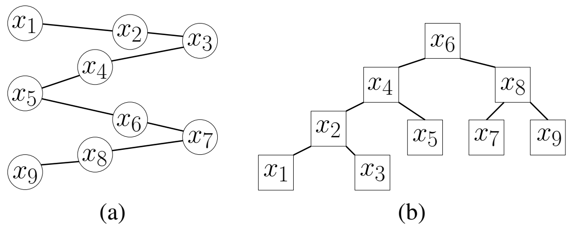


Figure 1: **(a)** A Markov chain subproblem. **(b)** A cluster tree formed by eliminating in the order: $x_1, x_3, x_2, x_5, x_7, x_9, x_4, x_8, x_6$. The depth of the cluster tree is $O(\log n)$ whereas the depth of the original chain is $O(n)$.

option is to include one subproblem per edge in the original graph; this leads to a large number of simple problems, which can then be easily parallelized. However, as observed in (Komodakis, Paragios, and Tziritas 2007), choosing larger subproblems can often improve the convergence rate (i.e., the increase in the lower bound per iteration) at the possible expense of parallelization. For example, single-subproblem models (Johnson, Malioutov, and Willsky 2007; Yarkony, Fowlkes, and Ihler 2010) create a single "covering" tree over several copies of each variable. This approach provides good convergence properties but is not easily amenable to parallelization.

Another advantage we propose for small subproblems is their ability to be *adaptive*, or more specifically to re-use previous iteration's solution. The subgradient update (1) depends only on the solution a^t of each subproblem t; if all parameters of a subproblem are unchanged, its solution remains valid and we need not re-solve it. We show in Sections 4 and 5 that this can lead to considerable computational advantages. However, although this is common in very small subproblems (such as individual edges), for larger problems with better convergence rates it becomes less likely that the problem will not be modified.

Thus, collections of small problems have significant speed (time per iteration) advantages, but larger problems have typically better convergence rates, or fewer iterations required. The focus of the remainder of this paper is to present a new framework that captures both the convergence properties of single-subproblem approaches, and the update speed of many, small subproblems.

3 Cluster tree data structure

Parallel calculation of tree-structured formulas has been studied in the algorithms community using a technique called *tree contraction* (Reif and Tate 1994). Algorithms based on this idea have been applied to speed up exact inference tasks in a variety of settings (Pennock 1998; Xia and Prasanna 2008; Acar et al. 2008; 2009a; Namasivayam, Pathak, and Prasanna 2006); here we give a brief summary of one such data structure called a *cluster tree*, and show how it can be applied effectively to improve DD solvers in Section 4.

As a motivating example, consider a subproblem that con-

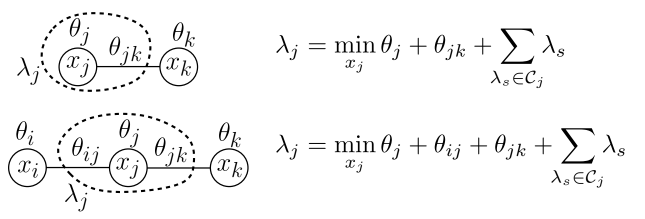


Figure 2: Eliminating x_j computes a message λ_j as shown, where $\mathcal{C}_j$ is the set of messages attached to x_j or any of its incident edges.

sists of a single Markov chain (Figure 1a). A standard solver for this model works by dynamic programming, sequentially eliminating each x_i from leaves to root and computing a message λ_i, interpreted as a "cost to go" function, then back-solving from root to leaves for an optimal configuration $\{a_i\}$. This process is hard to parallelize, since any exact solver must propagate information from x_1 to x_9 and thus requires a sequence of $O(n)$ operations in this framework.

The cluster tree data structure is based on the observation that the lack of effective parallelism is due to the "unbalanced" nature of the chain. By also "clustering" (i.e., eliminating) non-leaf variables in the model, specifically degree-two nodes, we create a *balanced* elimination ordering whose operations can more easily parallelized. The new partial ordering is shown in Figure 1b. Degree-two eliminations introduce some ambiguity to the notion of "messages", since we cannot determine the direction (recipient) of the message λ_j when x_j is eliminated. Instead, we simply create a new edge joining its neighbors x_i, x_k and attach λ_j to the edge. It also results in a small additional overhead; for variables of size d, degree-two eliminations require $O(d^3)$ cost compared to $O(d^2)$ for leaf-only eliminations, but we shall see that the framework has advantages that can outweigh this cost. An illustration and equations for λ_j are given in Figure 2.

Concretely, the cluster tree is a rooted tree where x_j is a parent of x_i if x_i's message λ_i is used during computation of λ_j. This data structure tracks the dependency among messages; that is, computation of a parent message requires all child messages be computed beforehand. The set of messages passed by x_j's children in the cluster tree is denoted $\mathcal{C}_j$. Moreover, for each x_j, we also store the set E_j of its neighbors at the time of its elimination. For example in Figure 2 when a leaf node x_j is eliminated, $E_j = \{x_k\}$, while for a degree-2 node $E_j = \{x_i, x_k\}$. Previous work shows that, with respect to the number of variables, the cluster tree can be constructed in linear time, and has logarithmic depth (Acar et al. 2008; 2009a). Moreover, the balanced shape of the cluster tree means that many branches can be computed in parallel; see Section 4. Using similar ideas, (Pennock 1998) showed that, with a sufficient number of processors, parallel exact inference requires $O(\log n)$ time.

Another major advantage of the cluster tree is its *adaptivity*, when it is used to compute an optimal configuration similarly to dynamic programming. Having computed the messages λ_i, we perform a downward pass (root to leaves)

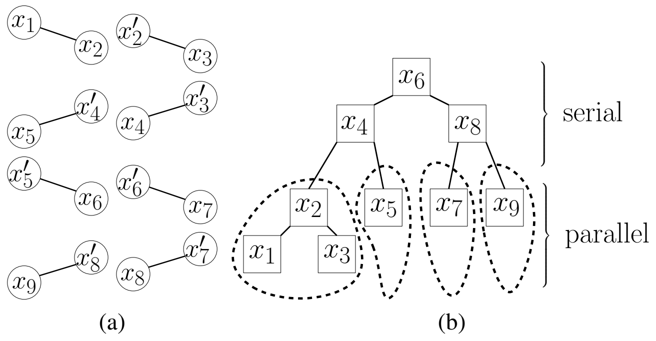


Figure 3: **(a)** Dual-decomposition using independent edges is highly parallelizable but slow to converge. **(b)** The cluster tree is also easily parallelizable, but since the underlying model is a covering tree, convergence is not compromised.

to select an optimal configuration for each variable. When this downward pass reaches variable x_j, we choose the optimal configuration using

$$x_j^* = \arg\min_{x_j} \; \theta_j + \sum_{x_k \in E_j} \theta_{jk}\delta(x_k = x_k^*) + \sum_{\lambda_s \in \mathcal{C}_j} \lambda_s. \quad (2)$$

We shall see in Section 4 that the cluster tree structure also allows us to efficiently update our solutions at each iteration, giving an additional speed-up.

4 Parallelism without sacrifice

As previously discussed, the benefit of parallelizing dual-decomposition inference methods can be lost if the subproblems are not chosen appropriately. For a concrete example, consider again the Markov chain of Figure 1a. A decomposition into edges (Figure 3a) achieves high parallel efficiency: each edge is optimized independently in parallel and disagreements are successively resolved by parallel independent projected subgradient updates. Given enough processors, each iteration in this setup requires constant time, but overall inference still requires $\Omega(n)$ iterations for this procedure to converge. Thus there is no substantial gain from parallelizing inference on even this simple model, since in the sequential case we reach the solution in $O(n)$ time. Thus, we must balance the degree of parallelism and the converge rate to achieve the best possible speedup. In the remainder of this section, we discuss how the cluster tree data structure permits us to achieve this balance, and how its adaptive properties provide an additional speedup near convergence.

Parallelization with a Cluster Tree

A cluster tree structure can manage both long-range influence in each iteration (improving convergence rate) as well as parallelization obtained by exploiting its balanced nature. To perform parallel computations in a cluster tree with n nodes, we split the tree into a set of *parallel trees* and a *serial tree*, as shown in Figure 3. For bottom-up computations, we can assign a processor to each parallel tree. When these computations are complete, the remainder of the bottom-up computation is performed on a single processor. Top-down

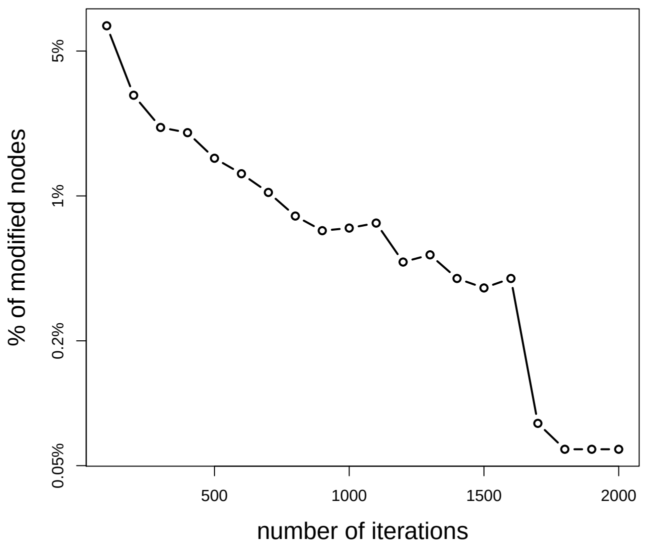


Figure 4: The number of changes made to the parameters by the projected subgradient updates, measured on a stereo matching example (Venus) from the experimental section.

computations are performed analogously, except that they start with the serial tree and then continue in parallel on the parallel trees. The number of parallel trees defines both the level of parallelism as well as the depth of the serial tree, and thus these two choices must be balanced. In practice we choose the depth of the serial tree to be half of the depth of the cluster tree, to keep synchronization costs low. Then, for a cluster tree with n nodes we achieve parallelism of $\sqrt{n}$, while preserving a high convergence rate.

Adaptivity for subgradient updates

Another major source of improvement in speed is obtained by judiciously reusing solutions at the previous iteration. In particular, the subgradient update (1) depends only on the optimal configuration a^t of each subproblem t, and modifies only those subproblems that do not share the optimal value of some x_i. Many θ_i^t, then, are not modified from iteration to iteration; the updates become sparse. This behavior is common in many real-world models: Figure 4 illustrates for a stereo matching problem, showing that in practice the number of updates diminishes significantly near convergence. The standard approach recomputes the optimal values for every variable in an updated subproblem θ^t.

This observation motivates an *adaptive* subgradient calculation, in which we leverage the previous iteration's solution to speed up the next. In collections of very small subproblems, this is easily accomplished: we can simply preserve the solution of any subproblem whose potentials θ^t were not modified at all. While this may appear hard to use in a cover tree representation (in which all variables are present), in this section we show how to use the cluster tree data structure to leverage update sparsity and give further speedup even if the underlying subproblem is a cover tree.

Let G be the tree corresponding to the subproblem θ^t and T be the associated cluster tree constructed as described in Section 3. Suppose that the projected subgradient updates modify singleton potentials $\theta_1, \theta_2, \ldots, \theta_\ell$. For the next iteration, we must: (i) update the cluster tree given the new potential values, and (ii) compute any changed entries in the optimal configuration.

To perform (i), we update the cluster tree messages in a bottom-up fashion, starting at nodes that correspond to the changed potentials and updating their ancestors. For each recomputation, we flag the messages as having been modified. Using a depth argument, this step recomputes only $O(\ell \log n/\ell)$ many messages.

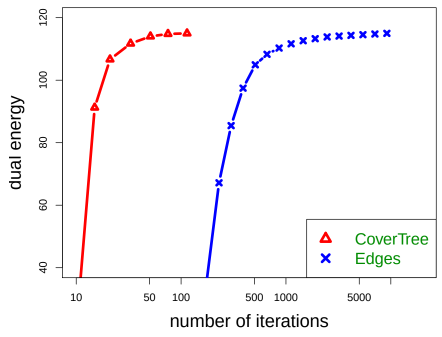
For part (ii), we efficiently compute a new optimal configuration by traversing top-down. We begin by selecting an optimal configuration for the root node x_r of the cluster tree. We then recursively proceed to any children x_i for which either any λ_j was modified for which $x_j \in C_i$, or the optimal configuration of the variables in E_i were modified. If the projected subgradient update modifies k configurations, we can solve the subproblem in $O(k \log n/k)$ time: potentially much faster than doing the same updates in the original tree. For illustration, consider Figure 4; between iterations 1700 and 2100, the re-computation required is about one 1000^{th} of the 1^{st} iteration. This property alone can provide significant speed-up near convergence.

5 Experiments

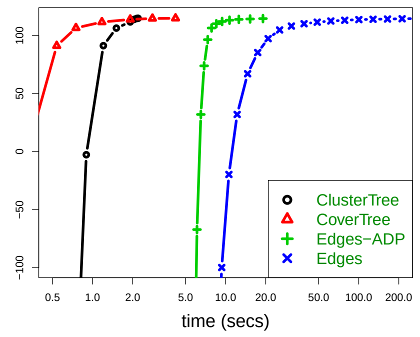
Our cluster tree representation combines the benefits of large subproblems (faster convergence rates) with efficient parallelization and effective solution re-use. To assess its effectiveness, we compare our algorithm against several alternative subproblem representations on both synthetic and real-world models. We focus on the general case of irregularly connected graphs for our experiments.[1]

We compared our framework for performing dual-decomposition against three different algorithms: COVERTREE, EDGES and EDGES-ADP. COVERTREE decomposes the graph using a cover tree and updates the model without any parallelism or adaptivity. On the other end of the DD spectrum, the EDGES and EDGES-ADP algorithms decompose the graph into independent edges and update them in parallel. EDGES-ADP is the trivially adaptive version of the EDGES algorithm, in which only the edges adjacent to an modified node are re-solved. We refer to our algorithm as CLUSTERTREE; it uses the same cover tree graph as COVERTREE in all of our experiments. All algorithms were implemented in Cilk++ (Leiserson 2009) without locks. For synchronization we used Cilk++ reducer objects (variables that can be safely used by multiple strands running in parallel). All experiments in this section were performed on a 1.87Ghz 32-core Intel Xeon processor.

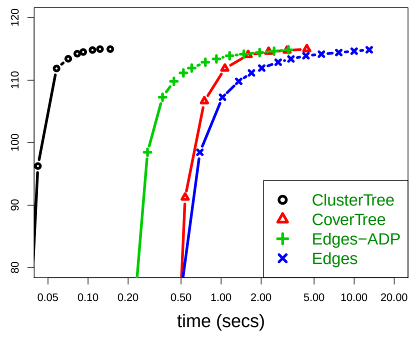
[1]On very regular graphs, it is often possible to hand-design updates that are competitive with our approach; for example on grid-structured graphs, the natural decomposition into rows and columns that results in $O(\sqrt{n})$ parallel subproblems of length $O(\sqrt{n})$ and will often attain a similar balance of adaptive, parallel updates and good convergence rate.



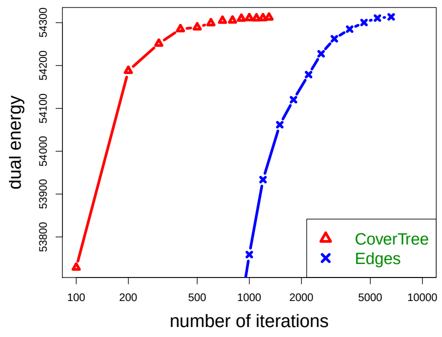
(a) Convergence rate



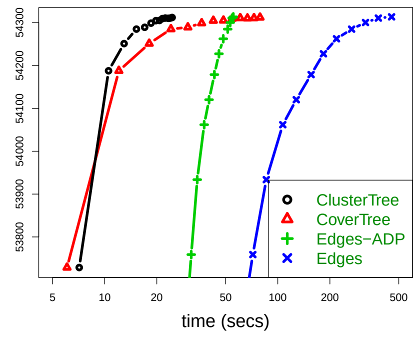
(b) Single-core convergence time



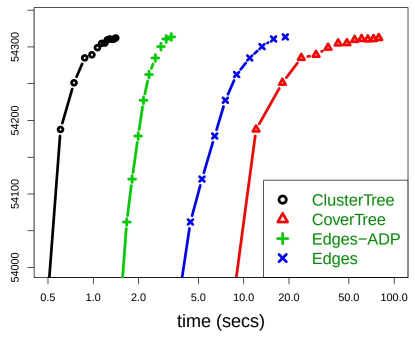
(c) 32-core convergence time



(d) Convergence rate for Venus



(e) Single-core convergence time for Venus



(f) 32-core convergence time for Venus

Figure 5: Representative convergence results on a random graph problem with 20000 nodes **(a-c)** and for stereo matching on the "Venus" dataset **(d-f)**. (Best viewed in color.)

Synthetic Examples

We first test our approach on random, irregular graphical models. We generated graphs over n variables $x_1, \ldots, x_n$, each with domain size $d = 8$. The graph edges were generated at random by iterating over variables $x_3, \ldots, x_n$. For node x_i, we choose 2 random neighbors without replacement from the previous nodes $\{x_{i-1}, x_{i-2}, \ldots, x_1\}$ using a geometric distribution with probabilities $p = 1/2$ and $q = 1/\sqrt{n}$, respectively. With these settings, every node x_i is expected to have two neighbors whose indices are close to i and two neighbors whose indices are roughly $i - \sqrt{n}$ and $i + \sqrt{n}$. Although highly irregular, the generated graph has characteristics similar to a grid with raster ordering, where each node x_i inside the grid has four neighbors indexed $x_{i-1}, x_{i+1}, x_{i-\sqrt{n}}$ and $x_{i+\sqrt{n}}$. Node potentials $\theta_i(x_i)$ are drawn from a Gaussian distribution with mean 0 and standard deviation $\sigma = 4$. Edge potentials $\theta_{ij}(x_i, x_j)$ follow the Potts model, so that $\theta_{ij}(x_i, x_j) = \delta(x_i \neq x_j)$.

We ran our algorithms for graph sizes ranging from 500 to $100,000$. To control for step size effects, γ was optimized for each algorithm and each problem instance. All algorithms were run until agreement between variable copies (indicating an exact solution). For COVERTREE and CLUSTERTREE, we used a breadth-first search tree as the cover tree. Figure 5a–c gives a comparison of convergence results for a representative model with $n = 20000$.

As expected, the cover tree has a better convergence rate than using independent edges as subproblems (see Figure 5a). When the algorithms were executed serially (see Figure 5b), although initally slower CLUSTERTREE catches up to finish faster than COVERTREE (due to adaptivity), and remains faster than EDGES-ADP (due to a better convergence rate). With parallel execution, we observe a speedup of roughly 20× for CLUSTERTREE, EDGES-ADP and EDGES; see Figure 5c. We can see that with parallelism, although EDGES-ADP is preferable to COVERTREE, CLUSTERTREE finishes roughly two orders of magnitude faster than the other algorithms.

We also consider the convergence time of each algorithm as the graph size increases (see Figure 6). For relatively small graphs (e.g. $n = 500$) the difference is negligible; however, as we increase the number of nodes, the CLUSTERTREE converges significantly more quickly than the other algorithms.

Stereo matching with super-pixels

The stereo matching problem estimates the depth of objects in a scene given two images, as if seen from a left and right eye. This is done by estimating the *disparity*, or horizontal shift in each object's location between the two images.

It is common to assume that the disparity boundaries coincide with color or image boundaries. Thus, one approach estimating stereo depth is to first segment the image into super-pixels, and then optimize a graphical model

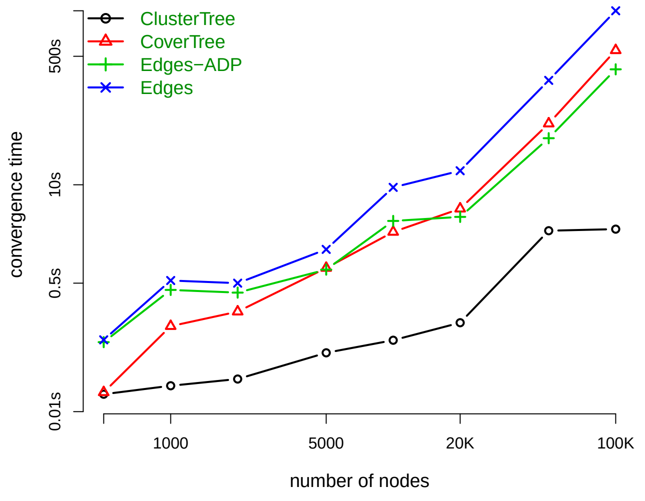


Figure 6: For randomly generated graphs, our algorithm achieves a significant speedup as the model size increases.

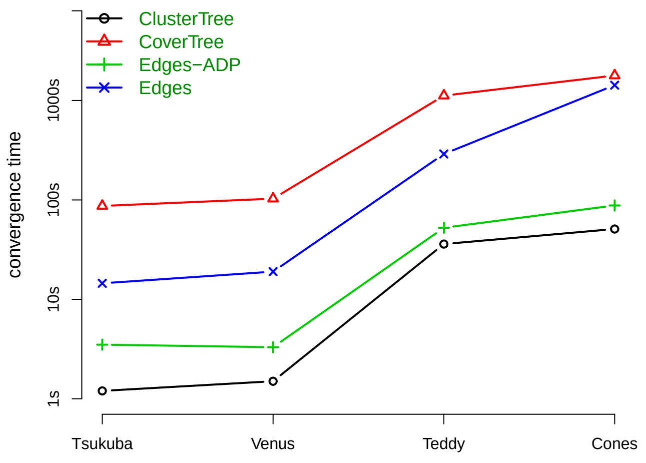


Figure 7: For images from the Middlebury dataset, our algorithm is about twice as fast as EDGES-ADP.

representing the super-pixels; see (Hong and Chen 2004; Trinh 2008). This approach allows stereo matching to be performed on much larger images. We studied the performance of our algorithm for the task of stereo matching using a model constructed from a segmented image in this manner.

To define a graphical model G given super-pixels $\{s_i, \ldots, s_n\}$, we define a node for each super-pixel and add an edge (s_i, s_j) if they contain adjacent pixels in the reference image. The node potentials are defined as the cumulative truncated absolute color differences between corresponding pixels for each disparity:

$$\theta_i(d) = \sum_{(x,y)\in s_i} \min\{|\mathcal{I}_L(x,y) - \mathcal{I}_R(x-d,y)|, 20\}$$

where $\mathcal{I}_L$ and $\mathcal{I}_R$ are the intensities of the left and right image, respectively. The edge potentials are defined as

$$\theta_{ij}(d_1, d_2) = 5 \cdot E(s_i, s_j) \cdot \min\left\{|d_1 - d_2|^{1.5}, 5\right\}$$

where $E(s_i, s_j)$ is the number of adjacent pixel pairs (p, q) where $p \in s_i$ and $q \in s_j$. This is a common energy function for grids, applied to super-pixels by (Hong and Chen 2004; Szeliski et al. 2008).

We tested our algorithm by constructing the above model for four images (Tsukuba, Venus, Teddy and Cones) from the Middlebury stereo data set (Scharstein and Szeliski 2001; Scharstein 2003), using the SLIC program (Achanta et al. 2010) to segment each input image into 5000 superpixels. For COVERTREE and CLUSTERTREE, we used the maximum-weight spanning tree of G (with weights $E(s_i, s_j)$) as part of the cover tree; this is a common choice for stereo algorithms that use dynamic programming (Veksler 2005). Since the model's gap between distinct energies is at least 1 the algorithms are considered converged when their lower bound is within 1 of the optimal energy.

As with synthetic graphs, for the image datasets we observed that CLUSTERTREE inherits the improved convergence rate of COVERTREE but parallelizes well and thus gives much better overall performance than EDGES or EDGES-ADP. Representative serial and parallel executions of the algorithms are shown for the Venus dataset in Figure 5d–f, while convergence times are shown for all datasets in Figure 7. While we still observe that COVERTREE has a better convergence rate than EDGES, it is less dramatic than in the synthetic models (Figure 5a vs. d); this is likely due to the presence of strong local information in the model parameters θ_i. This fact, along with most modifications also being local, means that EDGES-ADP manages to outperform COVERTREE in the serial case (Figure 5e). In the parallel case, CLUSTERTREE remains ahead, with a speedup of about 2× over EDGES-ADP.

6 Conclusion

We have introduced a novel framework for dual-decomposition solvers that balances the intrinsic tradeoffs between parallelism and convergence rate. For the choice of subproblems, we use a cover tree to obtain rapid convergence, but use a balanced *cluster tree* data structure to enable efficient subgradient updates. The cluster tree is guaranteed to have logarithmic depth, and so is amenable to a high degree of parallelism. Moreover, it can be used to efficiently update optimal configurations during each subgradient iteration. For randomly generated models, our approach is up to two orders of magnitude faster than other methods as the model size becomes large. We also show that for the real-world problem of stereo matching, our approach is roughly twice as fast as other methods.

References

Acar, U. A.; Blelloch, G.; Harper, R.; Vittes, J.; and Woo, M. 2004. Dynamizing static algorithms with applications to dynamic trees and history independence. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 531–540.

Acar, U.; Ihler, A. T.; Mettu, R. R.; and Sümer, O. 2007.

Adaptive Bayesian inference. In *Advances in Neural Information Processing Systems (NIPS)*. MIT Press.

Acar, U. A.; Ihler, A. T.; Mettu, R. R.; and Sümer, O. 2008. Adaptive Bayesian inference in general graphs. In *Proceedings of the 24th Annual Conference on Uncertainty in Artificial Intelligence (UAI)*, 1–8.

Acar, U. A.; Ihler, A. T.; Mettu, R. R.; and Sümer, O. 2009a. Adaptive updates for maintaining MAP configurations with applications to bioinformatics. In *Proceedings of the IEEE Workshop on Statistical Signal Processing*, 413–416.

Acar, U. A.; Blelloch, G. E.; Blume, M.; Harper, R.; and Tangwongsan, K. 2009b. An experimental analysis of self-adjusting computation. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 32(1):3:1–3:53.

Acar, U. A.; Blelloch, G. E.; and Vittes, J. L. 2005. An experimental analysis of change propagation in dynamic trees. In *Workshop on Algorithm Engineering and Experimentation*.

Achanta, R.; Shaji, A.; Smith, K.; Lucchi, A.; Fua, P.; and Süsstrunk, S. 2010. SLIC Superpixels. Technical report, EPFL, École Polytechnique Fédérale De Lausanne.

Globerson, A., and Jaakkola, T. 2007. Fixing max-product: Convergent message passing algorithms for MAP LP-relaxations. In *Advances in Neural Information Processing Systems (NIPS)*.

Hammer, M. A.; Acar, U. A.; and Chen, Y. 2009. CEAL: a C-based language for self-adjusting computation. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*.

Hong, L., and Chen, G. 2004. Segment-based stereo matching using graph cuts. In *Computer Vision and Pattern Recognition (CVPR)*, 74–81.

Johnson, J. K.; Malioutov, D. M.; and Willsky, A. S. 2007. Lagrangian relaxation for MAP estimation in graphical models. In *In Allerton Conference Communication, Control and Computing*.

Jojic, V.; Gould, S.; and Koller, D. 2010. Fast and smooth: Accelerated dual decomposition for MAP inference. In *Proceedings of International Conference on Machine Learning (ICML)*.

Komodakis, N.; Paragios, N.; and Tziritas, G. 2007. MRF optimization via dual decomposition: Message-passing revisited. In *International Conference on Computer Vision (ICCV)*.

Leiserson, C. E. 2009. The cilk++ concurrency platform. In *Proceedings of the 46th Annual Design Automation Conference (DAC)*, 522–527. ACM.

Ley-Wild, R.; Fluet, M.; and Acar, U. A. 2008. Compiling self-adjusting programs with continuations. In *Proceedings of the International Conference on Functional Programming*.

Namasivayam, V. K.; Pathak, A.; and Prasanna, V. K. 2006. Scalable parallel implementation of bayesian network to junction tree conversion for exact inference. In *Information Retrieval: Data Structures and Algorithms*, 167–176. Prentice-Hall PTR.

Pennock, D. M. 1998. Logarithmic time parallel Bayesian inference. In *Proceedings 14th Annual Conference on Uncertainty in Artificial Intelligence (UAI)*, 431–438.

Reif, J. H., and Tate, S. R. 1994. Dynamic parallel tree contraction. In *Proceedings 5th Annual ACM Symp. on Parallel Algorithms and Architectures*, 114–121.

Scharstein, D., and Szeliski, R. 2001. A taxonomy and evaluation of dense two-frame stereo correspondence algorithms. *International Journal of Computer Vision* 47:7–42.

Scharstein, D. 2003. High-accuracy stereo depth maps using structured light. In *Computer Vision and Pattern Recognition (CVPR)*, 195–202.

Szeliski, R.; Zabih, R.; Scharstein, D.; Veksler, O.; Kolmogorov, V.; Agarwala, A.; Tappen, M.; and Rother, C. 2008. A comparative study of energy minimization methods for Markov random fields with smoothness-based priors. *IEEE Transactions on Pattern Analysis and Machine Intelligence (PAMI)* 30:1068–1080.

Trinh, H. 2008. Efficient stereo algorithm using multiscale belief propagation on segmented images. In *British Machine Vision Conference (BMVC)*.

Veksler, O. 2005. Stereo correspondence by dynamic programming on a tree. In *Computer Vision and Pattern Recognition (CVPR)*, CVPR '05, 384–390. IEEE Computer Society.

Wainwright, M.; Jaakkola, T.; and Willsky, A. 2005. MAP estimation via agreement on (hyper)trees: message-passing and linear programming approaches. *IEEE Transactions on Information Theory* 51(11):3697–3717.

Xia, Y., and Prasanna, V. K. 2008. Junction tree decomposition for parallel exact inference. In *IEEE International Parallel and Distributed Preocessing Symposium*, 1–12.

Yarkony, J.; Fowlkes, C.; and Ihler, A. 2010. Covering trees and lower-bounds on quadratic assignment. In *Computer Vision and Pattern Recognition (CVPR)*, 887–894.