Parallel Algorithms for Graph Optimization using Tree Decompositions

Blair D. Sullivan
Oak Ridge National Laboratory
P.O. Box 2008 MS6015
Oak Ridge, TN 37831-6015
sullivanb@ornl.gov

Dinesh Weerapurage
Link Analytics
1050 Crown Pointe Pkwy, Ste 1580
Atlanta, GA 30338
dinesh.weerapurage@linkanalytics.com

Chris Groër
Link Analytics
1050 Crown Pointe Pkwy, Ste 1580
Atlanta, GA 30338
chris.groer@linkanalytics.com

[*] ***Abstract*—Although many $\mathcal{NP}$-hard graph optimization problems can be solved in polynomial time on graphs of bounded tree-width, the adoption of these techniques into mainstream scientific computation has been limited due to the high memory requirements of the necessary dynamic programming tables and excessive runtimes of sequential implementations. This work addresses both challenges by proposing a set of new parallel algorithms for all steps of a tree decomposition-based approach to solve the maximum weighted independent set problem. A hybrid OpenMP/MPI implementation includes a highly scalable parallel dynamic programming algorithm leveraging the MADNESS task-based runtime, and computational results demonstrate scaling. This work enables a significant expansion of the scale of graphs on which exact solutions to maximum weighted independent set can be obtained, and forms a framework for solving additional graph optimization problems with similar techniques.**

***Keywords:* graph algorithms, dynamic programming, parallel programming, independent set, tree decomposition,**

I. Introduction

Discrete optimization problems on graphs are notoriously difficult to parallelize and finding exact solutions to such optimization problems is often severely limited by the size of the instance. Algorithms based on tree decompositions provide a tantalizing possibility, since the complexity of many NP-hard optimization problems is transformed to become polynomial in the number of vertices in the graph (and exponential in the decomposition's width).

Tree decompositions were introduced by Robertson and Seymour in 1984 [12] as tools in the proof of the Graph Minors Theorem. Each decomposition has an associated measure of width, and the minimal achievable width for a graph is its treewidth. This can be thought of as a measure of how "tree-like" the graph is. The computational community became interested in tree decompositions after it was shown that numerous $\mathcal{NP}$-hard graph problems can be solved in polynomial time on graphs with bounded treewidth [2].

However, nearly all the work assessing the viability of such approaches has been purely theoretical in nature; we are aware of no other attempts to parallelize either the construction of a tree decomposition or the subsequent dynamic programming. Here we present the first serious effort at applying high performance computing (HPC) to this area.

There is a potential for parallelism inherent to tree decomposition-based algorithms: (i) the dynamic programming computation at each node in the tree decomposition requires only partial information about the graph, and (ii) the dynamic programming tables for large sets of tree nodes can be computed independently of one another. While these features of the computation seem to lead to a natural sort of parallelism, exploiting them requires a non-traditional approach when designing a parallel algorithm. Our work leverages the task-based framework offered by MADNESS (Multiresolution Adaptive Numerical Environment for Scientific Simulation) [14] in order to handle the work distribution, load balancing, and asynchronous nature of the dynamic programming. We establish links between algorithms from several disparate communities by using PARMETIS [10] as part of the tree decomposition construction, and the MADNESS runtime to handle the irregular, asynchronous properties of the dynamic programming.

For graphs with large size (which we measure by the number of vertices) and low treewidth, the results are compelling, and we achieve near linear speedups

[*]The submitted manuscript has been authored by a contractor of the U.S. Government under Contract No. DE-AC05-00OR22725. Accordingly, the U.S. Government retains a non-exclusive, royalty-free license to publish or reproduce the published form of this contribution, or allow others to do so, for U.S. Government purposes.

relative to a serial implementation. These new algorithmic enhancements and improvements in implementation efficiency enable us to exactly solve optimization problems on networks that are several orders of magnitude larger (in terms of the number of nodes/edges) than attainable with other methods from the literature.

The algorithms described in this paper have been implemented as part of the open source INDDGO[†]software package [6].

II. Background

II-A Definitions and terminology

Formally, a *graph* $G = (V, E)$ is a set of vertices V and a set of edges E formed by unordered pairs of vertices. All graphs in this paper are assumed to be finite, simple and undirected. We also assume the graphs under consideration are *connected*, since otherwise, the techniques being discussed here can be applied to find solutions for each connected component, which can then be easily combined into a solution for the entire graph.

For a vertex $v \in V$, let $N(v) = \{u : (u, v) \in E\}$ be the *neighbors* of v. We say $H = (W, F)$ is a *subgraph* of $G = (V, E)$, denoted $H \subseteq G$, if both $W \subseteq V$ and $F \subseteq E$. An *induced subgraph* is one that satisfies $(x, y) \in F$ for every pair $x, y \in W$ such that $(x, y) \in E$. We denote the induced subgraph of G with vertices $X \subseteq V$ as $G[X]$.

The last graph-related definition we need is of *chordal* graphs — those where every cycle with more than three vertices has an edge connecting two non-consecutive vertices [3].

A *tree decomposition* of a graph $G = (V, E)$ is a pair (X, T), where $X = \{X_1, \ldots, X_n\}$ is a collection of subsets of V and $T = (I, F)$ is a tree (acyclic graph) with $I = \{1, \ldots, n\}$, satisfying three conditions:

1) $\cup_{i \in I} X_i\}$ is equal to the vertex set V $(i \in I)$,
2) for every edge uv in G, $\{u, v\} \subseteq X_i$ for some $i \in I$, and
3) for every $v \in V$, if X_i and X_j contain v for some $i, j \in I$, then X_k also contains v for all k on the (unique) path in T connecting i and j. In other words, the set of nodes whose associated subsets contain v form a connected sub-tree of T.

Note that we will use the term *vertex* to refer to elements of V and *node* to to refer to elements of I to avoid confusion. The subsets X_i are often referred to as *bags* of vertices. The *width* of a tree decomposition $(X, (I, F))$ is the maximum over $1 \in I$ of $|X_i| - 1$, and the *treewidth* of a graph G, denoted $\tau(G)$, is the minimum width over all valid tree decompositions of G. An *optimal tree decomposition* for a graph G is one with width $\tau(G)$.

An important class of graphs of class used in this paper are the *k-trees*, which are defined recursively. In the smallest case, a clique on $k + 1$ vertices is a k-tree. Otherwise, for $n > k$, a k-tree on $n + 1$ vertices can be constructed from a k-tree H on n vertices by adding a new vertex v adjacent to some set of k vertices which form a clique in H. A k-tree has treewidth exactly k (the bags of the optimal tree decomposition are the cliques of size $k + 1$). We call the set of all subgraphs of k-trees the *partial k-trees*. It easy to see that any partial k-tree has treewidth at most k (one can derive a valid tree decomposition of width k from that of the k-tree which contains it). Furthermore, any graph with treewidth at most k is the subgraph of some k-tree [15]. Thus the set of all graphs with treewidth at most k can be generated by finding all k-trees and their subgraphs, leading us to a simple generator for random graphs of bounded treewidth.

In this paper, randomly generated partial k-trees are denoted with the prefix "pkt" followed by the number of nodes, maximum width, and edge density. For example, pkt.500000.10.80 is a partial k-tree generated by keeping 80% of the edges from a random 10-tree on 500,000 nodes. We may write things like pkt.500000.width.80 to denote a set of graphs that all have 500,000 nodes and 80% of the edges of their parent k-tree, whose treewidth (k) is being varied.

II-B Sequential Algorithms

Before presenting the details of our parallel algorithm and related computational results, we describe the general idea behind a serial algorithm for solving the maximum weighted independent set (MWIS) problem using dynamic programming and discuss the various bottlenecks that one encounters in the serial environment. The general process for solving MWIS using tree decompositions to achieve fixed parameter tractability is shown in Figure 1, and described in detail in [13]. At a high level, after initializing the graph, an elimination ordering is computed and used to guide triangulation (line 4 in Algorithm 1), a process of adding edges to make the graph chordal. It is then easy to compute a tree decomposition for the chordal graph using Gavril's construction routine (lines 7-20 in Algorithm 1).

After construction, the resulting decomposition (X, T) is rooted, and we execute the dynamic programming following a post-order walk on the tree T. At each node, we construct a hash table of partial solutions — for MWIS, this stores each independent set within the bag

[†]Integrated Network Decompositions and Dynamic programming for Graph Optimization

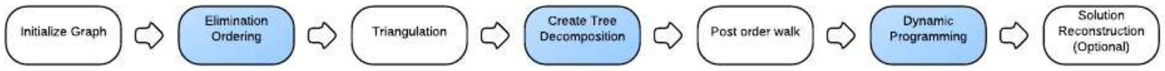


Fig. 1. Solving MWIS using Dynamic Programming and Tree Decompositions

with the weight of the best possible "expansion" of that set into the subtree rooted at that bag. As shown in Algorithm 2, we can reduce memory requirements by storing only the independent sets in the intersection of a bag with that of its parent node (also described in [13]). When the post-order walk terminates at the root, the highest weight in the root node's hash table is the maximum weight of any independent set in the graph G. If the elements in the maximum weighted independent set are desired, extra bookkeeping is required during the computation, and the algorithm must walk back "down" the tree to construct the solution from the hash tables.

Algorithm 1 Gavril's tree decomposition algorithm

```
 1: procedure GAVRIL($G$,$\pi$)
 2: ▷ Graph $G = (V, E)$, $\pi$ a permutation of $V$
 3:     Initialize $T = (X, (I, F))$ with $X = I = F = \emptyset$
 4:     H = TRIANGULATE($\pi, G$);
 5:     $n = |V|$, $k = 1$, $I = \{1\}$, $X_1 = \{\pi_n\}$
 6:     $t[\pi_n] = 1$                    ▷ $t$ is an $n$-long array
 7:     for $i = n - 1$ to 1 do
 8:         $B_i$=GETNEIGHBORS($H, \pi_i, \{\pi_{i+1}, \ldots, \pi_n\}$);
 9:         Find $m = j$ such that $j \leq k$ for all $\pi_k \in B_i$;
10:         if $B_i = X_{t[m]}$ then
11:             $X_{t[m]} = X_{t[m]} \cup \{\pi_i\}$;
12:             $t[\pi_i] = t[m]$;
13:         else
14:             $k = k + 1$;
15:             $I = I \cup \{k\}$; $X_k = B_i \cup \{\pi_i\}$;
16:             $F = F \cup \{k, t[m]\}$;          ▷ update $T$
17:             $t[\pi_i] = k$;
18:         end if
19:     end for
20:     return $T = (X, (I, F))$;
21: end procedure
```

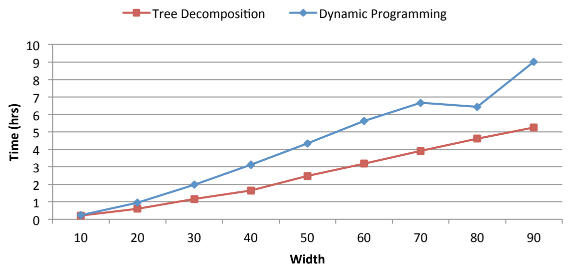
II-C Sequential Implementation Results

In previous work [13], we presented sequential software to construct tree decompositions using various heuristics alongside a fast, memory-efficient dynamic programming implementation for solving MWIS. Although that work showed that algorithms based on these techniques could be practical from a computational standpoint, it also revealed numerous bottlenecks in the computation that limited the scale of the graphs which could be analyzed. Two limiting factors are the amount of memory required to store the dynamic programming tables of partial results from numerous subproblems at multiple tree nodes simultaneously, and the total time required for the generation and subsequent node-by-node analysis of the tree decomposition. We now quickly review the serial performance on a test suite of partial k-trees of various widths and sizes. We consider two metrics: runtime measured in seconds and memory high water mark (HWM) measured in gigabytes (GB), which is the maximum memory in use at any given time during execution. In Figures 2 and 3, we provide separate timing and HWMs for the tree decomposition (TD) and

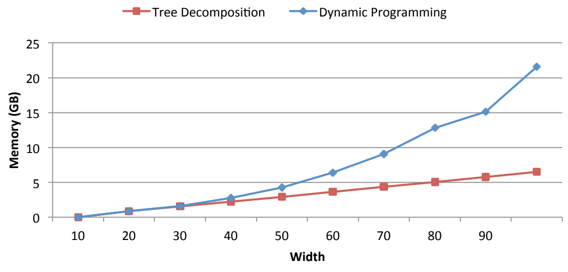
Algorithm 2 Compute a node's hash table

```
 1: procedure COMPUTEDPTABLE($G$, $T$,$k$)
 2: ▷ Graph $G$, a tree decomposition $T$, node $k$
 3:     Let $T = (X, (I, F))$ where $c_1, c_2, \ldots, c_d$ denote
 4:     the children of node $k$ and $p$ the parent
 5:     Let $D_j$ be the DP hash table for node $j$
 6:     with $f_j(s)$ the value of a set $s$ in $D_j$
 7:     $S$ = FINDALLWIS($G[X_k]$)
 8:   ▷ $S$ a set of ordered pairs $(s, w(s))$
 9:     for all $(s, z) \in S$ do
10:         for $i = 1$ to $d$ do
11:             $t_i = s \cap X_{c_i}$
12:             Look up $t_i$ in table $D_{c_i}$: $(t_i, \cdot, f_{c_i}(t_i))$
13:             $z = z + f_{c_i}(t)$
14:         end for
15:         ▷ Subtract the weight of the parent intersection
16:         Let $s_p = s \cap X_p$; $f_k(s_p) = z - w(s_p)$
17:         if $(s_p, \cdot, \cdot) \notin D_k$ then
18:             $D_k = D_k \cup (s_p, s, f_k(s_p))$
19:         else    ▷ The key $s_p$ exists in the hash table
20:             Let $(s_p, s', x)$ be current entry in $D_k$
21:             if $f_k(s_p) > x$ then
22:                 Update $D_k$ to $(s_p, s, f_k(s_p))$
23:             end if
24:         end if
25:     end for
26:     return $D_k$
27: end procedure
```

dynamic programming (DP) steps as the width and size of the graph change. For example, Figure 2 illustrates the increase in runtimes and memory usage when the graph size is fixed and its width increases. We note that for higher width graphs, dynamic programming takes up to twice as much time as tree decomposition, and uses nearly four times the memory.



(a) Time

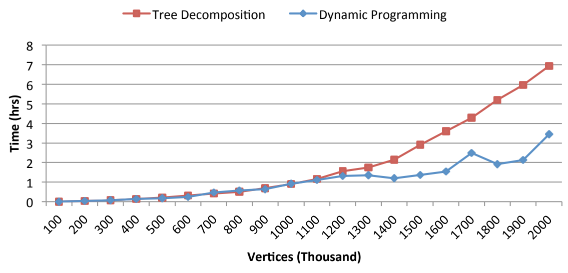


(b) Memory

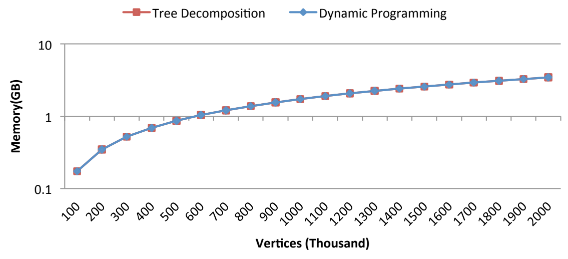
Fig. 2. Runtime and memory variation with the width of graph $pkt.500000.width.80$

Since the real benefit of tree decomposition-based algorithms lies in fast solutions for low-width graphs with a high vertex/edge count, we also include a plot of performance at a fixed width, with increasing graph size, in Figure 3. It is interesting to note that this changes the balance of resources needed for each step of the algorithm drastically. Here, decomposition generation dominates the runtime, and the memory used during dynamic programming is only negligibly higher than that needed for tree construction (hard to see even at a logarithmic scale, as used in Figure 3(b)).

These results indicate that both the tree decomposition generation and the dynamic programming would benefit from a parallel implementation — the former primarily to reduce the runtime, and the latter to provide both acceleration and the distribution of objects across the memory of numerous nodes.



(a) Time



(b) Memory

Fig. 3. Runtime and memory variation with the number of nodes of the graph $pkt.nodes.10.80$

III. Creating tree decompositions in parallel

As discussed in Section II-C, generating the tree decomposition can become a bottleneck as the number of nodes in the graph grows. In order to address this, we developed parallel algorithms for two key steps in the process: (i) finding an elimination ordering, and (ii) generating the bags for the tree decompositions. Finding the bags is computationally intensive since it repeatedly searches for the neighbors of a vertex v which occur after v in the ordering, i.e., the *forward neighbors*. Here we describe our approaches to improving the performance of both on a distributed memory architecture.

III-A Parallel elimination order generation

A close look at the profile of the subroutines needed for tree decomposition, shown at log scale in Figure 4, clearly shows elimination ordering (using the METIS implementation of the multiple minimum degree heuristic) takes roughly 90% of total time. To address this, we integrated routines from the ParMETIS library [8] [9] [10], which provides a distributed fill-reducing ordering using nested dissection techniques. Although this provided a drastic reduction in runtime, the widths of the resulting tree decompositions (shown in Figure 6) were often unacceptably larger than their sequential analogues. We note that more generally, the heuristics employed to find the ordering were originally designed to minimize the number of edges added in the triangulation and offer mixed results with respect to width (see [13]).

Since the complexity of the dynamic programming is exponential in the treewidth, it is critical to devise a parallel ordering routine that limits this inflation. Based on ideas in a paper of Hendrickson et al. [4], we created a two-step procedure shown in Algorithm 3 that uses ParMETIS to partition the nodes into subsets which have known relative positions in the final order, then runs a second fill-reducing heuristic on the subgraph induced by each subset to refine the ordering. We chose to use the AMD algorithm [1] for the second step, based on experimental results in [13]. For computational results and conclusions, please see Section V-A.

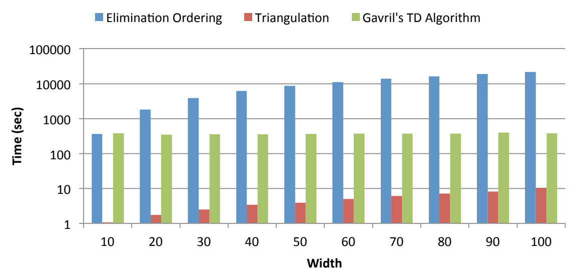


Fig. 4. Breakdown of TD construction time on $pkt.500000.width.80$ graphs

III-B A parallel version of Gavril's algorithm

Once the elimination ordering time has been reduced, Gavril's algorithm (Algorithm 1) dominates the tree decomposition computation. We parallelize this algorithm using a hybrid MPI + *pthreads* approach as described in Algorithm 4. This distributes the work of finding forward neighbors and calculating tree edges across multiple compute nodes, with multiple bags being generated simultaneously on each node. Since the bag $X_{t[m]}$ in line 10 of Algorithm 1 is not locally known in a distributed environment, we postpone the bag unions to a sequential refinement stage, temporarily creating a tree with exactly n nodes. The master process then operates on the full set of n bags and $n-1$ edges, merging appropriate nodes to produce a tree equivalent to the sequential algorithm described in Algorithm 1 (for the same elimination ordering). In Algorithm 5, the subroutine REPLACEPARENT is used to remove redundancy when a parent and child in the unrefined tree have identical bags by essentially removing the child from the tree, making its children now directly inherent from what was their grandparent.

Algorithm 3 Find elimination ordering using ParMETIS (optionally with AMD)

```
 1: procedure ParFindElimOrder(G, useAMD)   ▷ Let $G=(V,E)$, $|V|=n$, and $size/rank$
 2:  be the MPI communicator size and task rank
 3:     $x = n/size, xm = n \mod size$
 4:     Let $vtxdist[]$ be an array of $size+1$, storing
 5:     MPI task displacement, with $vtxdist[1]=0$
 6:     $vtxdist[i] = vtxdist[i-1]+x \quad \forall\, i \in [2, size+1]$
 7:     $V = \cup_{i=1}^{size}\alpha_i;\ \alpha_i = \{v_{ix}, v_{ix+1}, \ldots, v_{x(i+1)-1}\}$
 8:     ▷ Local node neighbor info in $adjncy[]$, $xadj[]$
 9:     $\left.\begin{array}{lcl} adjncy & = & \text{concat}(adjncy, N(v_j)) \\ xadj_{v_j} & = & |adjncy| \end{array}\right] \forall\ v_j \in \alpha_i$
10:     ParMETIS_V3_NodeND($vtxdist$,$adjncy$,$xadj$)
11:     Re-ordered vertices of $i^{th}$ MPI task are in $\alpha'_i$,
12:     collect vertices into root MPI task.
13:     $V' = (\alpha'_1, \ldots, \alpha'_{size})$ with
14:     $\alpha'_i = (v'_{ix}, v'_{ix+1}, \ldots, v'_{x(i+1)-1})$
15:     ▷ Translate ParMETIS indices back to vertices
16:     $V''[v'_i] = v_i\ where\ i \in [1, n]$;
17:     if !useAMD then
18:         return $V''$;
19:     end if
20:     ▷ Redistribute $V''$ among tasks
21:     $V'' = \{v''_1, v''_2, \ldots, v''_n\} = \{\beta_1 \cup \beta_2 \cup \ldots \cup \beta_{size}\}$
22:     $\beta_i = \{v''_{ix}, v''_{ix+1}, \ldots, v''_{x(i+1)-1}\} \quad i \in [1, size]$
23:     $\pi_i = $ FindElimOrder($G[\beta_i], AMD$)
24:     return $\Pi = (\pi_1, \pi_2, \ldots, \pi_{size})$
25: end procedure
```

Algorithm 4 Threaded bag processing

```
 1: procedure ThreadBag(G, Π, prefix, start, end)
 2:     for $i$ = $start$ to $end$ do
 3:         $B_i$=GetNeighbors($G, \pi_i, \{\pi_{i+1},\ldots,\pi_n\}$)
 4:         Find $m = \pi_j$ such that $j \le k$ for all $\pi_k \in B_i$
 5:         $X_k = B_i \cup \{\pi_i\}$
 6:         store ($\pi_i$,$\pi_m$)            ▷ edge to parent bag
 7:         store $X_k$          ▷ content of the current bag
 8:     end for
 9:     send all stored edges and bags to rank 0.
10: end procedure
```

IV. Task-oriented parallel dynamic programming

Having described our novel ideas for parallelizing the construction of tree decompositions, we now move on to our approach for applying distributed computing to the dynamic programming. Our approach uses task-oriented computing, where large operations are divided

Algorithm 5 Parallel tree decomposition

```
 1: procedure ParallelTreeDecomposition
 2:     Let $\Pi$=($\pi_1, \pi_2, \ldots, \pi_n$); $wr, ws$ be MPI rank and communicator size with $nthreads$ pthreads per task.
 3:     $xr = n/ws$, $startpos = wr * xr$
 4:     $tr = xr/nthreads$
 5:     for $i$ = 0 to $nthreads$ do
 6:         $tstart = startpos + i * tr$
 7:         $tend = tstart + i * tr$
 8:         ThreadBag $(G, \Pi, prefix, tstart, tend)$
 9:     end for
10:     Barrier()
11:     Gather tree edges and bags from threads to create a tree decomposition $(T, X)$ with $n$ nodes, $n-1$ edges.
12:     if $wr == 0$ (serial refinement) then
13:         Let $X_i$ be the bag of node $i$, associated with vertex $\pi_i$. Let $Adj_i$ be the neighbors of $i$ in $T$, with $P_i$ the parent node. Let $R$ be root of $T$.
14:         for $i = n-1$ to $1$ do
15:             if $X_i \backslash \pi_i == X_{P_i}$ then
16:                 ReplaceParent(i)
17:             end if
18:         end for
19:     end if
20:     Write $(T', X')$ with $X' = \{X'_1, X'_2, \ldots, X'_m\}$ where $X'_i \in X$ and $Adj'_i \neq \emptyset$
21: end procedure
```

into smaller units called tasks. These tasks are then executed asynchronously across a distributed computer. MADNESS is a fast and accurate environment for computational chemistry, now used in many other fields including nuclear, atomic, and molecular physics. The MADNESS library and parallel runtime provide a versatile scientific computing environment that abstracts the intricacies of task-oriented computing, allowing application developers to focus on algorithm development and implementation. The commonly used tools in MADNESS are implemented on top of its parallel runtime, which provides an API for parallel data structures and functionality required for task-oriented computation. Key features include a) futures to manage dependencies and hide latency; b) global name spaces; c) non-process-centric computing; and d) dynamic load balancing and data redistribution [14].

When solving problems using dynamic programming and tree decompositions, the (rooted) decomposition determines a number of data dependencies in the algorithm. For instance, in Figure 5 (which shows a small part of a tree decomposition with children drawn below their parents) the processing of tree node B cannot finish until computation at all its children (D, E and F) has completed. Therefore we use MADNESS tasks to encapsulate the computation required at each tree node, and define dependencies using futures.

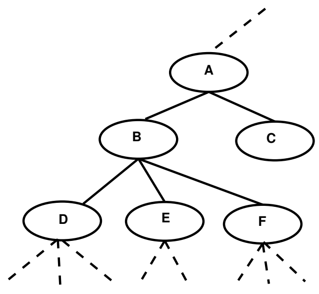


Fig. 5. Example of dependencies in a tree

Important steps in any parallel algorithm for solving the maximum weighted independent set problem using dynamic programming are a) creating a table for storing all independent sets; and b) updating weights of the independent sets at a tree node based on the values from children.

Tree nodes are distributed across tasks using a mapping function provided by the MADNESS parallel runtime. The task owning each node is responsible for populating the local data structures and adding it to a MADNESS distributed container (a built-in distributed hash-map-like data structure). The local data structures are small (e.g. a $w \times w$ bitwise matrix, where w is the bag size), and identical to those populated in the serial implementation; they include the bag intersections with children/parent nodes, list of adjacent tree nodes, and the associated induced subgraph. Since the work for "preparing" each node is independent, we employ OpenMP within the task to speed up this pre-processing phase. All tasks can then access information using a key in the distributed container regardless of the physical location of the data.

Once all nodes are prepared, and the distributed container is populated, we define MADNESS tasks recursively via a pre-order walk starting at the root, using Algorithm 7. At the leaf nodes, full hash tables are computed and returned to the parent node via a special LeafTable routine, similar to Algorithm 5 in [13]. After launching tasks for their children, internal nodes of the tree wait for each child to finish, then incorporate the partial solutions into the current table. The MADNESS task model enables us to implement asynchronous table updates — the parent can update the table with input from each child as it finishes, in an arbitrary order. Some care must be taken to avoid race conditions, but the overall impact is a faster implementation (since the work at a parent can be partially completed prior to receiving updates from its last child). It should be noted that although the tree nodes will likely be processed on different compute nodes, the MADNESS runtime frees the application developer from having to worry about the physical location of each table.

Algorithm 6 Update weights at a node i with values from child C

```
1: procedure UPDATETABLE($i, C$)
2:     Let $W_{X_i}[p]$ denote the weight of an independent set $p \subseteq X_i$
3:     for all independent sets $s \subseteq X_i$ do
4:         $W_{X_i}[s] = W_{X_i}[s] + W_C[s]$
5:     end for
6: end procedure
```

Algorithm 7 Compute hash table for bag X_i

```
1: procedure COMPUTETABLE($i$)
2:     Let node $i$ have bag $X_i$ and neighbors $Adj_i$
3:     if $i$ is a leaf then
4:         return MADTask(LEAFTABLE($i$))
5:     end if
6:     Let $U[], V[]$ be arrays and $j = k = 0$
7:     for $n$ in $Adj_i$ do
8:         $U[j]$ = MADTask(COMPUTETABLE($n$))
9:         $V[k]$ = MADTask(UPDATETABLE($n$, $U[k]$))
10:        $j = j + 1$; $k = k + 1$
11:    end for
12:    Wait for all Futures in $V$ to return
13:    return $X_i$
14: end procedure
```

V. Experimental Results

In the final section of the paper, we describe some computational results that demonstrate the scaling behavior of the parallel procedures described in the previous sections. We partition our scaling results into two parts, first reporting on the performance of the parallel tree construction, then on the dynamic programming. While we do achieve reasonable speedups for the tree construction phase, we believe the scaling of the dynamic programming will dominate overall behavior when using these algorithms to solve large realistic problem instances. As the graph size grows in terms of number of nodes and edges, one can reasonably expect the treewidth of the graph to also increase. While the tree decomposition construction times grow relatively slowly with both the graph size and its width (see Figures 2 and 3, for example), the dynamic programming running time and memory consumption increase exponentially with the width, making it a more significant computational hurdle. Furthermore, the dynamic programming step is more difficult to parallelize due to the asynchronous nature of the computation. While it is often difficult to achieve good scaling results for graph optimization problems and dynamic programming in general, we demonstrate linear (and in some cases even superlinear) speedups for the dynamic programming phase of our optimization algorithm, and consider our parallelization of the dynamic programming to be the major computational contribution of our work.

V-A *Parallel elimination order generation*

To evaluate our algorithms for generating elimination orderings in parallel, we compare both the widths of the resulting decompositions and the required runtime. In Figure 6 we illustrate the differences between k (an upper bound on the treewidth, since the graphs are partial k-trees), and the widths produced by ParMETIS_V3_NodeND (v3) and our hybrid combining ParMETIS with AMD (v3+amd). The partial k-trees were generated with parameters chosen to maintain a constant edge density (e.g. 500K nodes, $k = 10$ has the same edge density as 1.2M nodes, $k = 24$). Using AMD improves the width over ParMETIS alone for graphs with smaller k values, but as the width increases, the advantage is lost. In Figure 7 we look at a second test suite of partial k trees, where we compare both runtimes and width against the sequential heuristic MetMMD.

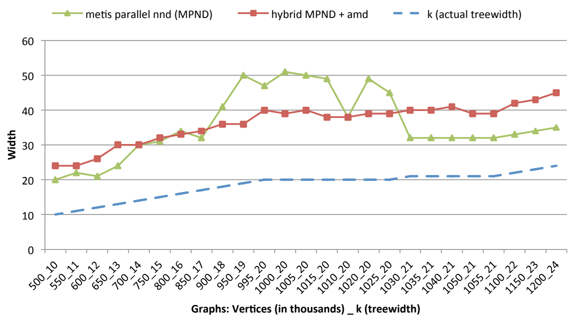
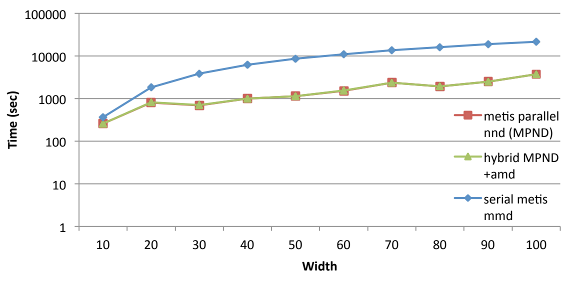
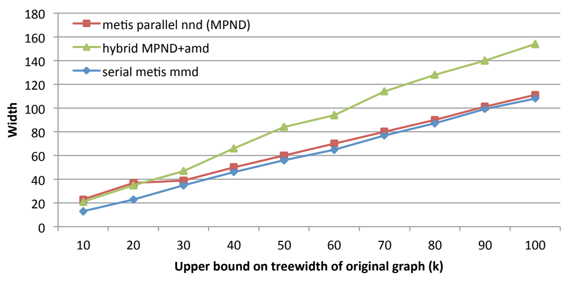


Fig. 6. Widths achieved on k-trees using ParMETIS and ParMETIS + AMD



(a) Time



(b) Width

Fig. 7. Timing for parallel and sequential elimination ordering heuristics on graphs $pkt.500k.width.80$

V-B *Parallel tree decomposition and dynamic programming*

Hüffner, et al., opined that "As a rule of thumb, the typical border of practical feasibility lies somewhere below a treewidth of 20 for the underlying graph" [5]. Prior work using tree decompositions to solve optimization problems was typically limited to graphs less than 5000 nodes, with the techniques described in a recent paper [11] restricting experiments even further — to just several hundred nodes. We report computational results on partial k-trees with five hundred thousand, one million, and two million nodes with $k = 10, 25$, and 50. These represent graphs we believe are of an unprecedented scale - one which was previously deemed impractical.

The sequential results used for comparison here were obtained using one core of a 2.80Ghz Intel Xeon X5560 processor with 8MB of L1 cache and 24GB of memory. Sequential runtimes were limited to a maximum of 24 hours. The parallel experiments were run on a partition of Jaguar [7], a Cray XK6 with one 16-core AMD 6200 series processor and 32GB of memory per node.

V-B1 Parallel tree decomposition: The INDDGO package implements Algorithms 3 through 5 to enable generation of tree decompositions where all steps leverage available parallel resources. Figure 7 shows the time

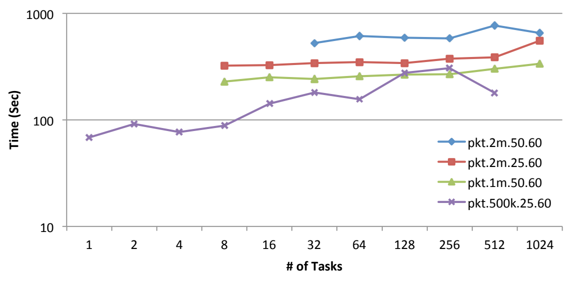


Fig. 8. Parallel tree decomposition runtimes

Graph	Speedup
pkt.500000.25.60	3.02
pkt.1000000.50.60	10.73
pkt.2000000.25.60	70.46
pkt.2000000.50.60	44.00

TABLE I
SPEEDUP WITH 8 TASKS X 4 THREADS FOR ALGORITHM 5

required and widths achieved by parallel elimination orderings relative to a serial heuristic.

In Algorithm 5, our current implementation collects all of the edges and bags created by each compute node (line 11) by writing one file per MPI task. The rank-zero task then performs a concatenation prior to refinement. Unfortunately, this usage of the file system appears to prevent the scaling of our parallel tree decomposition beyond a modest number of MPI processes, as seen in Figure 8. We believe that replacing the file system interactions by a set of local memory stores followed by a global gather would resolve this problem, and is planned as a future improvement to the code.

Despite the delays inherent from file system use, we achieve up to a 70× speedup over the sequential version on just 32 cores (see Table I). This superlinear speedup can be attributed to a new, more efficient version of the FINDNEIGHBORS routine that takes advantage of caching within each task.

V-B2 Parallel dynamic programming: Although the parallelization of the tree decomposition construction does not show compelling scaling results, our dynamic programming implementation shows exceptional, even superlinear scaling. For example on a partial 25-tree with 500,000 nodes, the dynamic programming portion of the sequential algorithm required almost 22 hours (76630 seconds), and the speedup shown in Figure 11 is superlinear. For example 512 tasks with 8 threads each completed the dynamic programming in only 13 seconds!

We first consider the behavior of our algorithms with respect to the shared memory parallelism. Figure 12

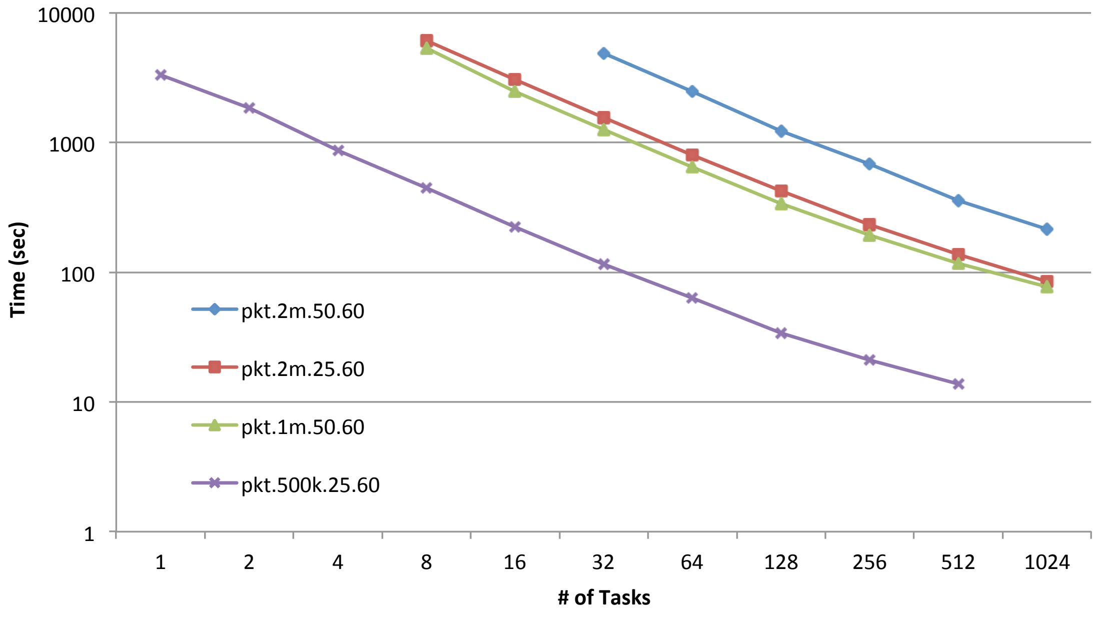


Fig. 9. Dynamic programming runtime

shows the runtime of the dynamic programming algorithm using a fixed number of MADNESS tasks (128) and varying the number of OpenMP threads per task. Threading improves the runtime until we reach 12 threads, where delays from increased memory contention offset the gains from parallel processing. Based on Figure 12, we chose to conduct the remainder of our scaling experiments using 8 OpenMP threads per MADNESS task to optimize performance.

Figures 9 and 10 illustrate the runtime and speedup, respectively, as we vary both the number of tasks and the size of the input problem. We note that all sequential experiments were run with a with a time limit of 24 hours, which prevented almost all graphs with width greater than 10 from completing. In Figure 9, this leads to trend lines that do not extend to the smaller numbers of tasks for graphs on a million or more nodes. To present speedup results, we used the smallest (and thus slowest) parallel run completed in less than 24 hours (the minimum number of tasks depended on the input size) as a baseline for computing our speedups. Figure 10 depicts scaling relative to the appropriate base case by using dashed lines to indicate linear speedup (e.g. Opt-1 is linear relative to performance with 1 task and Opt-32 relative to an initial run with 32 tasks) on the same graphs used in Figure 9. As one can see, when the graph is large enough, the MADNESS tasks were not starved for work and we were able to achieve speedups that are approximately linear. Increases in communication overhead as the number of tasks increases prevent perfect scaling.

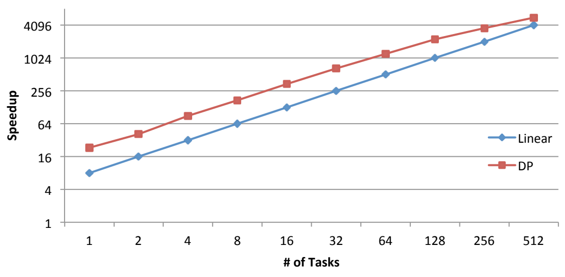


Fig. 11. DP speedup for pkt.500000.25.60

VI. Conclusions and Future Work

In this paper we present what we believe to be the first application of high performance parallel computing to tree decompositions and the related dynamic programming. We propose novel parallel algorithms to find elimination orderings, generate tree decompositions from these orderings, and to perform the memory-intensive dynamic programming using the MADNESS runtime. Our techniques are able to exactly solve MWIS instances

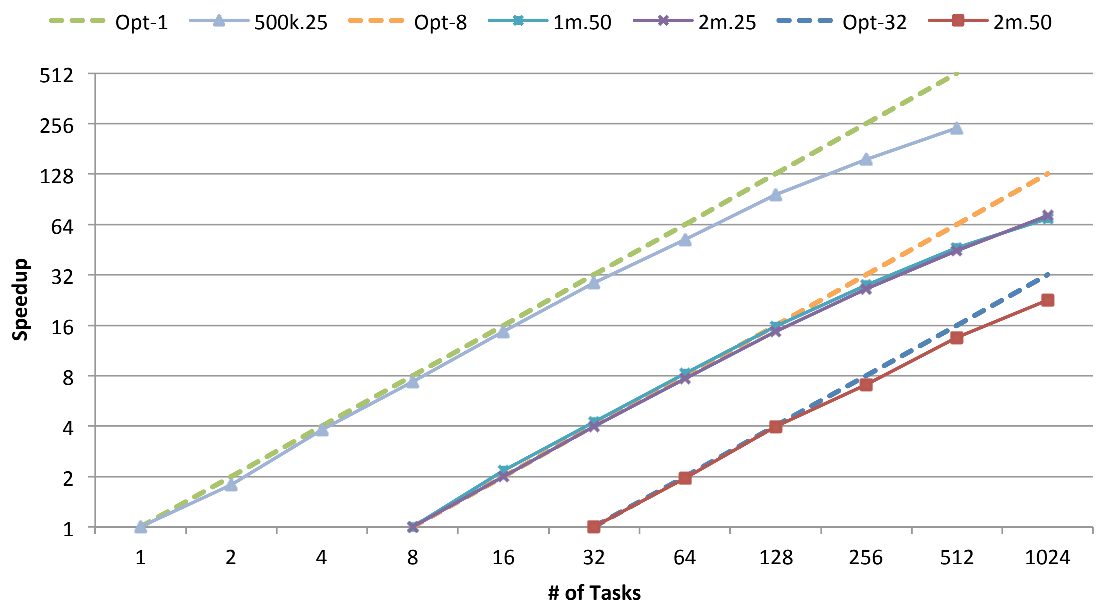


Fig. 10. DP speedup, shown relative to smallest successful parallel execution time

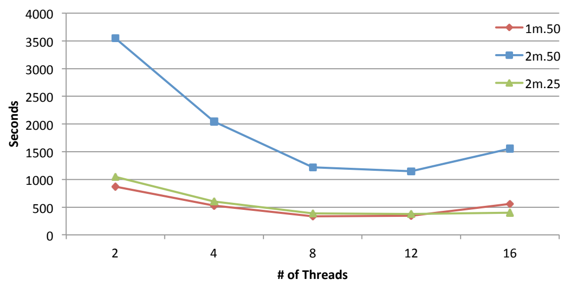


Fig. 12. DP runtime with 128 tasks and varied OpenMP thread count

that are orders of magnitude larger (in terms of number of vertices and edges) than any other problems solved in the literature. Since the scaling of other branch-and-bound based algorithms depends strongly on the number of nodes and edges in the graph, we believe our work presents the only known method to solve problems of this size to optimality.

Our empirical study shows the performance of each algorithm when run on graphs with various widths and sizes, and many cases exhibit excellent scaling. Our latest code is available for other researchers as part of the INDDGO software package [6], and one of our longer term goals is to develop a more general framework for solving graph optimization problems via the type of dynamic programming we discuss in this paper. Such a framework would allow the application developer to write only the problem-specific code for the dynamic programming; everything related to the tree decomposition and the distribution of tasks would be handled by the framework.

VII. Acknowledgements

We would like to acknowledge the help of several ORNL colleagues in making this paper a reality. For their help with the MADNESS parallel runtime, we thank Robert Harrison, Rebecca Hartman-Baker and Benjamin Mintz. We would also like to thank Josh Lothian for his help with getting INDDGO to a releasable state.

This work was performed while all three authors were at Oak Ridge National Laboratory, and was funded by the Department of Energy Office of Science, Office of Advanced Scientific Computing Research under a grant through the Applied Mathematics Program.

References

[1] P.R. Amestoy, T.A. Davis, and I.S. Duff. Algorithm 837: AMD, an approximate minimum degree ordering algorithm. *ACM Transactions on Mathematical Software (TOMS)*, 30(3):381–388, 2004.

[2] Bruno Courcelle. Graph rewriting: An algebraic and logic approach. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics (B)*, pages 193–242. 1990.

[3] F. Gavril. The intersection graphs of subtrees in trees are exactly the chordal graphs. *Journal of Combinatorial Theory, Series B*, 16(1):47–56, 1974.

[4] Bruce Hendrickson and Edward Rothberg. Improving the run time and quality of nested dissection ordering. *SIAM J. Sci. Comput.*, 20(2):468–489, December 1998.

[5] Falk Huffner, Rolf Niedermeier, and Sebastian Wernicke. Developing fixed-parameter algorithms to solve combinatorially explosive biological problems. 453, May 2008.

[6] Integrated network decompositions and dynamic programming for graph optimization (INDDGO). http://www.github.org/bdsullivan/INDDGO.

[7] http://www.olcf.ornl.gov/computing-resources/jaguar/.

[8] G. Karypis and V. Kumar. Parallel multilevel graph partitioning. In *Parallel Processing Symposium, 1996., Proceedings of IPPS'96, The 10th International*, pages 314–319. IEEE, 1996.

[9] G. Karypis and V. Kumar. Parallel multilevel k-way partitioning scheme for irregular graphs. In *Supercomputing, 1996. Proceedings of the 1996 ACM/IEEE Conference on*, pages 35–35. IEEE, 1996.

[10] G. Karypis, K. Schloegel, and V. Kumar. PARMETIS: Parallel graph partitioning and sparse matrix ordering library. *Version 1.0, Dept. of Computer Science, University of Minnesota*, 1997.

[11] Ethan Kim, Adrian Vetta, and Mathieu Blanchette. Clique cover on sparse networks. In *SIAM: Algorithm Engineering & Experiments (ALENEX)*, 2012.

[12] Neil Robertson and Paul D. Seymour. Graph minors III: Planar tree-width. *Journal of Combinatorial Theory, Ser. B*, 36(1):49–64, 1984.

[13] Blair D. Sullivan, Chris Groër, and Dinesh Weerapurage. Software for constructing tree decompositions and solving weighted independent set problems. *ORNL/TM-2012/176*, 2012.

[14] W.S. Thornton, N. Vence, and R. Harrison. Introducing the madness numerical framework for petascale computing.

[15] J. van Leeuwen. Graph algorithms. In *Handbook of Theoretical Computer Science, A: Algorithms and Complexity Theory*. North Holland, 1990.