Aggregated and Filtered Grain Graphs

Ananya Muddukrishna
Norwegian University of Science and Technology
ananyam@idi.ntnu.no

ABSTRACT

Grain graphs simplify OpenMP performance analysis by visualizing performance problems from a predictable programmer perspective. However, grain graphs with tens of thousands of *grains* – task and parallel for-loop chunk instances – take a long time to lay out hierarchically and to navigate. The paper presents aggregation and filtering methods that exploit the grain graph's hierarchical structure and problem-pointing precision to group related grains together into summary nodes and filter out non-problematic grains. Aggregated and filtered grain graphs are not only laid out quicker but also enable programmers to progressively navigate the program structure and converge faster on problematic sections. This further enhances the productivity of grain graph based performance analysis.

1. INTRODUCTION

While it is relatively quick in OpenMP [1] – a standardized, widely-used API – to create working parallel programs, optimizing them for high performance is a struggle for programmers due to poor support in visualizing performance problems. Existing visualizations [2–6] predominantly show program execution from a runtime system or thread centric perspective that is unpredictable and disconnected from the fork-join progression expected by programmers. To cope with the semantic gap, programmers rely on experts or trial-and-error tuning methods for performance.

Grain graphs [7] is a recent visualization method that simplifies OpenMP performance analysis by pin-pointing performance problems from a predictable programmer perspective. Structurally, the grain graph is a directed acyclic graph (DAG) that captures the order of creation and synchronization between *grains* – task and parallel for-loop chunk instances – in the executed program. The grain graph is laid out hierarchically using the *Sugiyama* layout [8] to visualize the fork-join progression of the program expected by programmers. Performance crippling conditions such as work inflation, inadequate parallelism, and low parallelization benefit are directly shown on the grain graph, along with precise links to problem areas in source code, enabling programmers to perform optimizations productively without relying on experts or trial-and-error tuning.

Grain graphs for OpenMP programs that expose abundant, fine-grained parallelism are huge in size containing tens of thousands of grains. The sheer size increases the time to lay the graph out to several minutes, impeding productivity. The long layout time is a direct consequence of the Sugiyama layout algorithm whose worst-case time complexity increases with the size of the graph. Navigating the laid out huge grain graph is also time consuming since programmers cannot understand program structure and performance problems at the first glimpse and are required to attentively zoom and pan to different sections of the visualization, while mentally remembering the characteristics of visited sections. Moreover, a powerful graphics workstation with a large monitor screen is needed to render huge grain graphs responsively.

A promising solution to manage the long layout and navigation times is to reduce the size of huge grain graphs without loss of programmer perspective. Aggregation and filtering are well-known processes to reduce the size of huge graphs while retaining the overall structure [9]. Aggregation reduces the size of a graph by collapsing related graph elements into a single group node that summarizes the attributes of members. Filtering is a process that removes irrelevant graph elements to bring properties of interest into sharper focus. Both aggregation and filtering are typically automatic processes requiring no user interaction.

The paper contributes with aggregation and filtering methods that make huge grain graphs smaller and more focused towards performance problems (Section 3). The aggregation method exploits the hierarchical structure of the grain graph to group related grains together into summary nodes while the filtering counterpart removes non-problematic grains to reduce distractions. Using a simple example, the paper demonstrates that aggregated and filtered grain graphs are quicker to lay out and enable programmers to progressively navigate the program structure and converge faster on problematic sections. This further enhances the productivity of grain graph based performance analysis.

Note: Colors are crucial to appreciate grain graphs. Readers are requested to print the paper in color. Optically magnifying feature-rich figures may be required.

2. BACKGROUND

Grain graphs is a visualization method for OpenMP that connects performance problems to the fork-join program structure at the resolution of *grains* – task and parallel for-loop chunk instances created during execution. Since programmers readily identify with the fork-join program structure in terms of grains, problem diagnosis is simplified. In contrast, existing visualizations complicate diagnosis by resolving performance problems from a runtime system or threads perspective that is unfamiliar and unpredictable to programmers.

A summary of the structure, problem-pointing precision, and prototype workflow of grain graphs are presented next.

2.1 Structure

The grain graph is a DAG where nodes denote grains and runtime system operations, and edges denote control-flow. To maintain a predictable programmer perspective, the grain graph places parent and child grains in close proximity without timing as a placement constraint as shown in Figures 1a-b.

The grain graph is laid out hierarchically using *Sugiyama* layout [8]. The layout removes cycles, places nodes in layers, and prevents edge crossings. These features are essential to depict fork-join control-flow in an uncluttered manner to programmers.

The original Sugiyama layout algorithm employs dummy nodes and edges profusely and has a time complexity of $\mathcal{O}(|V||E|log|E|)$ (where V is the set of vertices, E the set of edges) and a memory complexity $\mathcal{O}(|V||E|)$, making it prohibitive for huge graphs. In practice, implementations employ heuristics such as computation timeouts and relaxing edge crossing constraints to manage the high complexity [9, 10]. An improved Sugiyama layout algorithm by Eiglsperger et al. [10] reduces the number of dummy nodes and decreases the time and memory complexities down to linear limits $\mathcal{O}(|V| + |E|log|E|)$ and $\mathcal{O}(|V| + |E|)$ respectively.

Simple aggregations that reduce the size of the grain graph without loss of programmer perspective are applied to speedup layout time as shown in Figures 1c-d. The aggregations group fork nodes of siblings, fragment nodes of parents, and book-keeping nodes of parallel for-loop chunks (not shown for space reasons). Group nodes contain both summarized and individual performance metrics and properties of members. However, the simple aggregations are inadequate to make the layout time manageable for huge grain graphs, creating the need for a more powerful aggregation method.

2.2 Pin-pointing problems

Properties and performance of grains measured during profiling are added as annotations on graph elements. These include standard metrics such as execution time, critical path, and memory system behavior such as cache miss ratios.

The graph structure is used to derive metrics that point to problems per grain. Examples of derived metrics include work inflation, instantaneous parallelism, parallelization benefit, and load imbalance. Derived metrics are also added as annotations to graph elements.

Some annotations are encoded as visual properties of graph elements for quick identification as shown in Figure 1e. The length of a grain is sized proportional to its execution time. Edges are highlighted in red if they are part of the critical path. Grains with metric values that cross sensible thresholds are highlighted with a color that encodes problem severity. Non-problematic grains are shown dimmed to help programmers focus on problems. For huge graphs, dimming alone is insufficient to make programmers spot problematic grains faster since zooming and panning over non-problematic sections is still required.

2.3 Prototype work-flow

The grain graph visualization method is implemented in a prototype that uses the MIR profiler [11] to profile per-grain performance and properties from OpenMP programs. Profiled data is processed in a post-profiling step using the *igraph* [12] R package to construct the grain graph and derive metrics. The grain graph is stored as a GRAPHML file and viewed on readily-available, large-scale graph viewers such as yEd [13] and Cytoscape [14].

The hierarchical laying out of the grain graph is performed on the graph viewer using its own Sugiyama algorithm implementation. The laying out process repeats every time the graph is reopened since layout results cannot be saved to to the standard GRAPHML format. For huge graphs, the layout process can take several minutes. In yEd and Cytoscape, it is unclear if the implementation of the Sugiyama layout algorithm includes the linear time improvements by Eiglsperger et al. [10]. We assume this as true since one of the authors of the improved algorithm, Markus Eiglsperger, also created yFiles [15], the library used by yEd and Cytoscape for laying out graphs hierarchically.

Visual performance analysis begins once the grain graph is laid out. The grain graph has multiple views with colors encoding a single problem or property per view. Programmers shift views to understand properties or pick problems to tackle. Problematic grains are readily identified since they are highlighted and non-problematic grains dimmed. Clicking on a grain opens up a separate window that shows its performance and properties.

Problems and structure are typically understood at the first glimpse in small grain graphs. However, for huge graphs, programmers are required to zoom and pan attentively to different sections of the graph to understand structure and problems, and remember characteristics of visited sections, as shown in Figure 1f. This process is particularly tiring if problems are spread out and the workstation executing the graph viewer program is loaded beyond capacity with work.

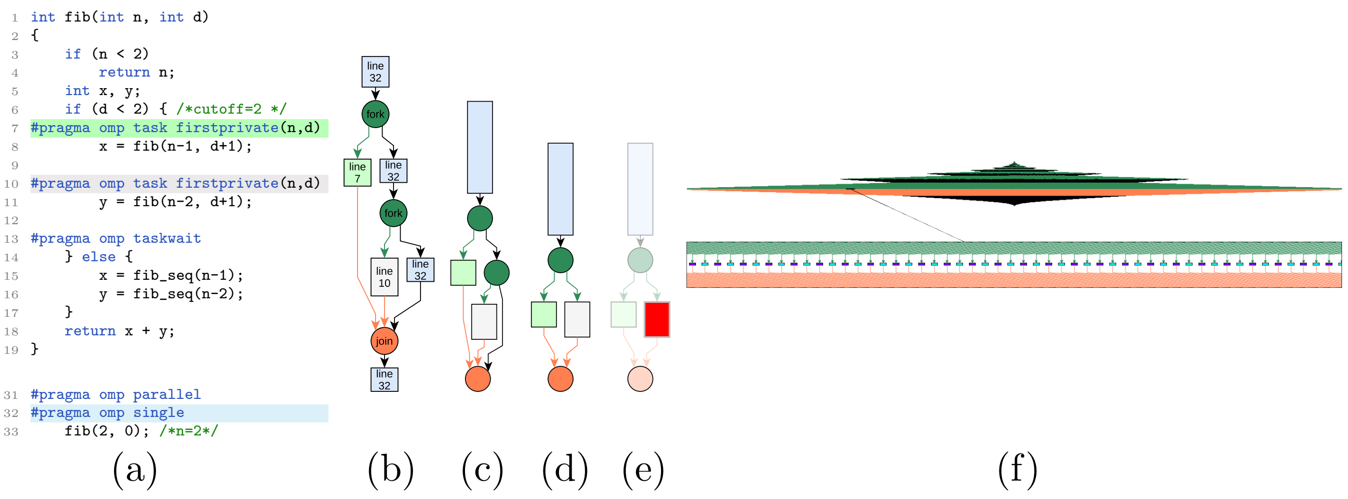
3. AGGREGATION AND FILTERING

The aggregation and filtering methods that overcomes the problem of long layout and navigation times for huge grain graphs are discussed in the section. First, the aggregation method that exploits relationships between grains to group them together is explained. Next, summary performance metrics and properties derived from aggregated group are described. This is followed by a discussion on filtering out non-problematic sections from aggregated grain graphs. Implementation details are summarized in the end.

3.1 Aggregating siblings and families

Siblings and families are natural groups that emerge from the order of grains captured in the grain graph. Siblings are grains that share the same parent and synchronize at the same join point (green and gray grains in Figure 1d). A family is defined by the parent grain and all its children (blue, green, and gray grains in Figure 1d).

The aggregation method exploits the hierarchical relationship between siblings and families as shown in Figure 2a. Child grains are first grouped into sibling groups including the shared fork and join nodes. A lone child is also added to sibling group of size one. Next family groups are created by grouping parent grains and siblings groups. A key observation is that family groups can be treated as child grains eligible for grouping as siblings. This enables siblings and families to be grouped iteratively and hierarchically, without loss of programmer perspective. The size of the grain graph reduces at every aggregation iteration, eventually to



```
1  int fib(int n, int d)
2  {
3      if (n < 2)
4          return n;
5      int x, y;
6      if (d < 2) { /*cutoff=2 */
7  #pragma omp task firstprivate(n,d)
8          x = fib(n-1, d+1);
9
10 #pragma omp task firstprivate(n,d)
11         y = fib(n-2, d+1);
12
13 #pragma omp taskwait
14     } else {
15         x = fib_seq(n-1);
16         y = fib_seq(n-2);
17     }
18     return x + y;
19 }

31 #pragma omp parallel
32 #pragma omp single
33     fib(2, 0); /*n=2*/
```

Figure 1: Grain graph of the task-based Fibonacci program. (a) Source code. (b) Grain graph for small input (n=2, cutoff=2). (c-d) Simple aggregations. (e) Highlighting problems. (f) Huge grain graph (top) for test input (n=45, cutoff=12). Graph has 8192 grains. Inset (below) zooms into graph at magnification 60X.

just one root group node in the end.

The algorithm used to iteratively aggregate siblings and families to produce the aggregated grain graph is not shown for space reasons but explained briefly next. In every iteration, the algorithm begins by grouping all *leaf siblings* into sibling groups. Leaf siblings are siblings that do not have children. Next, family groups are created by grouping parents whose children belong to sibling groups. Family groups are marked as leaf siblings and handed over to the next iteration. Note that parents whose children are not part of sibling groups are not eligible for grouping as a family, and have to wait for a future iteration where all children have been added to sibling groups. With each iteration, new sibling and family groups are created with grains and groups as members. Iterations cease when only a singe root group node remains.

3.2 Group metrics, inferring problems, and navigation

Metrics and properties are derived for groups immediately upon their creation in the aggregation process.

Group metrics are derived by combining performance metrics of member grains and groups using sensible mathematical functions. Addition is used to combine execution time. Memory hierarchy utilization, parallel benefit, and instantaneous parallelism are combined using the minimum function. The maximum function is used to combine load imbalance, work deviation, and scatter. The combinations ensure that the most severe problems of members are transferred to the group.

Group metrics are inferred as problematic if they cross the same sensible thresholds used to infer problems with grains. The thresholds, repeated here for convenience, are memory hierarchy utilization less than two, parallel benefit below one, load balance greater than one, work deviation greater than two, instantaneous parallelism less than the number of cores used to execute the program, and scatter farther than the number of cores in a CPU socket as likely problems.

The aggregated grain graph is navigated progressively by opening/closing group nodes to reveal/hide members, as shown in Figure 2a. Starting with the root group node, programmers understand the structure by successively opening groups until leaf siblings are reached. It is not possible to jump directly to leaf siblings without opening their parent group hierarchy. As a result, the entire structure of a huge graph can be understood only by opening all groups at least once. However, once the structure of a group is known, it can be closed to avoid panning. This also reduces load on the graphics workstation significantly. In addition, the *similarity* metric for groups (discussed next) reduces the time programmers spend in understanding the structure of the graph.

New properties are derived for groups to aid navigation.

Strength is the number of members in a group encoded as a tuple (x, y) where x counts only immediate members and y includes strengths of member groups. The property enables programmers to understand how large a group is without opening it.

Local critical path is the critical path in a group. This may not necessarily overlap the global critical path and is calculated using the execution time of member grains and groups. The property assists programmers to isolate long-running tasks within a group during optimization.

Similarity indicates how similar the structure of a group is in comparison to another group. Similarity is understood based on graph isomorphism decided by a Weisfeiler-Lehman graph kernel [16]. The structure of member groups is considered while deciding similarity. The property enables programmers to understand structure without opening similar groups.

Local critical path and similarity properties are computed on demand during navigation since they are prohibitively expensive to calculate for each group created during aggregation.

Group metrics and properties are added as annotations and encoded as visual properties, similar to grains. Group nodes are shown as rectangles with rounded corners. All

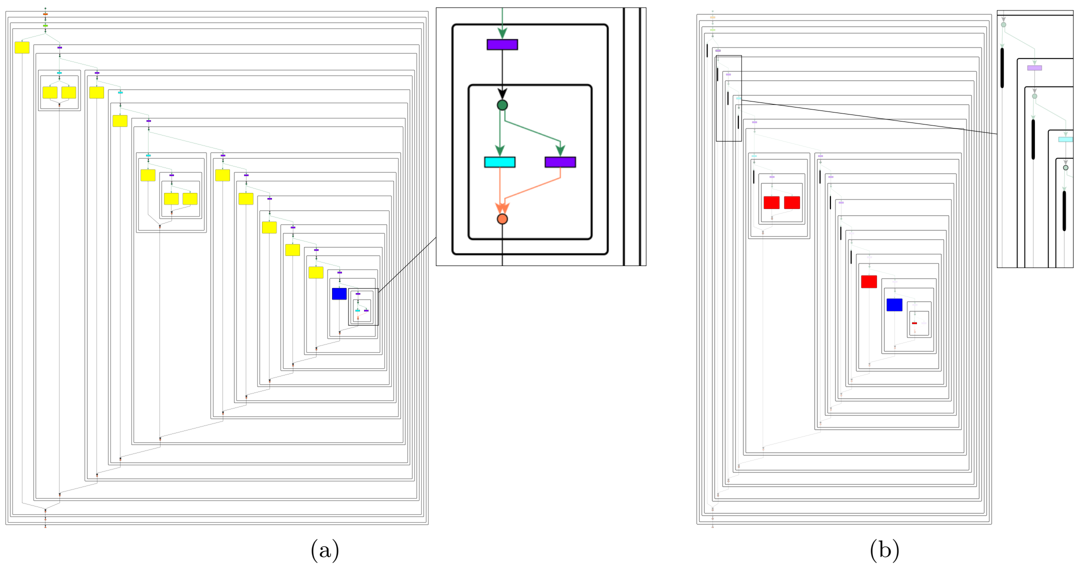


Figure 2: Aggregated and filtered grain graph of Fibonacci program for test input (n=45, cutoff=12) (a) Aggregated graph. Opened groups are colored white and unopened yellow. In the zoomed inset (right), outer group is a family group, inner a sibling group (b) Filtered graph. Non-problematic sections are replaced by fast-forward edges shown thick in the zoomed inset (right). The exact position of the selected group (in blue) is pin-pointed on the aggregated graph to orient programmers.

group nodes are equally sized and their fill color typically encodes the severity of the problem. A *tooltip* (textbox shown when the mouse pointer is hovered on a visual element) in the context of a group node shows its strength. Borders of group nodes on the global critical path are red whereas those on the local critical path are colored orange.

3.3 Filtering non-problematic sections

As mentioned earlier, the grain graph aids programmers to quickly identify problems by dimming non-problematic grains in problem views. However, dimming does not remove the need to pan and zoom huge graphs to identify problematic sections. In addition, dimmed grains equally load the graphics workstation as non-dimmed grains.

Filtering improves over dimming by completely removing non-problematic groups in the aggregated grain graph as shown in Figure 2b. To prevent discontinuity in the graph structure, removed group nodes are replaced with a special edge called a *fast-forward edge*. Successive fast-forward edges are reduced to a single edge. This enables programmers to converge to problematic grains without having to go through non-problematic sections first. The load on the graphics workstation is reduced since non-problematic sections are not rendered.

Filtering changes the structure of the grain graph specific to the problem. As a result, non-problematic groups removed in one problem view can appear in other problem views where they are inferred as problematic. The changing structure can disorient programmers when shifting views to pick out problems to solve. To orient programmers firmly, the location of a selected group in a problem view is highlighted on a separate view that shows the non-filtered but aggregated grain graph, as shown in Figure 2b. The orientation is crucial to maintain the programmers perspective that is core to the grain graphs approach.

3.4 Implementation

The aggregation and filtering methods are implemented using igraph within post-profiling processes of the prototype used to evaluate grain graphs. The similarity property of groups is calculated using a fast, third-party implementation [17] of the Weisfeiler-Lehman graph kernel.

4. RELATED WORK

Review of related work is kept short for space reasons.

To the best of the author's knowledge, none of the DAG visualization methods [18–23] for performance analysis of parallel programs written in OpenMP or similar high-level interfaces provide graph aggregation and filtering to manage huge graphs.

Consult the grain graphs debut paper [7] for limitations and a detailed comparison of grains graphs against related visualization methods.

5. CONCLUSION

The paper presented on-going effort to incorporate aggregation and filtering features in grain graphs. Future work includes refinement and formal analysis of aggregation and filtering methods, more compelling writing, and a thorough evaluation of the efficiency in solving performance problems in standard OpenMP programs.

Acknowledgment

The paper is funded by the TULIPP project, grant number 688403 from the EU Horizon 2020 Research and Innovation programme. The author thanks MCC2016 reviewers for helpful comments, and colleagues Peter A. Jonsson (SICS), Donn Morrison (NTNU), Magnus Jahre (NTNU), Nico Reissmann (NTNU), and Peder Voldnes Langdal (NTNU) for constructive discussions.

6. REFERENCES

[1] L. Dagum and R. Menon, "OpenMP: an industry standard API for shared-memory programming," *Computational Science & Engineering, IEEE*, vol. 5, no. 1, pp. 46–55, 1998.

[2] Intel Corporation, "Intel VTune Amplifier 2013 (document number: 326734-012)," 2013, http://software.intel.com/en-us/vtuneampxe_2013_ug_lin. Accessed 10 April 2015.

[3] M. Geimer, F. Wolf, B. J. Wylie, E. Ábrahám, D. Becker, and B. Mohr, "The Scalasca performance toolset architecture," *Concurrency and Computation: Practice and Experience*, vol. 22, no. 6, pp. 702–719, 2010.

[4] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent, "HPCToolkit: Tools for performance analysis of optimized parallel programs," *Concurrency and Computation: Practice and Experience*, vol. 22, no. 6, pp. 685–701, 2010.

[5] H. Brunst and B. Mohr, "Performance analysis of large-scale OpenMP and hybrid MPI/OpenMP applications with Vampir NG," in *OpenMP Shared Memory Parallel Programming*, ser. LNCS. Springer, 2008, no. 4315, pp. 5–14.

[6] V. Pillet, J. Labarta, T. Cortes, and S. Girona, "Paraver: A tool to visualize and analyze parallel code," in *Proceedings of WoTUG-18: Transputer and occam Developments*, vol. 44, 1995, pp. 17–31.

[7] A. Muddukrishna, P. A. Jonsson, A. Podobas, and M. Brorsson, "Grain Graphs: OpenMP performance analysis made easy," in *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '16. New York, NY, USA: ACM, 2016, pp. 28:1–28:13. [Online]. Available: http://doi.acm.org/10.1145/2851141.2851156

[8] K. Sugiyama, S. Tagawa, and M. Toda, "Methods for visual understanding of hierarchical system structures," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 11, no. 2, pp. 109–125, 1981.

[9] T. Von Landesberger, A. Kuijper, T. Schreck, J. Kohlhammer, J. J. van Wijk, J.-D. Fekete, and D. W. Fellner, "Visual analysis of large graphs: state-of-the-art and future research challenges," in *Computer graphics forum*, vol. 30, no. 6. Wiley Online Library, 2011, pp. 1719–1749.

[10] M. Eiglsperger, M. Siebenhaller, and M. Kaufmann, "An efficient implementation of Sugiyama's algorithm for layered graph drawing," in *International Symposium on Graph Drawing*. Springer, 2004, pp. 155–166.

[11] A. Muddukrishna, P. A. Jonsson, and M. Brorsson, "Characterizing task-based OpenMP programs," *PLoS ONE*, vol. 10, no. 4, p. e0123545, 2015.

[12] G. Csardi and T. Nepusz, "The igraph software package for complex network research," *InterJournal*, vol. Complex Systems, p. 1695, 2006.

[13] yWorks GmBh, "yEd graph editor," 2015, http://www.yworks.com/en/products_yed_about.html. Accessed 10 April 2015.

[14] M. E. Smoot, K. Ono, J. Ruscheinski, P.-L. Wang, and T. Ideker, "Cytoscape 2.8: new features for data integration and network visualization," *Bioinformatics*, vol. 27, no. 3, pp. 431–432, 2011.

[15] R. Wiese, M. Eiglsperger, and M. Kaufmann, *yFiles: Visualization and Automatic Layout of Graphs*. Springer Berlin Heidelberg, Jan. 2002.

[16] N. Shervashidze, P. Schweitzer, E. J. van Leeuwen, K. Mehlhorn, and K. M. Borgwardt, "Weisfeiler-Lehman graph kernels," *Journal of Machine Learning Research*, vol. 12, no. Sep, pp. 2539–2561, 2011.

[17] "Mahito-sugiyama/graph-kernels," https://github.com/mahito-sugiyama/graph-kernels.

[18] S. Brinkmann, J. Gracia, and C. Niethammer, "Task debugging with TEMANEJO," in *Tools for High Performance Computing 2012*. Springer, 2013, pp. 13–21.

[19] Barcelona Supercomputing Center, "OmpSs task dependency graph," 2013, http://pm.bsc.es/ompss-docs/user-guide/run-programs-plugin-instrument-tdg.html. Accessed 10 April 2015.

[20] V. Subotic, S. Brinkmann, V. Marjanovic, R. M. Badia, J. Gracia, C. Niethammer, E. Ayguade, J. Labarta, and M. Valero, "Programmability and portability for exascale: Top down programming methodology and tools with StarSs," *Journal of Computational Science*, vol. 4, no. 6, pp. 450 – 456, 2013.

[21] V. Tovinkere and M. Voss, "Flow graph designer: A tool for designing and analyzing Intel® threading building blocks flow graphs," in *ICPP Workshops*. IEEE Computer Society, 2014, pp. 149–158.

[22] A. Huynh, D. Thain, M. Pericàs, and K. Taura, "DAGViz: a DAG visualization tool for analyzing task-parallel program traces," in *Proceedings of the 2nd Workshop on Visual Performance Analysis*. ACM, 2015, p. 3.

[23] B. Haugen, S. Richmond, J. Kurzak, C. A. Steed, and J. Dongarra, "Visualizing Execution Traces with Task Dependencies," in *Proceedings of the 2Nd Workshop on Visual Performance Analysis*, ser. VPA '15. New York, NY, USA: ACM, 2015, pp. 2:1–2:8.