Diagnosing Highly-Parallel OpenMP Programs With Aggregated Grain Graphs

Nico Reissmann and Ananya Muddukrishna

Norwegian University of Science and Technology
firstname.lastname@ntnu.no

Abstract. Grain graphs simplify OpenMP performance analysis by visualizing performance problems from a fork-join perspective that is familiar to programmers. However, when programmers decide to expose a high amount of parallelism by creating thousands of task and parallel for-loop chunk instances, the resulting grain graph becomes large and tedious to understand. We present an aggregation method that hierarchically groups related nodes together to reduce grain graphs of any size to one single node. This aggregated graph is then navigated by progressively uncovering groups and following visual clues that guide programmers towards problems while hiding non-problematic regions. Our approach enhances productivity by enabling programmers to understand problems in highly-parallel OpenMP programs with less effort than before.

1 Introduction

The *grain graph* [1] is a recent visualization method that simplifies OpenMP performance analysis by highlighting problems from a fork-join perspective. Task and parallel for-loop chunk instances are collectively termed *grains* in the grain graph method. Grains that suffer performance problems such as work inflation, inadequate parallelism, and low parallelization benefit are pinpointed on the grain graph along with precise links to the problematic source code. This enables programmers to perform optimizations productively without relying on experts or trial-and-error tuning.

Programmers optimize OpenMP programs for large machines with hundreds of cores by exposing a high amount of parallelism during execution. This is achieved by adjusting special program inputs called *cutoffs* and *chunk sizes* such that a large number of fine-grained tasks and for-loop chunks are created. Scalability problems invariably occur when the runtime system is unable to efficiently handle the parallelism exposed [2–4]. These problems are pinpointed on the grain graph using metrics that isolate low parallelization benefit, work inflation, and poor memory hierarchy utilization to specific grains.

However, the large grain graphs resulting from highly-parallel OpenMP execution make problem diagnosis tedious (Fig. 1). Programmers have to zoom and pan to different sections while remembering characteristics of visited sections. 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 memory is required 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.

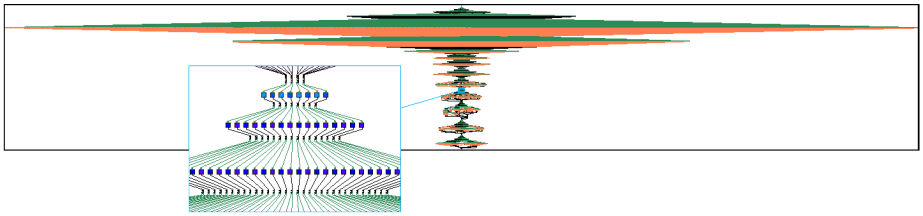


Fig. 1. The grain graph of the task-recursive Sort program from the Barcelona OpenMP Task Suite (BOTS) for a high-parallelism input (`n=20971520, cutoffs={65536,8192,128}`) is dense with 11059 grains. Inset (blue box) zooms into a section at magnification 40X.

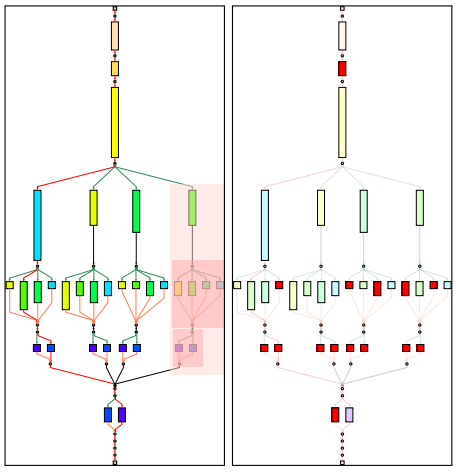
This paper contributes with a new aggregation method that makes visual analysis of large grain graphs practical. The aggregation method (Section 3) groups related nodes by matching recurrent patterns in the grain graph, ultimately resulting in an aggregated graph with a single group node. Programmers 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. Navigation is further sped up through new group-based metrics that enable programmers to traverse the critical path and compare groups for structural similarity. Using highly-parallel executions of standard OpenMP programs, we demonstrate (Sections 3 and 4) that aggregated grain graphs enhance the the state-of-the-art in OpenMP problem diagnosis.

2 Background on Grain Graphs

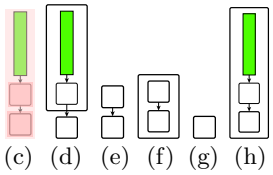
The grain graph [1] is a visualization for OpenMP that connects performance problems to the fork-join program structure at the resolution of *grains* – task and parallel for-loop chunk instances created during execution. This simplifies problem diagnosis as programmers can readily identify with the fork-join program structure. 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 [1, 5]. Experts who understand scheduling internals nevertheless find it tiring to follow timelines and call graphs that depict recursive task-based execution – a popular style of using OpenMP.

2.1 Structure

The grain graph is a directed acyclic graph 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 using *logical-time* placement [6, 5] to maintain familiarity with the fork-join perspective (Fig. 2a[1]). The grain graph is laid out using the *Sugiyama* layout [7, 8]. 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.



(a) (b)



(c) (d) (e) (f) (g) (h)

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

(i)

Fig. 2. Grain graph of the task-based Sort program from BOTS for small input (`n=512, cutoffs={256,64,16}`). (a) Structural view (b) Problem view highlighting low parallel benefit in red (c) After two fork-join pattern reductions of the highlighted subgraph (d-g) Linear pattern reductions leading to a single group node (h) After normalization (i) Reduction pseudocode

2.2 Diagnosing problems

Grains are annotated with unique schedule-independent identifiers, links to source code locations, as well as performance metrics measured during profiling and derived post profiling. Profiled metrics include execution time, cache miss ratio, memory latency, and timestamps of control-flow events such as grain creation and synchronization. These metrics 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 of this paper. It is computed by dividing a grain's execution time by its parallelization cost

[1] Readers should print in color as they are crucial to appreciate grains graphs.

including creation time. This metric aids inlining and cutoff decisions as 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 (Fig. 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 programmer-defined thresholds are inferred as problematic. The thresholds have sensible values by default. Problematic grains are highlighted with a color that encodes problem severity in a separate view while non-problematic grains are dimmed (Fig. 2b). Additionally, problems are summarized in a separate text file and highlighted in a tabular form of the grain graph shown on a separate visualization widget.

Grain graphs have multiple conceptual views with colors encoding a single problem or property per view. Programmers shift between these views to understand properties or tackle problems. Problematic grains are highlighted and non-problematic grains are dimmed, and clicking on a grain opens a separate window that shows the grain's properties and performance metrics. Fig. 2b-a show the programmer cycling between the low parallel benefit problem view and the structural view where no problems are highlighted.

3 Grain Graph Aggregation Method

Our aggregation method for grain graphs conceptually consists of four phases:

1. **Reduction** matches and replaces subgraph patterns with group nodes to construct an *aggregation tree*. This tree captures the graph structure and serves as a basis for further processing. After aggregation is complete, the tree is converted back to an aggregated grain graph with problematic grains exposed and non-problematic grains hidden.
2. **Normalization** transforms the aggregation tree into a canonical form, simplifying further processing.
3. **Propagation** propagates grain metrics at the leaves of the tree to upper levels in a sensible manner.
4. **Separation** transforms the aggregation tree to separate problematic nodes. This enables grouping and hiding of non-problematic grains in the resulting aggregated graph.

The algorithmic complexity of all four phases is linear in the number of graph nodes plus edges. The rest of this section explains the phases in detail and discusses the navigation of the resulting aggregated graph at the end.

3.1 Reduction

The reduction phase matches a *fork-join* and *linear* pattern, and replaces them with group nodes to construct an *aggregation tree*. 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 (Fig. 2c). The linear pattern has two nodes, either a grain or a group node, that are connected to each other (Fig. 2d). Both patterns are repeatedly matched, and replaced by a single group node until the entire grain graph is reduced to a single node (Fig. 2d-g).

The pseudocode of the reduction algorithm is shown in (Fig. 2i). The key steps in the psuedocode are explained next:

- Line 6 matches the linear pattern (Fig. 2d-g). 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 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 (Fig. 2c). 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 (Fig. 2a) and 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.

The grain graph is reduced greedily by the reduction algorithm. It always continues with the newly-created group node after a pattern match and never traverses past a join node. This ensures that the innermost fork-join in a nesting is reduced first.

The aggregation tree consisting of group and grain nodes explicitly captures the grain graph's nesting and fork-join structure. The leaves of the tree are grains and its intermediate nodes are the newly-created group nodes. Linear group nodes have the two matched nodes from the pattern as children, whereas fork-join group nodes have the children of the matched fork node as children.

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

3.2 Normalization

Normalization transforms the aggregation tree into a canonical form by flattening nested linear group nodes. In the reduction phase, linear group nodes are 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 Fig. 2d-g. Normalization flattens these subtrees to a single linear group node with all non-linear group

nodes from the subtree as its children (Fig. 2h). In practice, this phase can be incorporated into the previous phase to speedup aggregation.

3.3 Propagation

This phase propagates leaf node metrics to the enclosing groups all the way up to the root node. It traverses the aggregation tree in post-order and attributes group nodes with metrics sensibly-derived from their children. For example, the *work* metric of a group node is the sum of the execution times of its children, while the schedule-independent identifiers of children are concatenated with the group node's depth to derive a schedule-independent identifier.

Metrics are attributed such that problems propagate to the root group. If a child is problematic, then its 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. Programmers can refine existing propagation metrics and define new ones. Given this ability, the range of values and other summary statistics of a group can be easily captured (for example, as string attributes). One useful custom metric that programmers could define is the percentage of time spent by a group on the critical path.

3.4 Separation

The separation phase groups non-problematic nodes to separate them from problematic nodes. This enables programmers to focus on problems and reduces graph viewer load. 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, while a separated graph requires only the rendering of one problematic grain and a non-problematic group node.

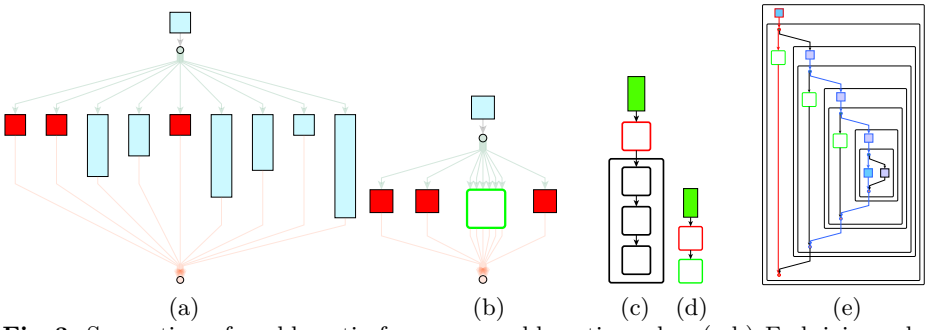


Fig. 3. Separation of problematic from non-problematic nodes. (a-b) Fork-join node separation. (c-d) Linear node separation. (e) Local (blue) and global (red) critical paths

Separation traverses the aggregation tree in post-order and separates subtrees rooted at fork-join and linear nodes. In a fork-join separation, all non-

problematic children of a fork-join node are grouped under a newly-created group node (Fig. 3a-b), while in a linear node separation, all consecutive non-problematic children of a linear group node are grouped under a new linear group node (Fig. 3c-d). After the separation phase, the aggregation tree is converted back to a grain graph where non-problematic subgraphs are hidden.

3.5 Navigation

The navigation of an aggregated graph starts at the root and continues by progressively opening/closing group nodes to understand graph structure and problems (Fig. 4). In contrast to the navigation in unaggregated graphs, the cognitive load on programmers and the graph viewer's resources are reduced as only a subset of the grains are laid out. Navigation is sped up using several optimizations:

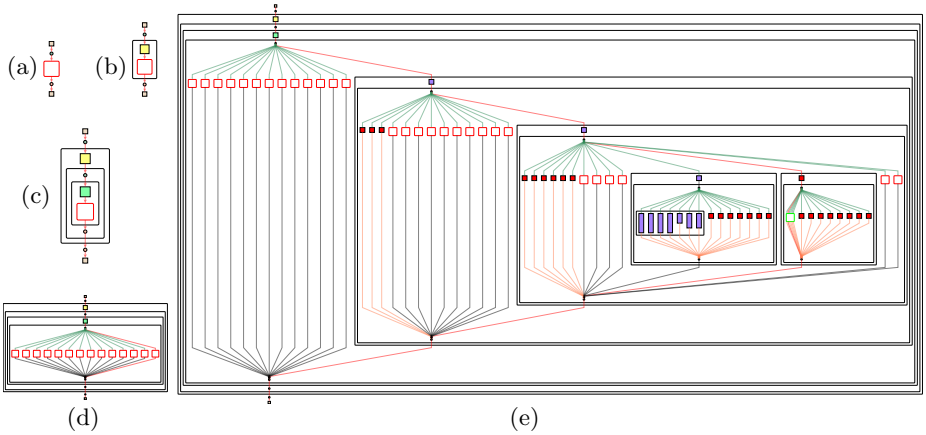


Fig. 4. Navigating the aggregated grain graph of NQueens program from BOTS for high-parallelism input (`n=14, cutoff=4`). The graph has 21492 grains and 3073 group nodes. Grains with low parallel benefit are highlighted as problems. (a-d) Drilling down to sibling groups at a depth of 3 from the root group. (a) Root group. (b) At depth 1. (c) At depth 2. (d) At depth 3. (e) Drilling down along the critical path to sibling groups at the lowest depth.

1. Groups can be opened to show all grains including those inside subgroups (full collapse), or drilled down to a specific group or depth level (Fig. 4).
2. Group nodes are drawn as rounded rectangles with no filling to differentiate them 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, while the borders of non-problematic groups are colored green for quick identification. Our choices of 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 similarly structured groups can be navigated confidently or skipped if problem-free. For example, twelve groups in Fig. 4d have the same structure. Group similarity is computed on-demand using a Weisfeiler-Lehman graph kernel [9].
4. Groups on the *global critical path* (gcb) are inspected first since they are good optimization candidates (Fig. 4e). The local critical path of groups not on the gcb can be computed on-demand and used for prioritized inspection (Fig. 3e). If off-gcb grains are optimized to reduce the total amount of work, the resulting slack can be used to execute grains on the gcb.

4 Prototype Implementation

The grain graph visualization is implemented in a prototype [10] that produces grain graphs in GRAPHML by processing profiling data from OMPT extensions [11] or the MIR runtime system [12, 4, 13]. We extended the prototype to produce aggregated graphs upon programmer request [14]. The aggregation method was implemented in C++, leveraging support for nested groups [15] in GRAPHML and using the igraph [16] library for basic graph processing.

We used the graph viewer yEd [17] to visualize aggregated grain graphs since it has sufficiently mature support for GRAPHML files with nested aggregations. For example, it has features to interactively open and close groups, and jump to groups at any hierarchy level. Its property editor dialog shows the annotations of group nodes. Switching between problem views was achieved by cycling through tabs that highlighted different problems.

External programs parameterized by group identifiers were used to compute 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 third-party implementation [18] of the Weisfeiler-Lehman graph kernel.

We recognize that interactions with aggregated graphs in yED have quite some room for improvement. Our plan is to incorporate improvements in a dedicated grain graph viewer as yEd is closed-source. The dedicated viewer will also enable programmers to define custom metrics derived from basic grain and group metrics in a GUI. This improves over the prototype where programmers customize metrics by editing source-code in convenient locations.

5 Evaluation

We tested our prototype on 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 a 48-core machine with 64GB memory and four AMD Opteron 6172 processors running at 2.1GHz with frequency scaling disabled. We provided input values that exposed abundant, fine-grained parallelism to standard OpenMP programs to obtain large grain graphs (Table 1).

5.1 Visible node count

We use the metric *visible node count* (θ) to judge the ability of our aggregation method to reduce programmer effort in navigating and diagnosing problems. θ is defined as the minimum number of visible nodes in a grain graph while diagnosing a problematic grain. If it is small, the cognitive load on programmers and the resource requirements of viewers are reduced.

The visible node count for a problematic grain in an aggregated 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 graph is equal to the number of nodes in the entire graph irrespective of the position of the problematic grain, assuming programmers do not pan and zoom to the vicinity of the problematic grain manually.

Table 1 shows the maximum θ for two cases. The first is a conservative case (θ_c^{max}) that assumes all grains in the graph are problematic, while the second (θ_{pb}^{max}) considers graphs with low parallel benefit. For both cases, the reduction in maximum θ compared to the total size of the graph, *i.e.*, the maximum θ for the unaggregated graph, is reported as *Savings*.

Table 1. Benefit of aggregation for standard OpenMP benchmarks.

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

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

For the conservative case, we see a large reduction in θ. The biggest saving is 99.97% for the Strassen benchmark and the smallest saving is 81.57% for Freqmine, with an average saving of 95.98%. This shows that aggregation can significantly reduce θ for any problematic grain in our evaluation setup.

For the second case, we see a further reduction in θ since non-problematic grains are 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.

5.2 Reducing distractions

We further illustrate the benefit of aggregation using the 367.imagick benchmark from SPEC-OMP12 for an input that SPEC programmers noticed as poorly scaling. The unaggregated grain graph shows a chain of nine dense for-loops (Fig. 5a). The sixth loop contains several chunks that suffer from low parallel benefit since several instances of the parallelization-throttling macro *omp_throttle* are missing in the source. Diagnosing these problematic chunks requires programmers to sweep attentively across the graph ignoring the abundance of non-problematic grains and the frequent non-responsive rendering of the graph. The aggregated graph enables programmers to diagnose problematic chunks group by group (Fig. 5b), keeping only those groups with problematic chunks open, while uninteresting loops and non-problematic chunks are hidden from sight. This results in a more responsive graph viewer since fewer nodes need to be rendered.

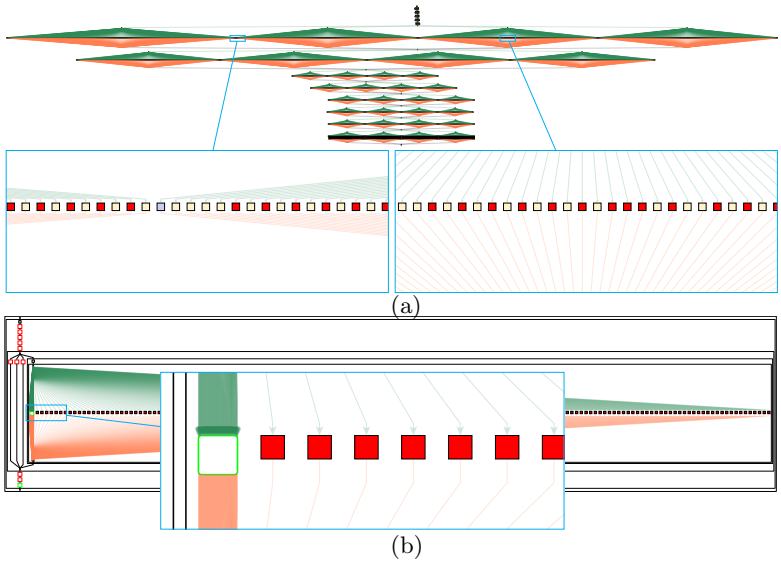


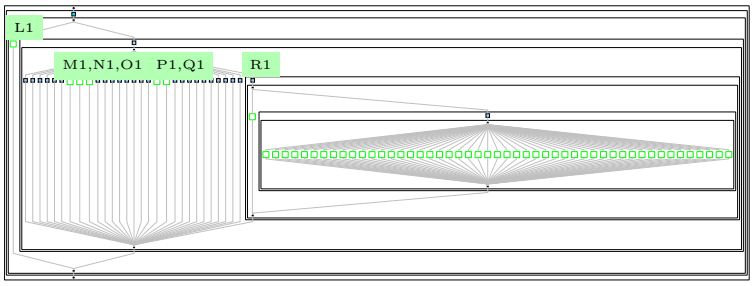
Fig. 5. Diagnosing problems with grains of 367.imagick from SPEC-OMP12 for input `-shear 31 -resize 1280x960 -negate -edge 14 -implode 1.2 -flop -convolve 1,2,1,4,3,4,1,2,1 -edge 100 ref/input/input1.tga`. (a) Sweeping across the entire unaggregated graph with 3801 grains to spot problems. (b) Aggregated grain graph enables programmers to diagnose problematic grains group-wise. Non-problematic grains are separated to promote focus (inset).

5.3 Similarity across runs

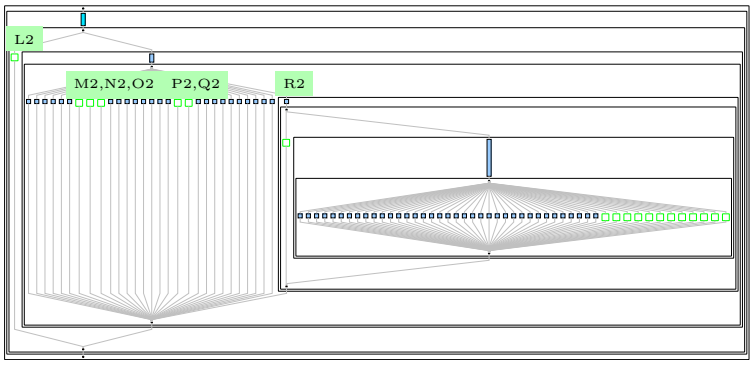
Grain graphs produced from two independent executions of a given program can be different in shape due to unpredictable inlining decisions taken by the run-

time system or if the program adapts its behavior sensitive to available execution resources. Understanding such changes can provide vital clues for problem diagnosis. However, detecting the dissimilar sections by manually inspecting a pair of large grain graphs is extremely tiring and akin to finding matches between fingerprints using a magnifying lens.

Similarity is a powerful metric that not just helps to skip over structurally similar groups within the same graph (as demonstrated in Section 3.5), but can also compare groups across runs to detect structural differences. Programmers can gradually open two graphs side-by-side and compute the similarity metric for visible groups using their schedule-independent identifiers. Those groups that have the same identifier but different similarity metrics are the sections that have changed between the graphs. We demonstrate this for the Floorplan program from BOTS in Figure 6. Floorplan is a search-based program whose pruning behavior changes non-deterministically when more cores are allotted for execution.



(a)



(b)

Fig. 6. Finding dissimilar sections in grain graphs from two independent executions of the non-deterministic Floorplan program from BOTS for input `cell-file=input.5, cutoff=5`. (a) Graph produced from execution on 4 cores has 7974 grains. (b) Graph produced from execution on 48 cores has 3190 grains. The similarity metric allows programmers to understand without inspection that groups L1-2, M1-2, N1-2, and O1-2 have the same structure but P1-2, Q1-2, and R1-2 do not. Groups R1-2 are opened to show the dissimilarity. R2 encloses fewer subgroups than R1.

6 Related Work

Aggregation is a standard approach to scale visualizations with increasing data [19, 20]. Sensible dimensions for aggregation include the program structure (*e.g.* tasks), middleware stack (worker threads), physical processing components (processors), and the visualization (node-links). However, aggregation can remove vital diagnosis data when applied aggressively across several dimensions. Isaacs *et al.* [19] recognize the balance between aggregation aggressiveness and information preservation as an important challenge. Our method strives to maintain this balance by reducing the size of the rendered graph and focusing it on problematic sections, while keeping the expected fork-join perspective.

For space reasons, we restrict the discussion to abstraction-centric, logical-time aggregated visualizations similar to grain graphs, and refer readers for other visualizations to recent surveys [19, 20] and a visualization explorer [21].

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 separately, typically as a property table [22–27]. The cognitive load of the main visualization is only reduced by zooming out to focus on large elements, while support for visual aggregation at the same zoom level is absent. Consequently, such visualizations suffer similar navigation and diagnosis difficulties as large unaggregated grain graphs.

The aggregation method for task graphs in *DAGViz* [28] resembles our work. It presents programmers with a single aggregated node that can be interactively opened to reveal subgraphs as well as a dedicated viewer. However, our approach is tailored to grain graphs and is unique in tracing the critical path and identifying the similarity of subgraphs. Unaggregated grain graphs are more effective in pinpointing problems than unaggregated DAGViz graphs due to more derived metrics. The expansion of DAGViz graphs results also in the rendering of more nodes as they show a fork-node per grain. Grain graphs avoid this thanks to fork-node reductions that produce a fork-node per set of siblings. DAGViz combats the scaling problem by using an elegant aggregation method that reduces subgraphs that executed wholly on a single worker-thread into a single, non-collapsible node.

ThreadScope [29] visualizes the logical-time structure of task-parallel programs. Its memory operations nodes can be grouped to improve clarity, but it is unclear whether programmers can interact with groups to uncover members.

The *causality graph* [30] visualization permits programmers to manually select and repeatedly aggregate nodes into *supernodes*, while special care must be taken to avoid graph cycles on their creation. Supernode metrics include the local critical path and metrics computed using user-defined combinators. The causality graph presents an unaggregated graph by default, while we present a fully aggregated graph and use sensible aggregation metrics to guide programmers.

7 Conclusion

This paper contributes an aggregation method for grain graphs that enables programmers to easily understand problems in highly-parallel OpenMP programs. Our method groups nodes arranged in recurring patterns to produce an aggregated graph that programmers can navigate by progressively opening and closing groups. Problematic groups are highlighted and non-problematic sections are cleared from sight, enabling focus without compromising the fork-join perspective expected by programmers. Using standard OpenMP programs as examples, we demonstrate a significant reduction of visible nodes throughout problem diagnosis. For future work, we plan to implement a dedicated grain graph viewer that smoothly and precisely guides programmers towards OpenMP problems and hints at solutions.

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 NTNU colleagues Peder Voldnes Langdal, Magnus Själander, Jan Christian Meyer, and Magnus Jahre for constructive comments and KTH Royal Institute of Technology for providing test machinery.

References

1. Muddukrishna, A., et al.: Grain Graphs: OpenMP performance analysis made easy. In: PPoPP. (2016)
2. Olivier, S.L., et al.: Characterizing and mitigating work time inflation in task parallel programs. In: SC. (2012)
3. Yoo, R.M., et al.: Locality-aware task management for unstructured parallelism: A quantitative limit study. In: SPAA. (2013)
4. Muddukrishna, A., et al.: Locality-aware task scheduling and data distribution for openmp programs on numa systems and manycore processors. Scientific Programming (2015)
5. Isaacs, K.E., et al.: Combing the communication hairball: Visualizing large-scale parallel execution traces using logical time. InfoVis (2014)
6. Cuny, J.E., et al.: Logical time in visualizations produced by parallel programs. In: IEEE Conference on Visualization. (1992)
7. Sugiyama, K., et al.: Methods for visual understanding of hierarchical system structures. SMC (1981)
8. Eiglsperger, M., et al.: An efficient implementation of Sugiyama's algorithm for layered graph drawing. In: International Symposium on Graph Drawing. (2004)
9. Shervashidze, N., et al.: Weisfeiler-Lehman graph kernels. JLMR (2011)
10. Muddukrishna, A., et al.: anamud/grain-graphs: Grain Graphs v1.0.0 (2017) `https://doi.org/10.5281/zenodo.439355`.
11. Langdal, P.V., et al.: Extending OMPT to Support Grain Graphs. In: IWOMP. (2017)

12. Muddukrishna, A., et al.: anamud/mir-dev: MIR v1.0.0 (2017) https://doi.org/10.5281/zenodo.439351.
13. Muddukrishna, A., et al.: Characterizing task-based OpenMP programs. PLoS ONE (2015)
14. Reissmann, N.: phate/ggraph: Vpa17 (2017) https://doi.org/10.5281/zenodo.836838.
15. Brandes, U., et al.: GRAPHML primer (2017) http://graphml.graphdrawing.org/primer/graphml-primer.html. Accessed 27 July 2017.
16. Csardi, G., et al.: The igraph software package for complex network research. InterJournal (2006)
17. yWorks GmBh: yEd graph editor (2015) http://www.yworks.com/en/products_yed_about.html. Accessed 10 April 2015.
18. Sugiyama, M., et al.: graphkernels: R and python packages for graph comparison. Bioinformatics (2017)
19. Isaacs, K.E., et al.: State of the art of performance visualization. EuroVis (2014)
20. Von Landesberger, T., et al.: Visual analysis of large graphs: state-of-the-art and future research challenges. In: Computer graphics forum. (2011)
21. Katherine Isaacs: 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.
22. Brinkmann, S., et al.: Task debugging with TEMANEJO. In: Tools for High Performance Computing 2012. (2013)
23. 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.
24. Subotic, V., et al.: Programmability and portability for exascale: Top down programming methodology and tools with StarSs. Journal of Computational Science (2013)
25. Blochinger, W., et al.: Visualizing structural properties of irregular parallel computations. In: Vissoft. (2005)
26. Haugen, B., et al.: Visualizing Execution Traces with Task Dependencies. In: VPA. (2015)
27. Drebes, A., et al.: Language-Centric Performance Analysis of OpenMP Programs with Aftermath. In: IWOMP. (2016)
28. Huynh, A., et al.: DAGViz: a DAG visualization tool for analyzing task-parallel program traces. In: VPA. (2015)
29. Wheeler, K.B., et al.: Visualizing massively multithreaded applications with ThreadScope. Concurrency and Computation: Practice and Experience (2010)
30. Zernik, D., et al.: Using visualization tools to understand concurrency. IEEE Software (1992)