Towards Aggregated Grain Graphs

Nico Reissmann
Norwegian University of Science and Technology
nico.reissmann@ntnu.no

Magnus Jahre
Norwegian University of Science and Technology
magnus.jahre@ntnu.no

Ananya Muddukrishna
Norwegian University of Science and Technology
ananya.muddukrishna@ntnu.no

ABSTRACT

Grain graphs simplify OpenMP performance analysis by visualizing performance problems from a fork-join perspective that is familiar to programmers. However, it is tedious to navigate and diagnose problems in large grain graphs with thousands of task and parallel for-loop chunk instances. We present an aggregation method that matches recurring patterns in grain graphs and groups related nodes together, reducing graphs of any size to one root group. The aggregated grain graph is then navigated by progressively uncovering groups and analyzing only those groups that have problems. This enhances productivity by enabling programmers to understand program structure and problems in large grain graphs with less effort than before.

CCS CONCEPTS

• **Human-centered computing** → **Graph drawings**; • **Computing methodologies** → *Parallel programming languages*;

KEYWORDS

performance visualization; OpenMP; grain graphs

ACM Reference format:
Nico Reissmann, Magnus Jahre, and Ananya Muddukrishna. 2017. Towards Aggregated Grain Graphs. In *Proceedings of Fourth International Workshop on Visual Performance Analysis (VPA), Denver, Colorado, USA, November 17, 2017,* 8 pages.
https://doi.org/DOI-NA

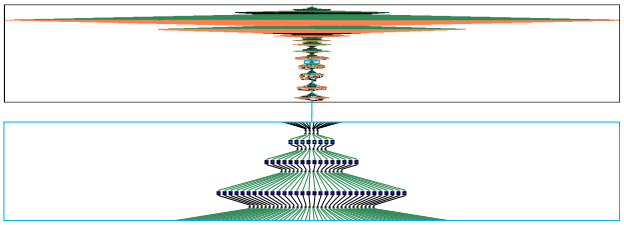


Figure 1: Grain graph of the task-based Sort program from the Barcelona OpenMP Task Suite (BOTS) for large input (`n=20971520, cutoffs={65536,8192,128}`) is dense with 11059 grains. Inset (bottom) zooms into a section at magnification 40X.

1 INTRODUCTION

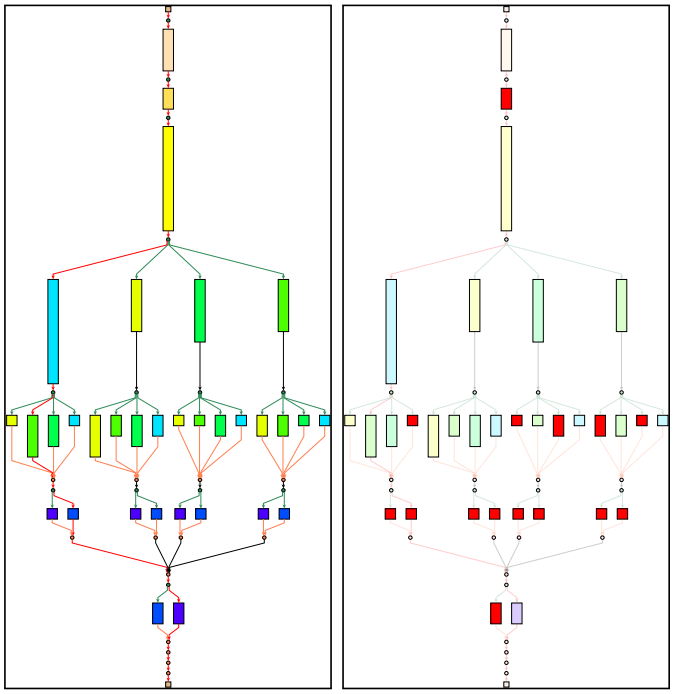
The *grain graph* [19] is a recent visualization method that simplifies OpenMP performance analysis by visualizing problems of task and parallel for-loop chunk instances, collectively called *grains*, from a fork-join perspective. Grains that suffer crippling performance problems such as work inflation, inadequate parallelism, and low parallelization benefit are pin-pointed on the grain graph along with precise links to problem areas in source code. This enables programmers to perform optimizations productively without relying on experts or trial-and-error tuning.

Large grain graphs with thousands of grains (Figure 1) are typical of OpenMP programs that expose abundant, fine-grained parallelism. The high degree of parallelism ensures scalability on large machines but requires low-overhead, locality-aware scheduling [17, 21, 30]. Scalability problems that occur when the scheduling requirement is not met are pin-pointed on the grain graph using metrics that indicate low parallelization benefit, work inflation, and poor memory hierarchy utilization.

Diagnosing problems in large grain graphs requires tedious inspection. Programmers have to zoom and pan attentively to different sections while remembering characteristics of visited sections (Figure 1 inset) . Problems that are spread out become difficult to locate. Non-problematic grains that are shown dimmed to increase focus on problems combine at lower zoom levels and become pronounced. Programmers can perceive the dimming effect and spot problematic grains only when zoomed into higher levels. A powerful workstation with a large screen and copious amount of main memory is required by the graph viewer program to render large grain graphs responsively. In light of these demands, programmers prefer to pore over text summaries and tabular formats of large graphs and reserve the visual approach only for small graphs with a few hundred grains since they are analyzed quickly with little or no navigation.

This paper contributes with a new aggregation method that makes visual analysis of large grain graphs practical. The aggregation method (Section 3) groups related nodes by matching recurrent patterns in the grain graph. This results in an aggregated graph with a single group node. Programmers navigate the aggregated graph by progressively opening and closing groups. Groups with problems are highlighted and non-problematic sections are removed from sight for distraction-free diagnosis. Using standard OpenMP examples, we demonstrate (Sections 3 and 4) that aggregated grain graphs enable programmers to understand program structure and diagnose problematic sections with less effort than what is required for unaggregated graphs. This further enhances the productivity of performance analysis using grain graphs.

Fourth International Workshop on Visual Performance Analysis (VPA), November 17, 2017, Denver, Colorado, USA
Year-NA. ACM ISBN ISBN-NA…$Price-NA
https://doi.org/DOI-NA



(a) (b)

Figure 2: Grain graph of the task-based Sort program from BOTS for small input (`n=512, cutoffs={256,64,16}`). (a) Graph contains 33 grains. Parent and child grains are placed close together using the Sugiyama layout. (b) Grains with low parallel benefit highlighted with a superimposed red color in a separate view are easily understood at first glimpse.

2 BACKGROUND ON GRAIN GRAPHS

The grain graph [19] is a visualization 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 based on timeliness and call graphs complicate diagnosis by connecting performance problems to scheduling events that are unfamiliar and unpredictable to programmers [12, 19].

2.1 Structure

The grain graph is a directed acyclic graph (DAG) whose nodes denote grains and runtime system operations, and edges denote control-flow. Parent and child grains are shown in close proximity on the graph without timing as a placement constraint to maintain the fork-join perspective that programmers are familiar with (Figure 2a[1]). The timing-independent placement is also called *logical-time* placement [6, 12].

The grain graph is laid out using a hierarchical layout called the *Sugiyama* layout [26]. This layout places nodes in layers, removes cycles, and prevents edge crossings. These features are essential to depict fork-join progression in an uncluttered manner to programmers. Practical implementations of the Sugiyama layout algorithm have time and space complexity of $O((|V| + |E|)log|E|)$ and $O(|V| + |E|)$, respectively, where V is the set of vertices and E the set of edges in the graph [8].

[1]Colors are crucial to appreciate grain graphs. Readers are requested to print the paper in color.

2.2 Diagnosing problems

Performance metrics of grains measured during profiling and derived post profiling are added as annotations to the grain graph. The profiled metrics include execution time, cache miss ratio, memory latency, and timestamps of control-flow events such as grain creation and synchronization. These are used to compute derived metrics such as critical path, work deviation, instantaneous parallelism, memory hierarchy utilization, scatter, load balance, and parallel benefit.

Parallel benefit is a custom metric used in several discussions in the paper. The metric is equal to a grain's execution time divided by its parallelization cost including creation time. This aids inlining and cutoff decisions by quantifying whether parallelization is beneficial. Grains with low parallel benefit should be executed sequentially to reduce overhead.

Commonly sought out metrics are encoded visually for quick identification on the graph (Figure 2a). The length of a grain is set proportional to its execution time. Grain colors denote source code locations by default. Edges are colored by type and highlighted red if they are on the critical path.

Grains with metric values that cross sensible thresholds are inferred as problematic and highlighted with a superimposed color that encodes problem severity in a separate view (Figure 2b). Programmers can refine the thresholds if required. Non-problematic grains are shown dimmed to help programmers focus on problems. Problems are also summarized in a separate text file and highlighted in the tabular form of the grain graph.

Diagnosis begins once the grain graph is laid out in Sugiyama style by the graph viewer program. The grain graph has multiple conceptual 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 properties and performance metrics. Figures 2b-a show the programmer cycling between the parallel benefit problem view and the structural view where no problems are highlighted.

3 GRAIN GRAPH AGGREGATION METHOD

In this section, we present the aggregation method that shrinks large grain graphs using pattern matching and enables programmers to understand graph structure and problems progressively by opening and closing groups.

Conceptually, the aggregated grain graph is produced in four consecutive phases called *reduction, normalization, propagation*, and *separation*. We explain these phases next, and discuss navigation of the aggregated graph at the end.

Reduction: This phase matches two recurring patterns called *fork-join* and *linear* patterns and replaces them with group

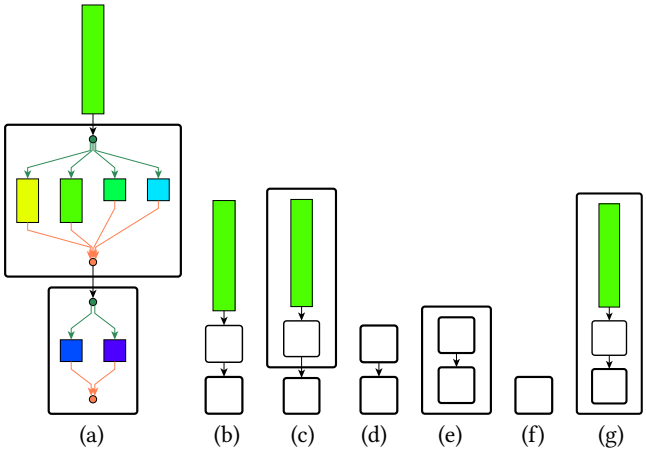


Figure 3: Reduction and normalization. (a) Example graph taken from the subgraph of the rightmost child of the third grain in Figure 2a. (b) After two fork-join pattern reductions. (c-f) Linear pattern reductions leading to a single group node. (g) Example graph after normalization.

```
1  bool is_graingroup(Node n) {
2    return is_grain(n) || is_forkjoin(n) || is_linear(n)
3  }
4
5  void try_reduce(Node n) {
6    if (is_graingroup(n) && is_graingroup(succ(n))) {
7      n′ ← reduce_linear(n)
8      try_reduce(n′)
9    } else if (is_graingroup(n) && is_fork(succ(n))) {
10     try_reduce(succ(n))
11     n′ ← reduce_linear(n)
12     try_reduce(n′)
13   } else if (is_fork(n)) {
14     forall s in succ(n)
15       try_reduce(s)
16     n′ ← reduce_forkjoin(n)
17     try_reduce(n′)
18   }
19 }
```

Listing 1: Pseudocode of the reduction algorithm.

nodes. The fork-join pattern consists of a single fork node connected to child grains or groups, which in turn are connected to a join node (Figure 3a). The linear pattern has two nodes, either a grain or a group node, that are connected to each other (Figure 3c). When matched, the fork-join and linear patterns are replaced by group nodes called *fork-join* and *linear*, respectively.

The repeated matching and replacement of the two patterns reduces the grain graph to a single group node as illustrated in Figures 3a-f.

Listing 1 shows the pseudocode of the reduction algorithm. The code recursively reduces the grain graph by matching fork-join and linear patterns as explained below:

- Line 6 in the pseudocode matches the linear pattern (Figure 3c-d). It uses the helper function *is_graingroup* to detect whether a node and its successor is a grain or a group, and reduces the pattern to a linear group node. Reduction continues with the newly-created linear group node.
- Line 9 matches a grain or group node with a fork node as successor. The matched fork node is recursively aggregated to a fork-join group node (Figure 3a-b). The resulting linear pattern is then reduced to a linear group node. Reduction continues with the linear group node.
- Line 13 matches a fork node (Figure 3a). Upon a match, it recursively aggregates all successors of the fork node. The resulting fork-join pattern is then reduced to a fork-join group node. Reduction continues with the fork-join group node.

Reduction greedily reduces the grain graph by always continuing with the newly-created group node after a pattern match. It does not traverse past a join node. This ensures that the innermost fork-join pattern in a nest is reduced first. Reduction builds a tree of group and grain nodes called the *aggregation tree*. The aggregation tree explicitly captures the fork-join structure and nesting of a grain graph. Its leaves are grains and its intermediate nodes are the newly-created group nodes. Linear group nodes have the two nodes that were matched in the corresponding pattern as children, while fork-join group nodes have the children of the matched fork node as children. The aggregation tree is used as the main operational data structure in later phases to simplify processing.

The reduction algorithm is applicable to grain graphs where parents synchronize with all their children before completion. This essential *completion property* ensures that fork-join patterns are properly nested, permitting their reduction in a hierarchy of group nodes. The completion property holds for well-behaved OpenMP 3.X programs. However, the `taskgroup` construct from the recent 4.0 version of the OpenMP standard permits parents to synchronize with their children and descendants in one step. This violates the completion property and makes reduction inapplicable unless the grain graph is restructured so that all descendants are placed as immediate children of the root parent.

Normalization: This phase transforms the aggregation tree into a canonical form by flattening nested linear group nodes. In the reduction phase, a linear group node is always created for a pair of grain or group nodes, even if more nodes are chained together. This constructs nested linear subtrees where linear group nodes are the children of other linear group nodes as exemplified in Figures 3b-f. Normalization flattens these subtrees to a single linear group node with all non-linear group nodes from the subtree as its children. The result of normalization for the example graph in Figure 3b is shown in Figure 3g.

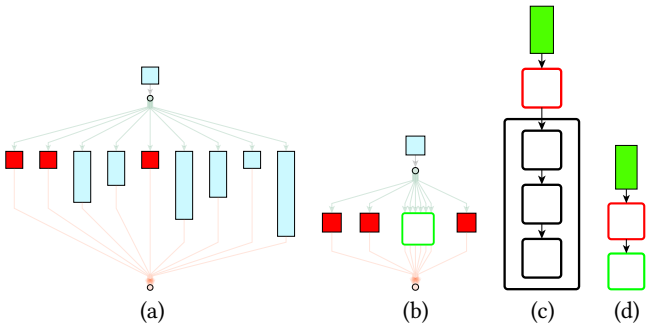


Figure 4: Separation of problematic and non-problematic nodes. (a-b) Fork-join node separation. (c-d) Linear node separation.

Propagation: After normalization, the metrics of child grains and groups are propagated to the enclosing group, all the way up to the root of the aggregation tree. This is accomplished by traversing the aggregation tree in post-order and attributing sensibly-combined metrics of children to the parent group. For example, the *work* metric of a parent group is set to the sum of the execution time of its children.
Metrics are attributed such that problems are propagated to the root group. If a child is problematic, then the parent is marked as problematic as well. The minimum of the memory hierarchy utilization, parallel benefit, and instantaneous parallelism as well as the maximum of the load balance, work deviation, and scatter metrics of children are attributed to the parent group.

Separation: This phase groups non-problematic nodes to separate them from problematic nodes, enabling programmers to focus on problems and reduces load on the graph viewer program. For example, consider a fork-join group that encloses a thousand grains among which only a single grain is problematic. An unseparated graph would require all grains to be rendered. In the separated graph, all the non-problematic children are grouped. As a result, only two nodes need to be rendered – the problematic grain and the non-problematic group node.
Separation traverses the aggregation tree in post-order and separates subtrees rooted at fork-join and linear nodes. In a fork-join separation, a new group node that encloses all non-problematic children of the fork-join node is created (Figures 4a-b). In a linear node separation, a new linear group node that encloses consecutive non-problematic children is created (Figures 4c-d).
After the separation phase, the aggregation tree is converted back to the grain graph where problematic subgraphs of each group are exposed and non-problematic subgraphs are hidden.

Navigation: Starting with the root group, the aggregated grain graph is navigated progressively by opening group nodes to understand structure and problems, and closing them when done (Figure 5). Since only a subset of grains in the graph are laid out during the process, cognitive load on programmers and resource requirements of the graph viewer program are reduced.
Navigation is sped up using several optimizations:

(1) Groups can be opened completely to show all grains including those inside subgroups or drilled down gradually (Figures 5a-d) to a specific group or depth level.
(2) Group nodes are drawn as rounded rectangles with no fill-color to differentiate from grains. Group metrics are shown in a separate property window, similar to grains. Opened groups grow as large as required to envelop members whereas closed group nodes have a constant size. The borders of problematic closed groups are colored red to draw programmer attention. Similarly, borders of non-problematic groups are colored green for quick identification. Our choices for group colors and sizes allow programmers already familiar with grain graphs to smoothly transit to the aggregation feature.
(3) Once a group's structure is known, other groups with similar structure can be navigated confidently or skipped if problem-free. For example, 12 groups in Figure 5d have the same structure. Similarity of groups is computed on-demand using graph isomorphism decided by a Weisfeiler-Lehman graph kernel [23]. The metric also finds its use in comparing groups across runs to detect structural changes. For example, grain graphs of search-based programs such as Floorplan from BOTS change structure based on the number of allocated threads. This can be detected using the similarity metric.
(4) Groups on the global critical path are inspected first since they are good optimization candidates (Figure 5e). Similarly, the *local critical path* of groups not on the global critical path can be computed on-demand and used for prioritizing inspection.

4 PROTOTYPE IMPLEMENTATION

The grain graph visualization is implemented in a reference prototype [20] that produces grain graphs in the GRAPHML format by processing grain profiling data from OMPT extensions [15] or the MIR runtime system [16–18]. Grain graphs are viewed on off-the-shelf, large-scale graph viewer programs, such as yEd [31] and Cytoscape [24].

We extended the reference prototype for grain graphs to produce aggregated graphs in GRAPHML upon programmer request. The aggregation method was implemented in C++, leveraging support for nested groups [3] in the GRAPHML standard and using the igraph [5] library for basic graph processing. Our prototype implementation of the aggregation method is available on GitHub for review [22].

We used yEd to our best effort to view aggregated grain graphs. At present, yEd is the only viewer with sufficiently mature support for GRAPHML files with nested groups. yEd has features to interactively open and close groups, and jump to groups at any level of the hierarchy. The property editor dialog in yEd shows annotations of nodes. Cycling between problem and structural views was achieved

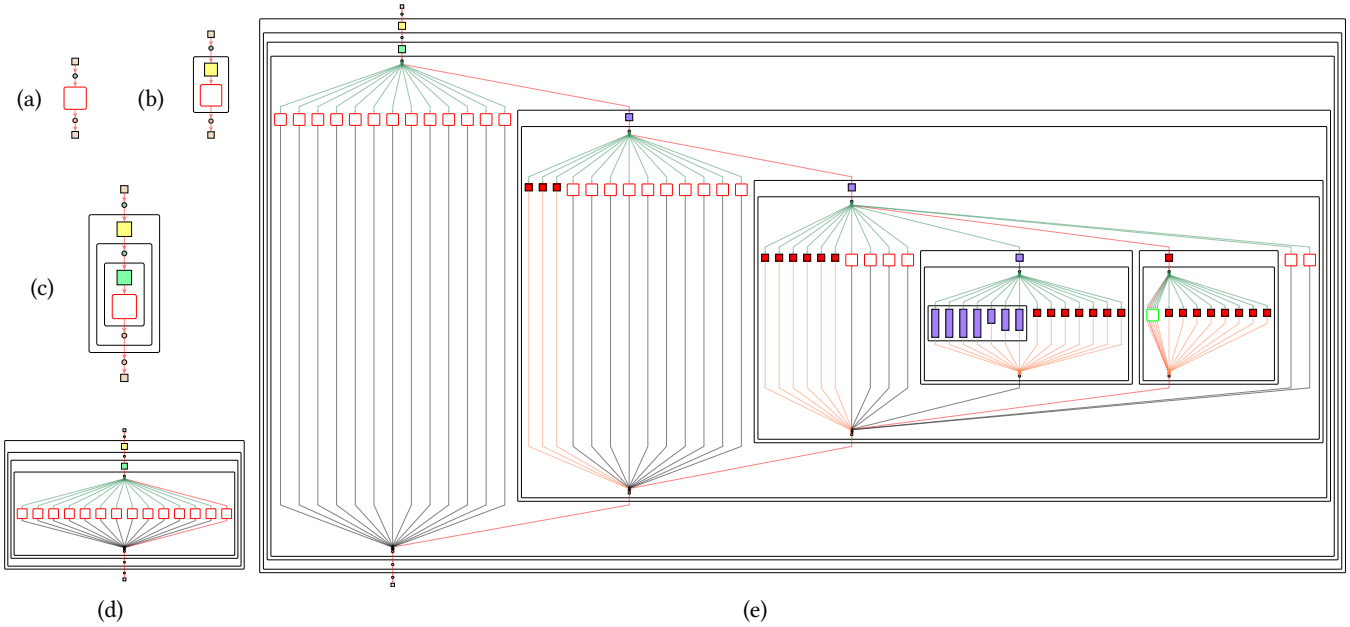


Figure 5: Navigating the aggregated grain graph of NQueens program from BOTS for input (`n=14, cutoff=4`) that exposes fine-grained parallelism. The graph has 21492 grains and 3073 group nodes. Grains with low parallelization benefit are highlighted as problems. (a-d) Drilling down to sibling groups at a depth of 3 from the root group. (a) Root group. (b) At depth 1. (c) At depth 2. (d) At depth 3. Red borders show that all groups are problematic. Programmers can quickly understand that 12 out of 14 groups are structurally similar to each other using the similarity metric. (e) Drilling down along the critical path to sibling groups at the lowest depth. The green-bordered group inside the sibling group hides non-problematic grains. This help programmers focus on problems. The non-problematic group in the adjacent group is distracting since it is opened despite being problem-free.

by switching to tabs of opened GRAPHML files. Each opened file highlighted a problem. External programs parameterized by group names were used to compute on-demand metrics – local critical path and similarity. These programs do not update the visualization and programmers are required to manually load their output into yEd. Similarity was computed using a fast, third-party implementation of the Weisfeiler-Lehman graph kernel [27]. We recognize that our aggregated graph interactions have quite some room for improvement and are designing a dedicated viewer for grain graphs as part of future work.

5 EVALUATION

We tested our aggregation prototype on grain graphs of C/C++ benchmarks from SPEC OMP 2012 (SPEC-OMP12), Barcelona OpenMP Task Suite v2.1.2 (BOTS) and Parsec v3.0 (Parsec). The benchmarks were compiled with MIR-linked GCC v4.4.7 and profiled on an 48-core test machine with 64GB memory and four 2.1GHz AMD Opteron 6172 processors with frequency scaling disabled. Input values that exposed abundant, fine-grained parallelism were provided to obtain large grain graphs (Table 1).

We use the metric *visible node count*, symbolized as θ, to judge the ability of the aggregation method to reduce programmer effort in navigating and diagnosing problems in large grain graphs. Visible node count is defined as the minimum number of nodes that are visible in the grain graph while diagnosing a problematic grain. When visible node count is small, cognitive load on programmers is reduced and fewer resources are consumed by the graph viewer program.

The visible node count for a problematic grain in an aggregated grain graph is the number of nodes exposed by opening groups in the path leading to the grain. In contrast, the visible node count in an unaggregated grain graph is equal to the number of nodes in the entire graph irrespective of the position of the problematic grain.

Table 1 shows the maximum visible node count for the evaluation benchmarks under two cases. The first is a conservative case that assumes all grains in the graph are problematic. Maximum visible node count for this case is denoted $max(\theta_c)$. The second case considers an actual problem – low parallel benefit. Maximum visible node count for the second case is denoted $max(\theta_{pb})$. For both cases, the reduction in maximum visible node count compared to the total size of the graph, *i.e.*, the maximum visible node count for the unaggregated graph, is reported as *Savings*.

For the conservative first case, we see a large reduction in visible node count. On average, the savings is 95.98%. The biggest savings is 99.97% for the Strassen benchmark and the smallest savings is 81.57% for Freqmine. This shows aggregation can significantly reduce the visible node count for any problematic grain in our evaluation setup.

For the case of low parallel benefit, we see a further reduction in visible node count since grains that are non-problematic are

Table 1: Benefit of aggregation for standard OpenMP benchmarks is measured using reduction in number of visible nodes during problem diagnosis.

Benchmark	Input	# Nodes	# Grains	$max(\theta_c)$	Savings (%)	Low Parallel Benefit		
						# Prbl. Grains	$max(\theta_{pb})$	Savings (%)
FFT[1]	16777216, 8192, 2	9240	4592	53	99.43	414	49	99.47
Floorplan[1]	15, 7	117960	82490	149	99.87	31125	148	99.87
NQueens[1]	14, 4	24565	21492	70	99.71	10540	66	99.73
Sort[1]	20971520, 65536, 8192, 128	20293	11509	55	99.73	288	51	99.75
Strassen[1]	8192, 128, 2000	176480	137258	60	99.97	157	49	99.97
Blackscholes[2]	4M	2205	1201	112	94.92	400	112	94.92
Bodytrack[2]	B261, 4, 261, 4000, 5, 3, 48, 0	126615	69061	5767	95.45	24627	5757	95.45
Freqmine[2]	kosarak_990k.dat, 790	2111	2017	389	81.57	66	30	98.58
358.botsalgn[3]	prot.200.aa	20505	20101	406	98.02	7	17	99.92
359.botsspar[3]	64, 64	24161	23905	1154	95.22	2	9	99.96
367.imagick[3]	See caption of Figure 6	3935	3801	405	89.71	649	182	95.37
376.kdtree[3]	200000, 10, 2	32808	16400	58	99.82	2055	57	99.83

[1] BOTS [2] Parsec [3] SPEC-OMP12

grouped during the separation phase (Section 3). Benchmarks Freqmine, 367.imagick, 358.botsalgn, 359.botsspar, show large savings from aggregation since they contain a small number of problematic grains. On the other hand, Bodytrack and Floorplan show barely any improvement over the conservative case due to a higher concentration of problematic grains that are clustered as siblings. Problematic siblings are ignored during separation by design.

We further illustrate the benefit of aggregation using the aggregated grain graph of 367.imagick benchmark from SPEC-OMP12 for the input that SPEC programmers have noted as poorly scaling. 367.imagick is a clone of *ImageMagick*, a software suite for image processing used widely on the command-line in UNIX-like systems [11].

The unaggregated grain graph of 367.imagick shows a chain of nine dense for-loops (Figure 6a). The sixth loop contains several chunks that suffer from low parallel benefit since they miss parallelization throttling macros called *omp_throttle* in source code. Diagnosing these problematic chunks requires programmers to sweep deeply across the graph, all the while ignoring the abundance of non-problematic grains and the frequent non-responsive rendering of the graph by the overloaded graph viewer program. The aggregated grain graph enables programmers to diagnose problematic chunks group by group (Figure 6b). Only the group that contains problematic chunks is kept open. Other uninteresting loops are hidden from sight in closed groups. Non-problematic chunks are separated to reduce distractions. The graph viewer program can respond quickly since only a small fraction of nodes need to be rendered.

6 RELATED WORK

Aggregation is a standard approach to make visualizations scale with increasing data [13, 28]. Sensible dimensions for aggregation are found in many places including the program structure (for example, tasks), middleware stack (worker threads), physical processing components (processors), and the visualization (node-links). However, since aggregation essentially reduces data and can be applied aggressively, insights seen only when the aggregation dimensions are differentiated can be lost. Isaacs *et al.* [13] recognize this balance between the amount of aggregation and showing useful information to programmers as an important challenge. Our aggregation method for grain graphs strives to maintain the same balance by reducing the size of the rendered grain graph and focusing it on problematic sections, without losing the fork-join perspective expected by programmers.

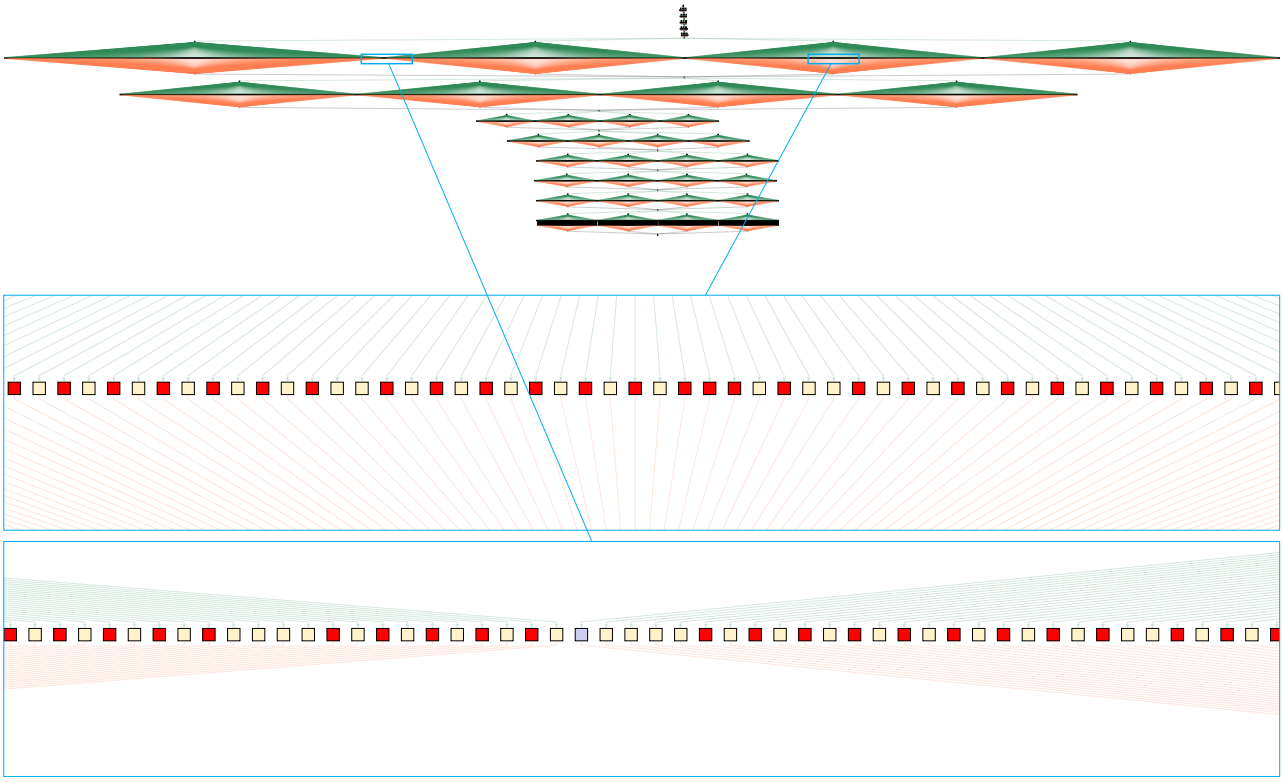
For space reasons, we restrict our discussion to aggregation of abstraction-centric, logical-time visualizations similar to grain graphs, and direct readers interested in other aggregated visualizations to recent surveys [13, 28] and an excellent visualization explorer [14].

The dominant aggregation scheme in visualizations is statistical rather than visual, *i.e.*, metrics of selected elements in the main visualization are aggregated statistically and reported in a separate visualization, typically as a property table [1, 2, 4, 7, 9, 25]. Cognitive load of the main visualization is reduced only by zooming out to focus on large elements. Support for visual aggregation while maintaining the same zoom level is absent. Consequently, such visualizations suffer similar navigation and diagnosis difficulties as large grain graphs.

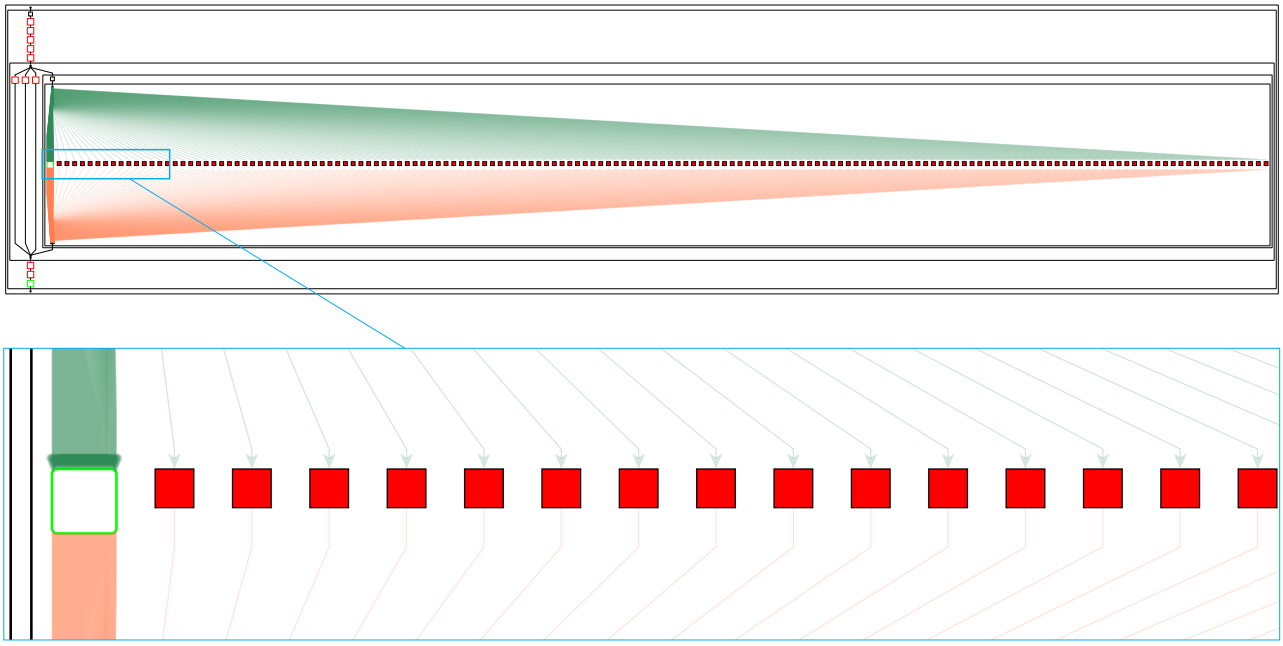
The aggregation method for task graphs in *DAGViz* [10] closely resembles our work. DAGViz presents programmers with a single aggregated node that can be interactively opened to reveal subgraphs. Our approach is tailored to grain graphs and is unique in identifying the critical path and similarity of subgraphs. Fluid interactions and a dedicated viewer are strengths of DAGViz that inspire our future work.

ThreadScope [29] visualizes the dynamically unfolded, logical-time structure of task-parallel programs. Memory operations, shown as nodes in the visualization, can be aggregated into groups to improve clarity. It is not reported whether programmers can interact with groups to uncover members.

The *causality graph* [32] visualization allows programmers overwhelmed by large graphs to manually select and aggregate nodes



(a)



(b)

Figure 6: Diagnosing problems with grains of 367.imagick from SPEC-OMP12 for input `-shear 31 -resize 1280x960 -negate -edge 14 -implode 1.2 -flop -convolve 1,2,1,4,3,4,1,2,1 -edge 100 ref/input/input1.tga`. (a) Sweeping across the entire unaggregated graph with 3801 grains to spot problems. (b) Aggregated grain graph enables programmers to diagnose problematic grains group-wise. Non-problematic grains are separated to promote focus (inset).

within the same scope into group nodes called *supernodes*. Supernodes can be repeatedly aggregated and uncovered to show member nodes. Special care is taken to ensure cycles are not created when supernodes are added to the graph. Supernode metrics include the local critical path and metrics computed using user-defined combination operations. Our grain graph aggregation method is similar except for the presentation of the aggregated graph. We present a fully-aggregated graph that programmers can progressively uncover and spot problems guided by sensible aggregation metrics.

7 CONCLUSION

The contribution of this paper is an aggregation method that reduces programmer effort spent in navigating and diagnosing problems in large grain graphs. The aggregation method groups nodes arranged in recurring patterns in the grain graph to produce a single-node aggregated graph that programmers navigate by progressively opening and closing groups. Problematic groups are highlighted and non-problematic sections are cleared from sight in the aggregated grain graph, enabling focus without compromising the fork-join perspective expected by programmers. Using standard OpenMP programs as examples, the paper demonstrated that aggregated grain graphs significantly reduce the number of visible nodes while diagnosing problems. For future work, we plan to implement a dedicated viewer for aggregated grain graphs that smoothly guides programmers towards urgent and important problems.

ACKNOWLEDGMENT

The paper was funded by the TULIPP project (grant number 688403) and the READEX project (grant number 671657) from the EU Horizon 2020 Research and Innovation programme. The authors thank Peder Voldnes Langdal (NTNU), Magnus Själander (NTNU), and Jan Christian Meyer (NTNU) for constructive comments and KTH Royal Institute of Technology for providing test machinery.

REFERENCES

[1] Barcelona Supercomputing Center. 2013. OmpSs Task Dependency Graph. (2013). http://pm.bsc.es/ompss-docs/user-guide/run-programs-plugin-instrument-tdg.html. Accessed 10 April 2015.

[2] Wolfgang Blochinger, Michael Kaufmann, and Martin Siebenhaller. 2005. Visualizing Structural Properties of Irregular Parallel Computations. In *Proceedings of the 2005 ACM Symposium on Software Visualization*. ACM, 125–134.

[3] Ulrik Brandes, Markus Eiglsperger, and Jürgen Lerner. 2017. GRAPHML primer. (2017). http://graphml.graphdrawing.org/primer/graphml-primer.html. Accessed 27 July 2017.

[4] Steffen Brinkmann, José Gracia, and Christoph Niethammer. 2013. Task Debugging with TEMANEJO. In *Tools for High Performance Computing 2012*. Springer, 13–21.

[5] Gabor Csardi and Tamas Nepusz. 2006. The igraph software package for complex network research. *InterJournal* Complex Systems (2006), 1695.

[6] Janice E. Cuny, Alfred A. Hough, and Joydip Kundu. 1992. Logical time in visualizations produced by parallel programs. In *Proceedings of the IEEE Conference on Visualization*. 186–193.

[7] Andi Drebes, Jean-Baptiste Bréjon, Antoniu Pop, Karine Heydemann, and Albert Cohen. 2016. Language-Centric Performance Analysis of OpenMP Programs with Aftermath. In *International Workshop on OpenMP*.

[8] Markus Eiglsperger, Martin Siebenhaller, and Michael Kaufmann. 2004. An efficient implementation of Sugiyama's algorithm for layered graph drawing. In *International Symposium on Graph Drawing*. Springer, 155–166.

[9] Blake Haugen, Stephen Richmond, Jakub Kurzak, Chad A. Steed, and Jack Dongarra. 2015. Visualizing Execution Traces with Task Dependencies. In *Proceedings of the 2Nd Workshop on Visual Performance Analysis*. ACM, 2:1–2:8.

[10] An Huynh, Douglas Thain, Miquel Pericàs, and Kenjiro Taura. 2015. DAGViz: a DAG visualization tool for analyzing task-parallel program traces. In *Proceedings of the 2nd Workshop on Visual Performance Analysis*. ACM, 3.

[11] ImageMagick Studio LLC. 2017. ImageMagick Official Website. (2017). https://www.imagemagick.org. Accessed 27 July 2017.

[12] Katherine E. Isaacs, Peer-Timo Bremer, Ilir Jusufi, Todd Gamblin, Abhinav Bhatele, Martin Schulz, and Bernd Hamann. 2014. Combing the communication hairball: Visualizing large-scale parallel execution traces using logical time. *IEEE Transactions on Visualization and Computer Graphics, Proceedings of InfoVis '14* 12 (2014).

[13] Katherine E Isaacs, Alfredo Giménez, Ilir Juster, Todd Gamblin, Abhinav Bhatele, Martin Schulz, Bernd Hamann, and Peer-Timo Bremer. 2014. State of the art of performance visualization. *EuroVis 2014* (2014).

[14] Katherine Isaacs. 2017. Performance Visualization: Living digital library of State of the Art of Performance Visualization. (2017). http://cgi.cs.arizona.edu/~kisaacs/STAR/. Accessed 31 July 2017.

[15] Peder Voldnes Langdal, Magnus Jahre, and Ananya Muddukrishna. 2017. Extending OMPT to Support Grain Graphs. In *Scaling OpenMP for Exascale Performance and Portability*. Springer. To appear in proceedings of the International Workshop on OpenMP (IWOMP) 2017.

[16] Ananya Muddukrishna, Peter A. Jonsson, and Mats Brorsson. 2015. Characterizing Task-Based OpenMP Programs. *PLoS ONE* 10, 4 (2015), e0123545.

[17] Ananya Muddukrishna, Peter A Jonsson, and Mats Brorsson. 2015. Locality-aware task scheduling and data distribution for OpenMP programs on NUMA systems and manycore processors. *Scientific Programming* 2015 (2015), 5.

[18] Ananya Muddukrishna, Peter A. Jonsson, and Peder Langdal. 2017. anamud/mir-dev: MIR v1.0.0. (March 2017). https://doi.org/10.5281/zenodo.439351

[19] Ananya Muddukrishna, Peter A. Jonsson, Artur Podobas, and Mats Brorsson. 2016. Grain Graphs: OpenMP Performance Analysis Made Easy. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, 28:1–28:13.

[20] Ananya Muddukrishna and Peder Langdal. 2017. anamud/grain-graphs: Grain Graphs v1.0.0. (March 2017). https://doi.org/10.5281/zenodo.439355

[21] Stephen L. Olivier, Bronis R. de Supinski, Martin Schulz, and Jan F. Prins. 2012. Characterizing and Mitigating Work Time Inflation in Task Parallel Programs. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE Computer Society Press, 65:1–65:12.

[22] Nico Reissmann. 2017. phate/ggraph: VPA17. (July 2017). https://doi.org/10.5281/zenodo.836838

[23] Nino Shervashidze, Pascal Schweitzer, Erik Jan van Leeuwen, Kurt Mehlhorn, and Karsten M. Borgwardt. 2011. Weisfeiler-Lehman Graph Kernels. *Journal of Machine Learning Research* 12, Sep (2011), 2539–2561.

[24] Michael E Smoot, Keiichiro Ono, Johannes Ruscheinski, Peng-Liang Wang, and Trey Ideker. 2011. Cytoscape 2.8: new features for data integration and network visualization. *Bioinformatics* 27, 3 (2011), 431–432.

[25] Vladimir Subotic, Steffen Brinkmann, Vladimir Marjanovic, Rosa M. Badia, Jose Gracia, Christoph Niethammer, Eduard Ayguade, Jesus Labarta, and Mateo Valero. 2013. Programmability and portability for exascale: Top down programming methodology and tools with StarSs. *Journal of Computational Science* 4, 6 (2013), 450 – 456.

[26] Kozo Sugiyama, Shojiro Tagawa, and Mitsuhiko Toda. 1981. Methods for visual understanding of hierarchical system structures. *IEEE Transactions on Systems, Man, and Cybernetics* 11, 2 (1981), 109–125.

[27] Mahito Sugiyama, M Elisabetta Ghisu, Felipe Llinares-López, and Karsten Borgwardt. 2017. graphkernels: R and Python packages for graph comparison. *Bioinformatics* (2017), btx602.

[28] Tatiana Von Landesberger, Arjan Kuijper, Tobias Schreck, Jörn Kohlhammer, Jarke J van Wijk, J-D Fekete, and Dieter W Fellner. 2011. Visual analysis of large graphs: state-of-the-art and future research challenges. In *Computer graphics forum*, Vol. 30. Wiley Online Library, 1719–1749.

[29] Kyle B Wheeler and Douglas Thain. 2010. Visualizing massively multithreaded applications with ThreadScope. *Concurrency and Computation: Practice and Experience* 22, 1 (2010), 45–67.

[30] Richard M. Yoo, Christopher J. Hughes, Changkyu Kim, Yen-Kuang Chen, and Christos Kozyrakis. 2013. Locality-aware Task Management for Unstructured Parallelism: A Quantitative Limit Study. In *Proceedings of the Twenty-fifth Annual ACM Symposium on Parallelism in Algorithms and Architectures*. ACM, 315–325.

[31] yWorks GmBh. 2015. yEd Graph Editor. (2015). http://www.yworks.com/en/products_yed_about.html. Accessed 10 April 2015.

[32] Dror Zernik, Marc Snir, and Dalia Malki. 1992. Using visualization tools to understand concurrency. *IEEE Software* 9, 3 (1992), 87–92.