We believe Reviewer C misunderstood the contributions of our paper. His/her overall assessment states that we present a parallel algorithm for computing the tree decomposition for a graph, and several related criticisms cite papers for this process. This paper includes a parallelization of that algorithm, however we feel the main contribution of the work is a parallel dynamic programming algorithm for solving MWIS after obtaining such a decomposition, since this step dominates the time and memory complexity for large problems. Although we agree that we should have cited several prior works directly instead of referring to our other paper which included the citations, the examples given (Lagergren and Bodlaender et al) differ from our work in two important ways: (1) both earlier works addressed only the problem of forming a tree decomposition in parallel; not that of subsequently performing the dynamic programming, and (2) both made assumptions, such as the availability of O(n) processors, which are unrealistic for handling graphs on the order of millions of nodes, even with a supercomputer.

Additionally, we disagree with the assessment that the dynamic programming is “embarrassingly parallel”, and limited by the depth of rooted tree. The algorithms presented allow partial solutions to be propagated up the tree as soon as the children of a node start completing; it is not a bulk-synchronous algorithm that moves layer by layer.  Different nodes in the tree decomposition can have vastly different bag sizes, and so there is a great deal of variation in the amount of work required to process different nodes in the tree decomposition when doing the dynamic programming. While the tables for the leaf nodes in the tree decomposition can indeed be processed in an embarrassingly parallel fashion, this does not hold at all when one moves up the tree.

Finally, we would like to rebut the statement that “memory should not be an issue in real life.” As width (w) increases, the expected number of entries in the dynamic programming table at a tree node grows like $O(w^{\log w})$ and so memory is indeed the most important parameter that affects the running time of  our algorithm!  Empirically, this can be illustrated by the fact that there are some very small graphs (256 and 512 nodes) that would have impossibly large memory requirements for this type of dynamic programming algorithm.  Both graphs are taken from <http://neilsloane.com/doc/graphs.html>: 1dc.256 has 256 nodes and 1dc.512 has 512 nodes.  For 1dc.256, a good heuristic finds a width 138 decomposition.  The table associated with the largest bag requires approximately 7 trillion entries (an estimate computed using methods outlined in our serial work).  For 1dc.512 (width 313), the table would have approximately 5 septillion entries, requiring approximately 10^14 bytes of memory. Thus, to say that memory is not an issue in real life demonstrates a misunderstanding of the impact of width on tree decomposition-based dynamic programming since these small graphs (in terms of number of nodes) are completely out of reach.

Several reviewers noted that our pseudocode was hard to follow without additional explanation; we were attempting to provide as much information as possible within the page limits, but are happy to incorporate the reviewers’ suggestions on improving the presentation of the algorithms. In a similar vein, we appreciate the suggestion of Reviewer B of including a small example graph and tree decomposition; we prepared such an example, but it ended up being cut due to page restrictions. Once we reduce the amount of space allocated to pseudocode, we can include this, as well as additional information on how the elimination ordering relates to the tree decomposition that is formed.

Direct comparison to branch and bound algorithms (B&B) for MWIS are difficult, as the size of the graphs (in terms of number of nodes and edges) solved in parallel are out of reach for most of the standard B&B codes we are aware of. In our serial work, we found certain types of graphs where the tree decomposition-based dynamic programming was up to 5x faster than the leading mixed integer programming solver, Gurobi. Moreover, our code used a single thread/core while Gurobi used four processing cores. We didn’t feel such a comparison was warranted in the parallel case as we are unaware of any commercial MIP solvers that are able to leverage distributed processing. If the reviewer has a specific code in mind that offers parallel computation on graphs of this magnitude, we would be thrilled to run some comparisons and include them in a revised version.

We would also like to clarify the hybrid nature of the algorithms discussed; we plan to revise to include more consistent terminology (task/processor/core/thread) throughout. Some confusion may have arisen due to the use of two separate paradigms for the tree decomposition construction versus dynamic programming steps of process. For the former, we do in fact use MPI + pthreads, and this does not leverage MADNESS. In the dynamic programming, MADNESS is used for distributing the computation at the bag-level across the machine, and allows for the asynchronous updates of the tables as children complete. The OpenMP code is used for generating the independent sets within a single bag (in a single MADNESS task), and is used in a part of the algorithm that does not rely on futures, which avoids incompatibilities.

*Other notes: we might want to address FINDNEIGHBORS – if what we’re doing wouldn’t work in the sequential environment, we should say so. This ties in with the superlinear speedup claims. It’s also probably important to say a tad more about why it might be interesting to have the ability to generate a TD in parallel even if you got a slightly worse width.*