Grain Graphs: OpenMP Performance Analysis Made Easy

Ananya Muddukrishna
KTH Royal Institute of Technology
ananya@kth.se

Peter A. Jonsson
SICS Swedish ICT AB
pj@sics.se

Artur Podobas
KTH Royal Institute of Technology
podobas@kth.se

Mats Brorsson
KTH Royal Institute of Technology
matsbror@kth.se

Abstract

Average programmers struggle to solve performance problems in OpenMP programs with tasks and parallel for-loops. Existing performance analysis tools visualize OpenMP task performance from the runtime system's perspective where task execution is interleaved with other tasks in an unpredictable order. Problems with OpenMP parallel for-loops are similarly difficult to resolve since tools only visualize aggregate thread-level statistics such as load imbalance without zooming into a per-chunk granularity. The runtime system/threads oriented visualization provides poor support for understanding problems with task and chunk execution time, parallelism, and memory hierarchy utilization, forcing average programmers to rely on experts or use tedious trial-and-error tuning methods for performance. We present *grain graphs*, a new OpenMP performance analysis method that visualizes *grains* – computation performed by a task or a parallel for-loop chunk instance – and highlights problems such as low parallelism, work inflation and poor parallelization benefit at the grain level. We demonstrate that grain graphs can quickly reveal performance problems that are difficult to detect and characterize in fine detail using existing visualizations in standard OpenMP programs, simplifying OpenMP performance analysis. This enables average programmers to make portable optimizations for poor performing OpenMP programs, reducing pressure on experts and removing the need for tedious trial-and-error tuning.

Categories and Subject Descriptors D2.5 [*Testing and Debugging*]: Diagnostics

Keywords OpenMP, Task-based Programs, Performance Analysis, Performance Visualization

1. Introduction

When programmers express parallelism using tasks or parallel for-loops with suitably fine granularity and good memory hierarchy utilization, OpenMP stands out as a productive and performance portable technique to write composable parallel programs. Parallel for-loops have been in OpenMP since version 1.0 and tasks were introduced in 2008 as fork-join programming constructs in OpenMP 3.0. Tasks were extended with support for data-flow task programming in OpenMP 4.0 and support for task-generating for-loops has recently been announced in the latest version 4.5 [31].

Although increased importance is being placed on tasks by the OpenMP committee, the quality of information provided by visualizations in OpenMP tools is insufficient for analyzing task-based problems [37]. Tools visualize program execution from the low-level perspective of the runtime system and show that tasks execute interleaved with other tasks in a runtime-optimized order without immediate connection to the program structure.

There is a similar problem with parallel for-loops where iterations are distributed to different threads in chunks. Tools once again only visualize aggregate thread-level statistics such as load imbalance without clear distinction between the different chunks.

The lack of convenient visualization forces experts to use a combination of clever manual instrumentation, deep algorithmic knowledge, as well as knowledge about the internals of the compiler and the runtime system to identify and pin-point performance problems. Average programmers instead struggle to understand problems in granularity, parallelism, and memory hierarchy utilization. Most often, the only practical use of existing visualizations is to infer load balance on different cores without any immediate connection to the root cause of the problem. Average programmers therefore abandon hope of actionable feedback from existing visualizations and hand over the problem to experts or resort to trial-and-error tuning methods to overcome performance problems.

Performance analysis would be simplified for experts and non-experts alike if tools provided visualizations of the execution which were immediately connected to the program structure.

We present *grain graphs*, a new OpenMP performance analysis method that visualizes *grains* – computation performed by a task or a parallel for-loop chunk instance – from a predictable program perspective while retaining essential runtime system execution aspects such as scheduling and memory hierarchy performance profiled at OMPT-like [16] events. Performance crippling conditions such as low parallelism, work-inflation [30], and poor parallelization benefit are derived at the grain level and depicted directly on the grain graph with precise links that connect problem areas to source code. By immediately revealing and pin-pointing problems that are difficult to detect using existing visualizations, our method enables average programmers to quickly make optimizations for poor performing OpenMP programs without resorting to trial-and-error tuning. Experts can start fixing problems right away without

PPoPP '16 March 12-16, 2016, Barcelona, Spain

DOI: http://dx.doi.org/10.1145/2851141.2851156

spending valuable time on tracking down and isolating the problem first.

Grain graphs have helped us discover new problems and peer deeper into known problems in standard OpenMP programs from SPEC OMP 2012 [29] (SPEC-OMP), Barcelona OpenMP Task Suite 1.1.2 [14] (BOTS), and Parsec 3.0 [5] (Parsec). We found that cutoffs in 376.kdtree from SPEC-OMP and Strassen from BOTS were disabled due to programmer mistakes. The cutoff in 376.kdtree has a recursive call where the depth is not incremented and Strassen has a hard-coded cutoff that overrides user input. We also pinpoint work-inflation at the grain level in 359.botsspar from SPEC-OMP, demonstrate low parallelism as an incurable condition in the Sort from BOTS and quickly add cutoffs to improve parallelization benefit in tasks of the troubled FFT program from BOTS.

We describe the overall contribution of the paper – the design of grain graphs and associated derived metrics to pin-point performance problems in OpenMP programs at the grain level from a predictable program perspective – in Section 3. Sections 2 and 4 demonstrate that grain graphs can pin-point performance problems in the C/C++ programs in three common benchmark suites – SPEC-OMP, Parsec, and BOTS. We improved the scalability of the programs based on grain graph visualization up to 54.9 times the original scalability on a 48-core NUMA machine. Our visualization method can guide average programmers to understand and make optimizations for poor performing standard programs. Limitations of existing visualizations are described in detail in the related work section (Section 5).

Note: Colors are crucial to appreciate grain graphs. We request readers to print the paper in color.

2. Need for Grain Graphs

376.kdtree is a fork-join task-based program from SPEC OMP 2012 that searches a k-d tree for neighboring points within a radius. Tasks are used to sweep the tree for points and to find neighbors for each point. The program takes a cutoff parameter that prevents creation of tasks after a threshold recursion depth is reached. The documentation states that low task counts are crucial for performance.

Figure 1 shows that 376.kdtree performs poorly on GCC and MIR runtime systems with the SPEC *reference* input (tree size 400,000, radius 10, and cutoff 2). Visualizations in existing tools

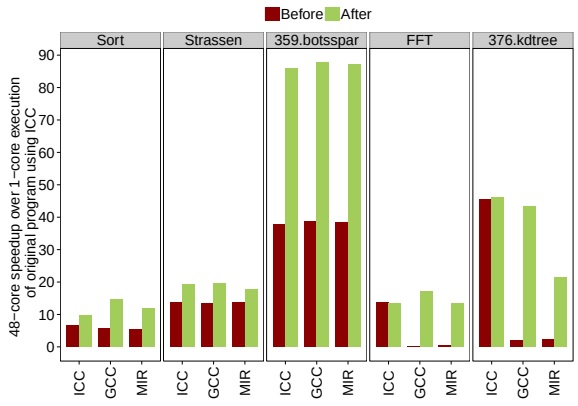

Figure 1: Performance improves after optimization on all runtime systems.

show that load is balanced and provide no further information to understand and improve performance.

Grain graphs place parent and child grains in close proximity using creation edges, without timing as a placement constraint.

The grain graph for 376.kdtree immediately reveals that the program creates many tasks by recursing to a large depth during the sweep phase, as shown in Figure 2. This is surprising since the cutoff should have limited the number of tasks created.

Inspecting `kdnode::sweeptree()`, the function that sweeps the tree, shows that the depth is not incremented for recursive calls which explains the large number (1488595) of tasks created for the SPEC reference input. Incrementing the depth for recursive calls and separating the sweep cutoff from the original cutoff improves performance. We increase the value of the original cutoff from 2 to 8 and use 10 as the sweep cutoff for GCC and MIR. We use 100 as the sweep cutoff value for ICC. With the new cutoffs, scalability of 376.kdtree increases on GCC and MIR runtime systems as shown in Figure 1. The ICC runtime system overcomes the faulty cutoff in the original program and performs well by using an internal cutoff [20] to limit the number of the tasks.

The missing depth increment escaped both the programmer and SPEC quality control for over three years. Our method immediately reveals structural anomalies that can cripple performance.

3. Grain Graphs

We describe the structure, the metrics and how performance problems are highlighted on grain graphs in this section.

3.1 Structure

Constructing grain graphs requires dealing with OpenMP peculiarities such as recursive task creation, *all-at-once* child synchronization, and parallel for-loop chunk assignment, all of which deviate from traditional data-flow graphs [39]. We contribute a concrete mechanizable structure that faithfully captures execution with aforementioned peculiarities.

The grain graph is a directed acyclic graph (DAG) that captures the order of creation and synchronization between grains. A grain denotes the computation performed by a task or a parallel for-loop chunk instance. For a deterministic task-based program with a fixed input, the grain graph is independent from machine size and scheduling choices during program execution. For for-loop based programs the shape of the graph is dependent on the number of threads used during profiling.

The grain graph consists of five types of nodes – fragment, fork, join, book-keeping nodes, and chunk nodes. (Figures 3c, g). Fragment nodes denote the execution of a task between its creation and synchronization. They are also essential to depict parent context execution after returning from runtime system calls. Fragment nodes are uniquely identified and are ordered sequentially within the context of the task instance. Green fork nodes denote task creation and orange join nodes denotes task synchronization. Book-keeping nodes represent computation performed by threads to divide the iteration space and assign iterations to themselves in chunks. Chunk nodes denote computation performed by the set of iterations assigned to chunks. Fragment nodes, book-keeping nodes, and chunk nodes are weighted with metrics measured during execution.

The grain graph has three types of control-flow edges – creation, synchronization and continuation. Green creation edges connect a fork node to the first fragment node of a child. Orange join edges connect the last fragment of children synchronizing with the parent. Black continuation edges connect fragments to fork or join nodes and denote continuation of execution after spawning children or synchronizing with children. There are connection constraints: a fork node can connect to a single child fragment; atleast one fragment connects to a join node; continuation edges can only connect fragments to fork and join nodes within the same context. Book-keeping nodes are followed by a chunk node when iterations remain and continue to a join node to synchronize with other threads otherwise. Chunk nodes always continue to a book-keeping

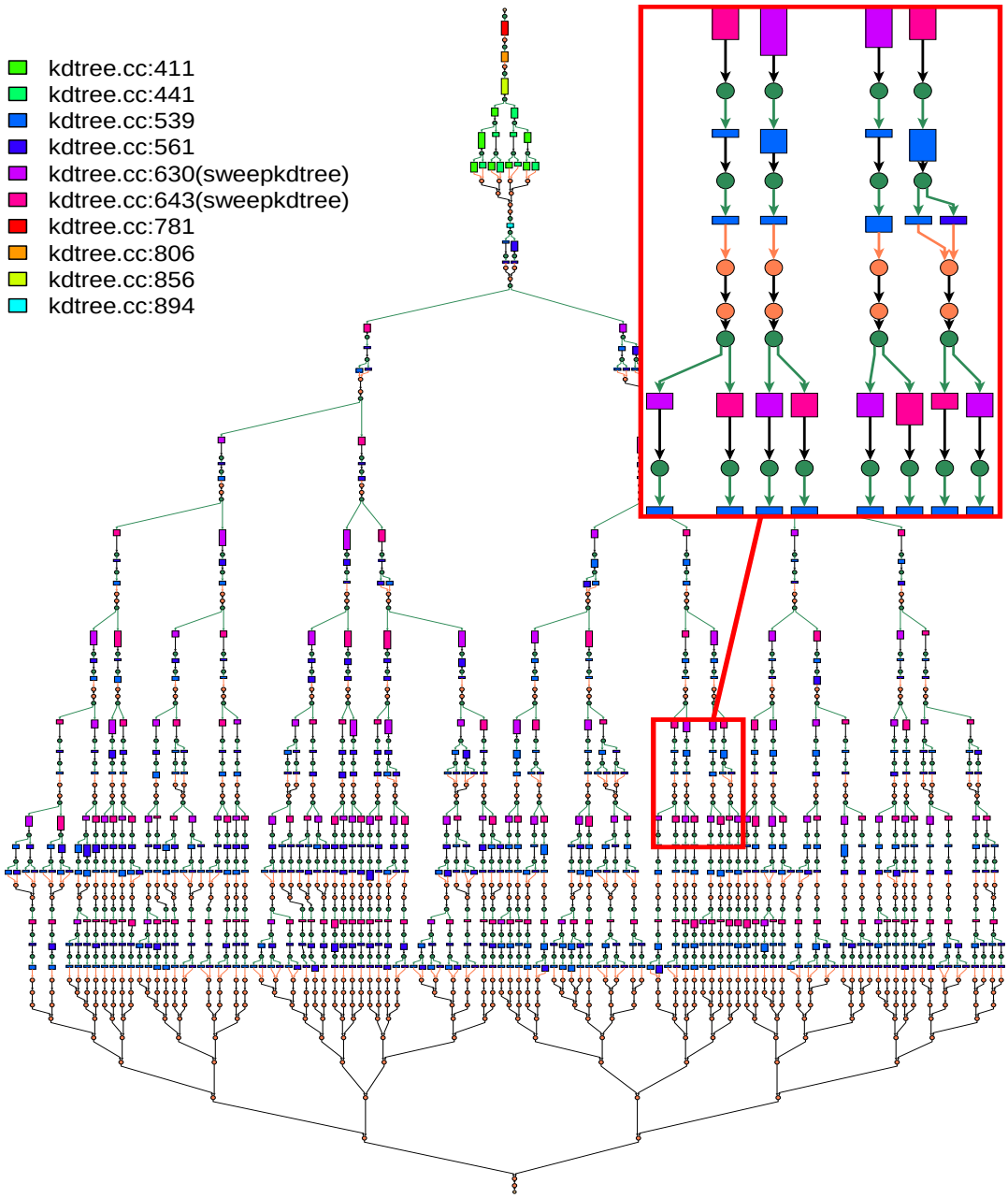

Figure 2: Grain graph of 376.kdtree for small input (tree size 200, radius 10, cutoff 2) containing 740 grains. Performance is lost due to many grains created by recursing to a large depth despite providing 2 as cutoff. The cutoff has no effect. The inset is a zoombox that zooms into a region of interest on the graph for clarity.

node. Edges never cross to ensure child fragments appear local to the parent and fragments of a task are aligned in sequence – essential features to convey recursive task creation.

We apply reductions to the graph structure by grouping nodes to speedup rendering times (Figure 3d-e, h). Grouped nodes retain weights of individual member nodes and also aggregate them. We group all book-keeping nodes per thread. Additionally, chunks are depicted as siblings since they are executable in parallel by definition.

Performance metrics are encoded as visual properties (size, color, shape, etc.) of graph elements. Grains are drawn as rectangles with length linearly scaled to execution time and fill color reflecting severity of the problem. Both edges and node borders are colored red if they are on the critical path of the grain graph.

Unique identification of grains is necessary for comparing graphs. Grains corresponding to tasks are identified using path enumeration which relies on the static nature of the graph for task-based programs. However, the path cannot identify grains from parallel for-loops since the shape depends on the number of threads and chunk assignment order. We identify chunks through the thread that started the loop, a sequence counter, and the iteration range.

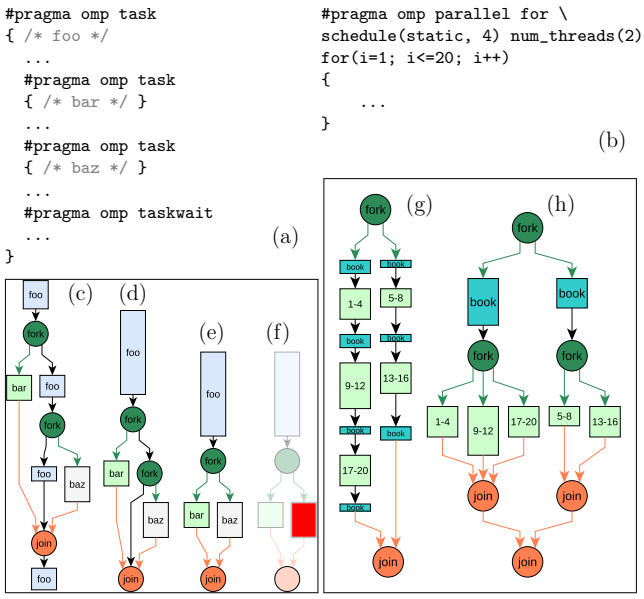

Figure 3: Grain graph structure. (a) Task-based OpenMP program. Task `foo` creates tasks `bar` and `baz`, performs computation in-between and synchronizes with the children tasks. (b) OpenMP program with parallel for-loop. Iteration space is divided into 5 chunks of size 4 and distributed evenly on two threads. (c) Grain graph captures task creation and synchronization during program execution. Performance of individual task instances are visually encoded so execution time is encoded as the length of fragments. Fragments of `foo` are sequentially aligned and child fragments are drawn local to the parent context to preserve the program perspective. (d-e) Reductions group nodes to reduce the size of the graph for quick rendering. Fragment reduction combines task fragments (d) and fork reduction combines fork nodes before every join node (e). Grouped nodes retain and aggregate performance weights of individual member nodes. After reductions, nodes are laid out symmetrically for space-efficiency. (f) Problematic grains are highlighted with a color that reflects severity and remaining elements are dimmed. (g) Execution of for-loop on two threads. Turquoise nodes represent book-keeping for delivering chunks to threads. Green rectangles show chunk execution labeled with the iteration range. The orange circle represents joining when all chunks are done. (h) Reductions group book-keeping nodes per-thread.

The starting thread is constant in programs without nested parallelism.

3.2 Metrics

We automatically derive metrics from the graph structure to detect problems. We annotate the graph with standard metrics that include the critical path of the graph and various memory system behavior, such as cache miss ratios and *memory hierarchy utilization* – a ratio of processor cycles spent performing computation to stalled cycles waiting for data. The derived metrics are:

Parallel benefit: is a grain's execution time divided by the parallelization costs borne by the grain's parent. The metric aids inlining and cutoff decisions by quantifying whether parallelization is beneficial so grains with low parallel benefit should be executed serially to reduce overhead. Parallelization cost of a grain is the sum of its creation time and average time spent by the grain's parent in synchronizing with all siblings. Parallelization cost for chunks uses book-keeping cost instead of child creation time.

Load balance: is the ratio between the length of the longest grain and the median length of all chains of consecutive grains in the unreduced graph. Load balance in Figure 3g is the ratio of the length of longest grain `9-12` to the median length of the two chains. Load balance much greater than one indicates presence of atleast one grain whose work time approaches the makespan of the parallel section. When approximately equal to one, the metric indicates that the load on the cores is balanced. The load balance metric helps understand when `chunk size` is too large and the parallel benefit metric when `chunk size` is too small.

Work deviation: is the change in execution time between single core and multicore grain execution. Work deviation is beneficial when it is less than one and problematic when it is greater than one. Work deviation below one typically happens when the working set fits in the private cache under multi-core execution. Olivier et al. [30] coined the name *work inflation* to refer to increased computation time when going from single-threaded to multi-threaded execution for the whole program. We compute work deviation per grain and refer to problematic work deviation as work inflation.

Instantaneous parallelism: is parallelism exposed by the program at different times during execution. Low instantaneous parallelism means cores idle because no work is available. An abundance of instantaneous parallelism with good parallel benefit is desirable for performance. The metric is calculated by counting the number of grains whose execution overlaps with intervals of program execution time. Interval size is a balance between accuracy and post-processing time. We provide the minimum grain length, the smallest difference between when a grain starts and another grain ends, and the median grain length as default choices. The metric comes in two flavors: *optimistic* includes all grains with any overlap of the interval, and *conservative* only includes grains with full overlap. Instantaneous parallelism of a grain is the smallest instantaneous parallelism among all its overlapping time intervals.

Scatter: is the median pair-wise distance in the system topology between cores executing sibling grains. Distances are obtained from the NUMA distance table or by subtracting core identifiers in some topologies. High scatter between grains that share data can lead to poor memory hierarchy utilization.

3.3 Highlighting Problems

Derived metric values that are likely to be problematic are highlighted (red grain in Figure 3f) and also made available in a summary form. We highlight 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.

4. Performance Analysis with Grain Graphs

We demonstrate how the grain graph enables programmers to quickly diagnose performance problems and find portable solutions to improve performance in OpenMP programs. We first indicate programs and evaluation methods used to test the effectiveness of grain graphs before diving into the performance analysis.

4.1 Programs

We profiled all C/C++ OpenMP programs except one in three standard benchmark suites – SPEC OMP 2012 [29] (SPEC-OMP), Barcelona OpenMP Task Suite [14] version 1.1.2 (BOTS), and

Parsec [5] version 3.0 (Parsec). Our aim is to use grain graphs to explain and improve poor performance so we characterize and pinpoint performance problems at the level of grains and link them back to source code which has not been done before. We give a detailed analysis of 359.botsspar and 376.kdtree from SPEC-OMP, Freqmine from Parsec, and Sort, Strassen, and FFT from BOTS. We conclude the section with a brief discussion of the metrics and performance of the remaining programs to demonstrate that performance analysis with grain graphs works for programs with good performance as well. We omit analysis of 352.nab in SPEC-OMP because the profiler does not support nested parallelism.

4.2 Evaluation Details

We used the MIR profiler [28] to obtain raw per-grain performance and properties required to construct the grain graph and derive metrics listed in Section 3.2.

The MIR profiler collects raw performance information with low overhead from hardware performance counters during grain events notified by the MIR runtime system [27]. The MIR runtime system supports OpenMP 3.0 tasks and parallel for-loops and notifies grain events based on a superset of the OMPT interface [16] that includes parallel for-loop chunk events and affinity information. The MIR runtime system uses a state-of-the-art work-stealing scheduler with lock-free task queues [8] by default and is implemented as a drop-in replacement for *libgomp*, the GCC OpenMP runtime system. Hardware performance counters are accessed through PAPI [26] to measure grain execution time and memory behavior statistics such as L1 cache misses and memory stall cycles. Less than 2.5% overhead is incurred by the MIR profiler to determine grain properties and hardware performance counts. Although we have used the MIR profiler, the grain graph visualization works irrespective of the profiling method.

Raw information from the MIR profiler was used in a post-profiling step to construct grain graphs and derive metrics. We used the *igraph* [10] package to construct the grain graph and annotated it with performance information. The grain graph is stored as a GRAPHML file that is viewable on off-the-shelf, large-scale graph viewers such as yED [45] and Cytoscape [40]. Grain graphs in the paper were viewed on yEd.

We extracted the programs from their benchmark suite ecosystems into standalone versions while retaining compiler and runtime options. We converted parallel for-loops with implied or explicit schedule `static` clauses to schedule `runtime` clauses and set the environment variable `OMP_SCHEDULE=static` to profile chunks from within the runtime system. For-loops with static schedules without this modification make the compiler insert chunk assignment code directly into the program. Our conversion affected performance by at most 3% on ICC.

Programs were compiled with GCC v4.9.2 and ICC v15.0.1 using optimization flags `-O3` except for BOTS which used `-O2` for ICC. Cutoffs were chosen for best performance on a 48-core test machine with 64GB memory and four 2.1GHz AMD Opteron 6172 processors with frequency scaling disabled. We used the *manual cutoff* version of BOTS programs.

We demonstrate that problems pin-pointed by our method are general problems by comparing program performance before and after optimization on other runtime systems. We used the median execution time of ten trials of the timed sections for comparison on ICC v15.0.1, GCC v4.9.2, and GCC v4.9.2 linked with MIR.

The grain graph based visual performance analysis work-flow is as follows. The grain graph has multiple views with colors encoding a single problem or property per view. Problematic grains, i.e., those that have crossed thresholds, are highlighted and other elements are dimmed in views where grain colors encode problems. Programmers shift views to understand problem areas to tackle. Clicking on a grain displays its timing, source location, and other properties in a separate window. Problem thresholds have sensible defaults listed in Section 3.3 and can be refined by programmers.

4.3 Analysis and Optimization

We demonstrate performance analysis using grain graphs on poor performing OpenMP programs that we optimize in the section.

4.3.1 Sort

Sort is a recursive fork-join task-based program from BOTS that sorts an array using divide-and-conquer in three phases. The first

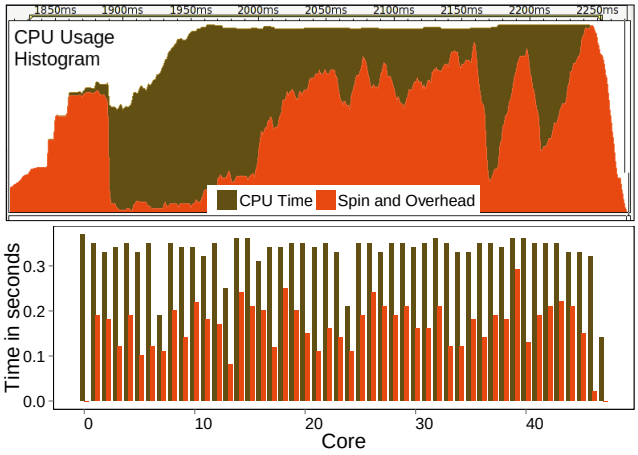

Figure 4: Existing visualizations show load imbalance and offer no actionable information about Sort performance. Intel VTune Amplifier shows cores performing uneven work and spending a large fraction of time inside the runtime system during execution of the parallel region. Nothing links the load imbalance to the culprit tasks. Visualizations in other tools suffer from the same problem.

phase uses parallel merge-sort, the second phase uses sequential quick sort, and the third uses sequential insertion sort. Phase shifts occur when the size of the divided array reaches thresholds specified by cutoffs which are crucial for performance [13, 28, 33].

Sort scales poorly for an input array with 16M elements on all runtime systems for the best cutoff values (Figure 1). Visualizations in existing tools point to load imbalance (Figure 4) as the reason for the poor performance and offer no further insight.

The reason behind Sort's load imbalance becomes clear from the grain graph in Figure 5. Sort exposes non-uniform parallelism. A lot of parallelism is available when the program starts executing, but the amount of parallelism gradually decreases in a waxing and waning pattern. The crux is that instantaneous parallelism is less than the number of cores available (48) at several points during execution.

We reduced the depth cutoff to increase instantaneous parallelism, but grains became too small and performance decreased due to low parallel benefit (Figure 5b). We can conclude that sorting 16M elements with the best cutoffs will always have load imbalance irrespective of scheduling.

Sort has additional problems that can be seen on the grain graph in a mutually exclusive manner. We summarize the problems and optimization results in the table below for space reasons:

	Affected grains (%)	
Problem	**Before**	**After**
Work Inflation	68.54	37.08
Poor Memory Hierarchy Utilization	56.05	30.11

We reduced both problems with round-robin memory page distribution to different NUMA nodes. Performance improved on all

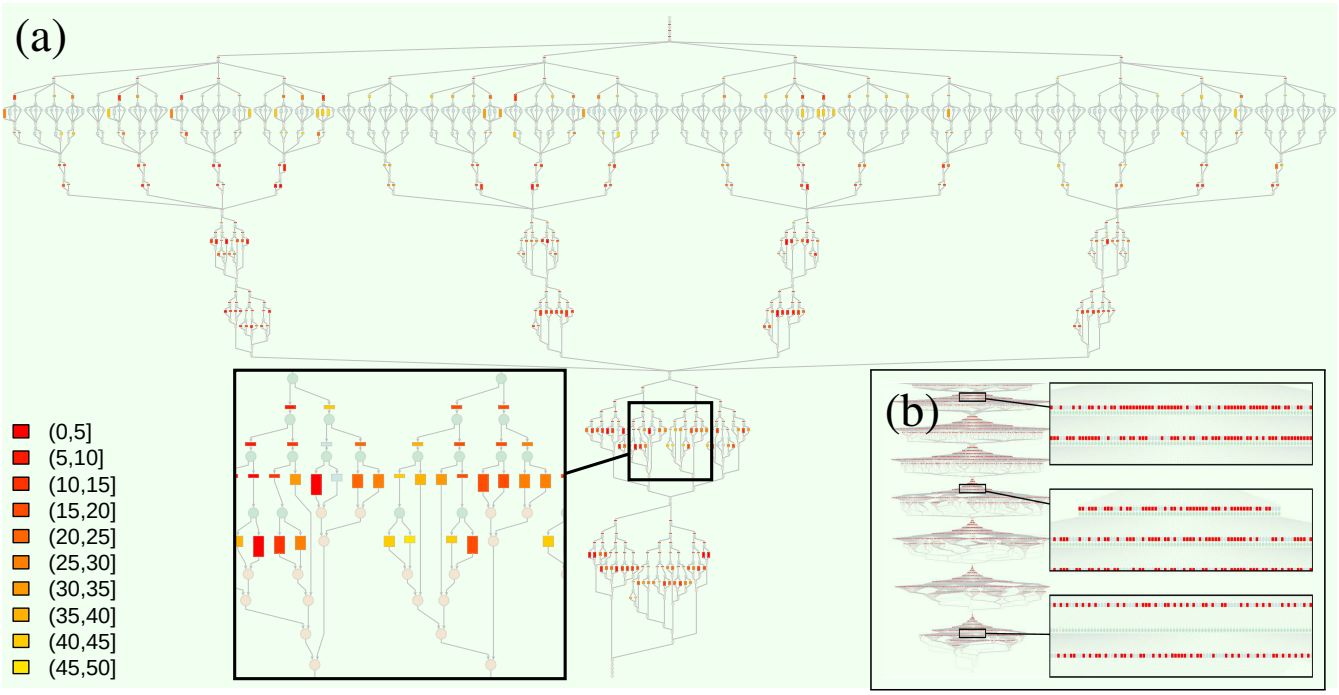

Figure 5: Sort grain graph. (a) Low instantaneous parallelism causes load imbalance. Phases with decreasing and non-uniform parallelism can be seen on the graph. Problematic grains are highlighted and others are dimmed. The highlighted grains have lower instantaneous parallelism than the 48 available cores on our machine. The highlight indicates the amount of instantaneous parallelism exposed using a red-to-yellow heatmap. The grain graph contains 815 grains. (b) Increasing instantaneous parallelism by lowering cutoffs reduces parallel benefit and does not improve performance. Low parallel benefit is highlighted in a red-to-yellow (red is low) linear color gradient. Grains without the problem are dimmed. Only lower half of graph shown for space reasons. Entire graph contains 18373 grains, 48% with low parallel benefit.

runtime systems as shown in Figure 1. Further improvements can come from algorithmic changes, access-pattern-aware data distribution and locality-aware scheduling.

4.3.2 359.botsspar

359.botsspar is an iterative task-based program from SPEC-OMP to compute the L-U factorization of a sparse matrix. The program scales poorly with an input matrix with 60X60 blocks of size 250X250 shown in Figure 1. The input is half of SPEC specified *test* input and is chosen for space reasons in the paper.

Figure 6a shows 359.botsspar exposing non-uniform and gradually decreasing parallelism in two distinct interleaved phases. The first phase consists of grains from `sparselu.c:229(fwd)` in light-green and `sparselu.c:235(bdiv)` in orange and exposes less parallelism than the second phase. The second phase exposes large amounts of parallelism from `sparselu.c:246(bmod)` (magenta grains in Figure 6a). Both gradually reduce the amount of parallelism and therefore fail to take advantage of all 48 cores on our machine.

359.botsspar has a more severe bottleneck than gradually reducing parallelism – work inflation. This is known since Olivier et al. [30] reported work inflation at program-level granularity for BOTS SparseLU, the ancestor of 359.botsspar. We gradually lower the work deviation problem threshold from 2 to 1.2 and see that many tasks suffer from work inflation (Figure 6c). Visualizations in existing tools do not flag any problems since load is balanced and tasks are sufficiently large. Olivier et al. found that enabling inter-procedural optimizations with ICC minimize work inflation and improve performance upto 60% during parallel and upto 3X for sequential execution. The drawbacks are being tied to ICC for these improvements and that culprit tasks cannot be discerned.

By sorting task definitions by creation count and work inflation, the graph pin-pointed `sparselu.c:246(bmod)` as the culprit. These grains were most frequent since they contribute work solely to the phase that exposes the largest amount of parallelism (magenta colored tasks in Figure 6a). Since `sparselu.c:246(bmod)` had similar work inflation to other tasks (Figure 6c), programmers could focus on optimizing that first.

We decided to improve the cache behavior since the graph also highlighted poor memory hierarchy utilization for the same task. Cache misses and coherence latency are the main sources of work inflation [30]. The body of `sparselu.c:246(bmod)` called `bmod` which had a triple-nested loop with a cache-unfriendly access pattern. We performed manual loop interchange to get a more cache-friendly access pattern which reduced work inflation (Figure 6d) and improved performance for all runtime systems (Figure 1).

4.3.3 FFT

FFT is a recursive fork-join task-based program from BOTS that calculates the 1-D DFT of a set of complex-valued samples. The program uses divide-and-conquer to divide the samples into smaller sets before calculating the DFT. Many tasks are created even for small inputs since several tasks are created for each divide.

FFT scales badly on all except ICC for an input of 16M samples (Figure 1). Visualizations in existing tools show that load is balanced and provide no further information to understand and improve performance.

The grain graph reveals the main problem with FFT: most grains are too small to provide parallel benefit. Figure 7 contains a sum-

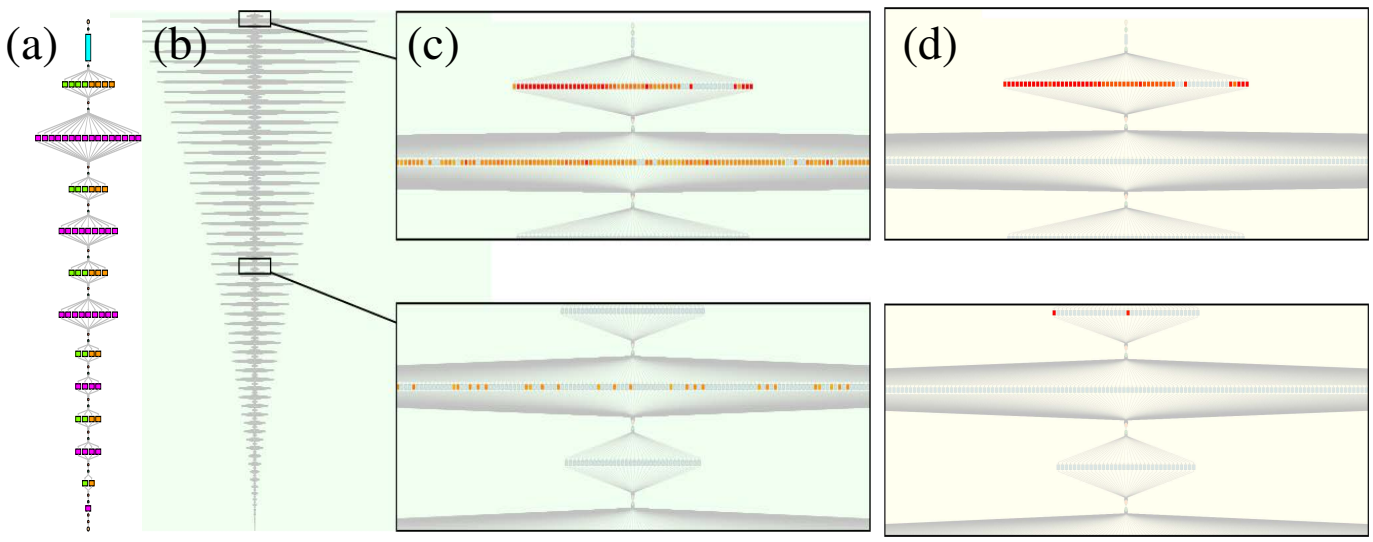

Figure 6: Grain graph of 359.botsspar. (a) 359.botsspar has two distinct, interleaved computation phases that expose gradually decreasing parallelism. The graph is for a small input (5,5). (b) Grain graph with evaluation input contains 19811 grains. Grains that suffer from work inflation are highlighted in a red-to-yellow (red is high) linear color gradient. Non-problematic grains are dimmed. (c) Performance is lost due to wide-spread work inflation (threshold set to 1.2). (d) Optimizations reduce work inflation. No work inflation is seen on grains in the larger parallelism phase. Work-inflated grains are isolated to the lesser parallelism phase.

mary of parallel benefit since 600 thousand grains can be displayed on a monitor but does not fit in this paper. Increasing parallel ben-

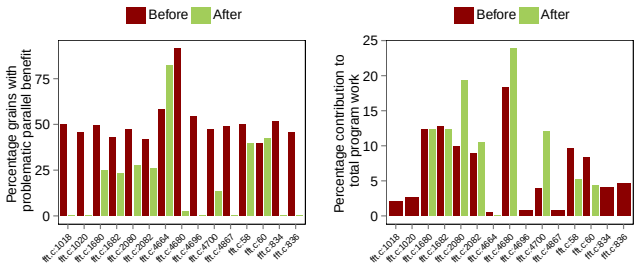

Figure 7: FFT performance grouped by definition in source files. Several grains have low parallel benefit in the original program. Grains show good parallel benefit after optimizations. Not all grains are created in the optimized program due to cutoffs.

efit is key to improving performance.

We began to optimize FFT to increase the parallel benefit. The graph singled out the first optimization candidate – `fft.c:4680`. Those grains have a high prevalence of poor parallel benefit and contribute most heavily to to total program work (Figure 7). We decided to increase the parallel benefit by adding cutoffs to prevent creation of too small grains. Inspecting `fft_aux` called solely in the body of `fft.c:4680` revealed an opportunity to add a cutoff based on recursion depth. Systematically inspecting other optimization candidates on the grain graph revealed similar opportunities to introduce cutoffs. The same cutoff could be used in several places which allowed us to reduce the number of cutoffs to two. Good values for the cutoffs were quickly obtained using performance and structural feedback from the graph.

FFT performance improves after optimizations as shown in Figure 1. All runtime systems benefit since tasks are big enough to provide good parallel benefit (Figure 7).

ICC performed well without optimizations, which led us to suppose that the ICC runtime system uses internal cutoffs to improve parallel benefit. We inspected the sources for version 15.0.1 [20] and found a queue-size based internal cutoff. GCC fares poorly despite limiting task creation at 64 times the number of threads [34].

Despite improving the parallel benefit, FFT fails to scale to a high degree. Figure 8 shows the graph for the optimized program which points out the next bottleneck. A majority of grains have poor memory hierarchy utilization. Since the problem is observed despite using a work-stealing scheduler, we can conclude that algorithmic changes and locality-aware scheduling (or any schedule better than work-stealing) are necessary to further improve FFT performance. Optimization focused on the critical path alone will not suffice since poor memory hierarchy utilization is wide-spread.

4.3.4 Freqmine

Freqmine is a parallel for-loop based program from Parsec that mines transactions for association rules using an array-based implementation of the FP-growth algorithm. The program takes a database and a minimum support and finds all frequent item-sets with support equal to or greater than the minimum support.

Freqmine performs poorly on all runtime systems under what Parsec calls *simlarge* inputs, `kosarac_990k.dat` as transaction database and 11000 as minimum support as shown in Table 1. Visualizations in existing tools show load is imbalanced while executing a dynamically scheduled parallel for-loop in function `FP_tree::FP_growth_first()`. We refer to the loop as *FPGF*.

Table 1: Freqmine performs poorly on all runtime systems due to the imbalanced FPGF loop. 7 cores are sufficient to maintain performance for the evaluation input. RTS is short for runtime system.

RTS	Speedup	48-core exec. time	7-core exec. time
ICC	6.58	1.71s	1.72s
GCC	6.68	1.68s	1.69s
MIR	7.2	1.65s	1.68s

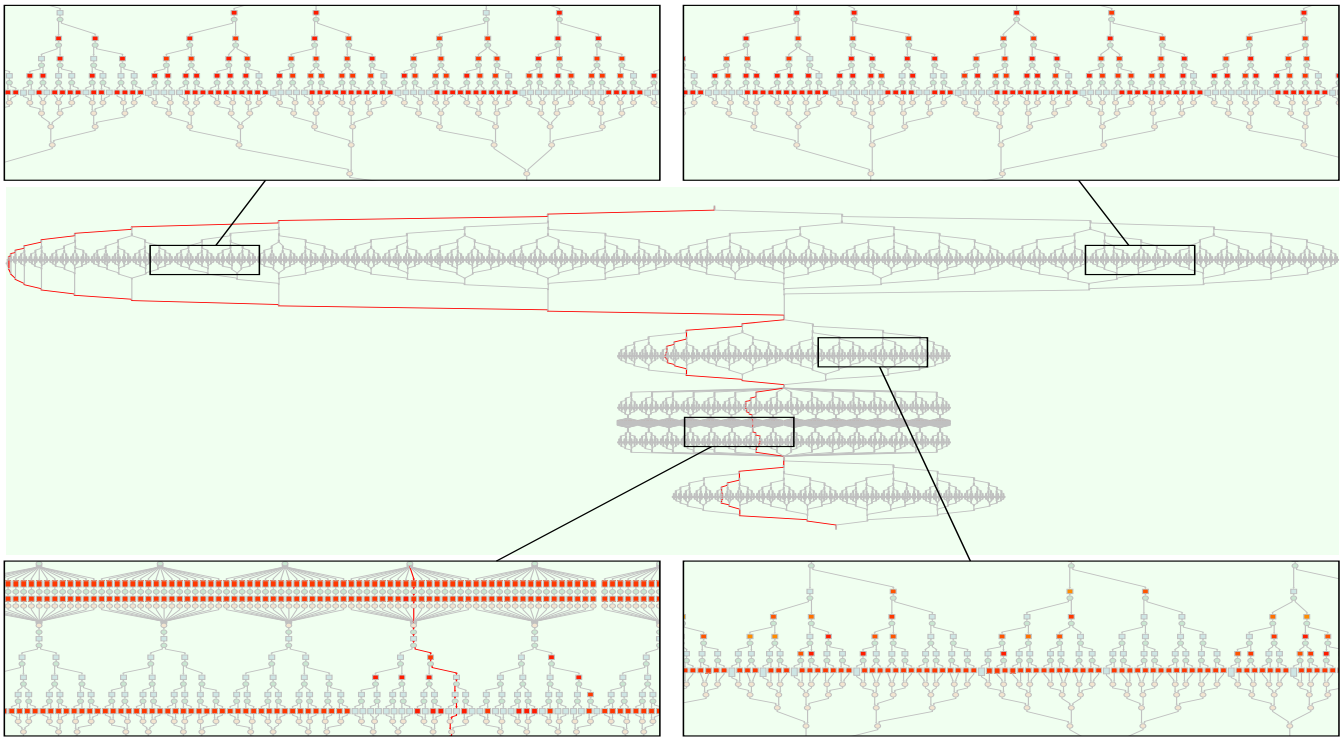

Figure 8: Grain graph of FFT shows the next problem to be tackled. Several grains have poor memory hierarchy utilization highlighted in a red-to-yellow (red is low) linear color gradient. Non-problematic grains are dimmed. Algorithmic changes and better scheduling are necessary to further improve performance. Grain graph has 4591 grains.

The loop is instantiated thrice and the second instance takes up 70% of the program execution time.

We first focused on improving the load balance of FPGF. Traditional wisdom advocates creating more chunks by reducing the `chunk size` but that is already set to the smallest value, one.

Existing tools also show that FPGF has high synchronization cost for most cores. Traditional wisdom advocates making synchronizations more infrequent by increasing the `chunk size`. This seemed reasonable considering the load imbalance was already bad. However, increasing the `chunk size` even by a small amount increases the load imbalance. At this stage, we looked for opportunities to speedup FPGF's body. Unfortunately, the README notes that functions called from the body are inherently sequential and algorithmic optimizations are unlikely to be revealed by tools.

Our method reveals the root cause of the problem: grains of FPGF have uneven size, as shown in Figure 10. Most grains are small and provide poor parallel benefit. Only a few grains are large. Inspection reveals that the large grains execute single loop iterations that are spaced irregularly across the iteration range and not isolated to a particular portion of the iteration range causing some cores get more work than others due to the greedy nature of the dynamic schedule. The many small chunks cause the high synchronization overhead indicated by existing tools. We increased the `chunk size` to improve parallel benefit, but that made the load balance even worse.

That FPGF is bound to have load imbalance presented us with an opportunity to optimize resource usage instead. We used a straight-forward bin-packer implemented in Gecode [42] to compute the minimum number of cores necessary to retain the same makespan – 7 cores. We set `num_threads` to 7 for the instance in the source code and show the resulting load balance with similar execution time in Figure 10 and Table 1.

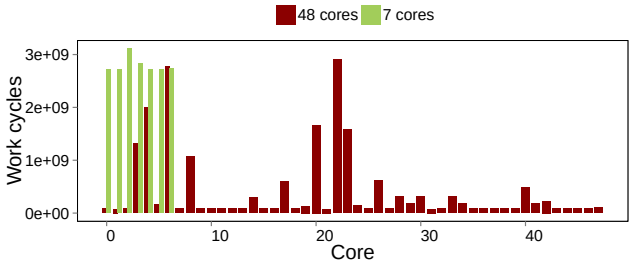

Figure 10: Load balance of second instance of loop FPGF which contains 1292 chunks of disproportionate size. Load balance is 35.5 on 48 cores and improves to 1.06 on 7 cores.

4.3.5 Strassen

Strassen is a recursive fork-join task-based program from BOTS to multiply matrices using the Strassen algorithm. The matrices are decomposed recursively into smaller submatrices and multiplication is performed at the leaves of the recursion on the smallest submatrices. The amount of recursive decomposition is controlled by a cutoff for the smallest submatrix size. We refer to the submatrix size cutoff as SC for the remainder of the section.

Strassen scales poorly on all runtime systems (Figure 1, input: 8192 X 8192 matrix, SC = 128). Visualizations in existing tools show that the load is imbalanced, and remains imbalanced despite lowering SC. The behavior contradicts the intuition that balance should improve when more tasks are created. Increasing SC does

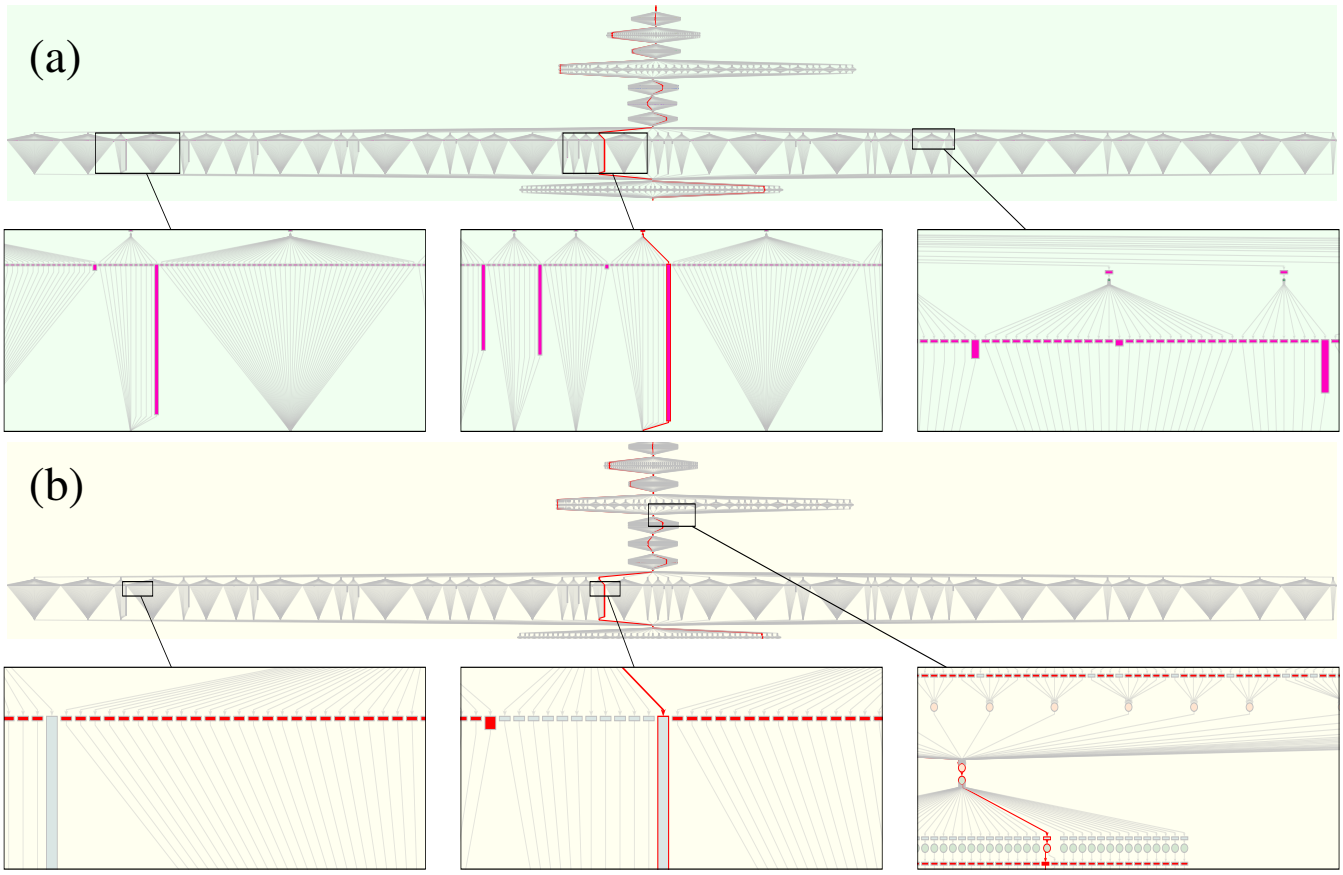

Figure 9: Grain graph of Freqmine with evaluation input contains 6985 grains. (a) The large magenta grains from for-loop in `FP_tree::FP_growth_first()` give bad load balance of 35.5. Grain colors encode for-loop definition in source code. (b) Most grains are too small and provide poor parallel benefit highlighted in a red-to-yellow (red is low) linear color gradient. Grains without the problem are dimmed. Poor parallel benefit also seen in other loops.

not worsen the balance either, which is surprising since tasks become larger.

The reason for the load imbalance becomes clear when we compare grain graphs for different SC values. All graphs are shallow and look the same (Figure 11a) indicating the cutoff has no effect.

Strassen suffers from a similar problem as 376.kdtree. We found a hard-coded cutoff that overrides SC and limits the exposed parallelism in the functions for matrix decomposition. Performance improves without cutoff on all runtime systems (Figure 1) since that provides sufficient parallelism for our machine (Figure 11a).

However, a new problem surfaces – poor memory hierarchy utilization (Figure 11b). Olivier et al. [30], who identify the same problem at a program-level granularity, catalog fixes that include using standard blocked matrix multiplication algorithm for multiplying submatrices at recursion leaves and placing data hierarchically using Morton ordering [43].

Strassen also performs poorly (48-core speedup of 10) with a central queue-based task scheduler. Sibling tasks are scattered under central queue scheduling (Figure 11d). Recall that scatter is highlighted when tasks are executed far away from each other (farther than 12 cores, i.e., off-socket on our machine) since communication and data fetching off-core takes more time. A work-stealing scheduler reduces scatter by adding children to the front of a local queue and other workers steal from the back of that queue. Scatter under work-stealing scheduling is shown in Figure 11c.

4.3.6 Other benchmarks

We summarize the performance analysis with grain graphs on the remaining programs. We group programs based on speedup with MIR over single core execution with ICC on the 48-core machine.

Speedup over 30: Over 65% of chunks of the sole parallel for-loop in Blackscholes (input 4 million points) have poor memory hierarchy utilization. Around 33% of the chunks also have low parallel benefit. Other metrics indicate good behavior. The other possible problems are serial sections.

367.imagick contains for-loops with poor parallel benefit (`magick_shear.c:1694`, `magick_decorate.c:406`, `magick_enhance.c:3554`, `magick_shear.c:1474` and `magick_transform.c:650`). These loops miss conditional for-loop throttling macros called `omp_throttle` present elsewhere. Our method points out these inconsistencies. The input we tested for 367.imagick is the chain `-shear 31 -resize 1280x960 -negate -edge 14 -implode 1.2 -flop -convolve 1,2,1,4,3,4,1,2,1 -edge 100` which is poor performing according to documentation by SPEC.

372.smithwa (input 34) parallel blocks `mergeAlignment.c:160` and `verifyData.c:46` suffer from load imbalance, low memory hierarchy utilization and poor parallel benefit. The load imbalance in `verifyData.c:46` is not visible in timings since

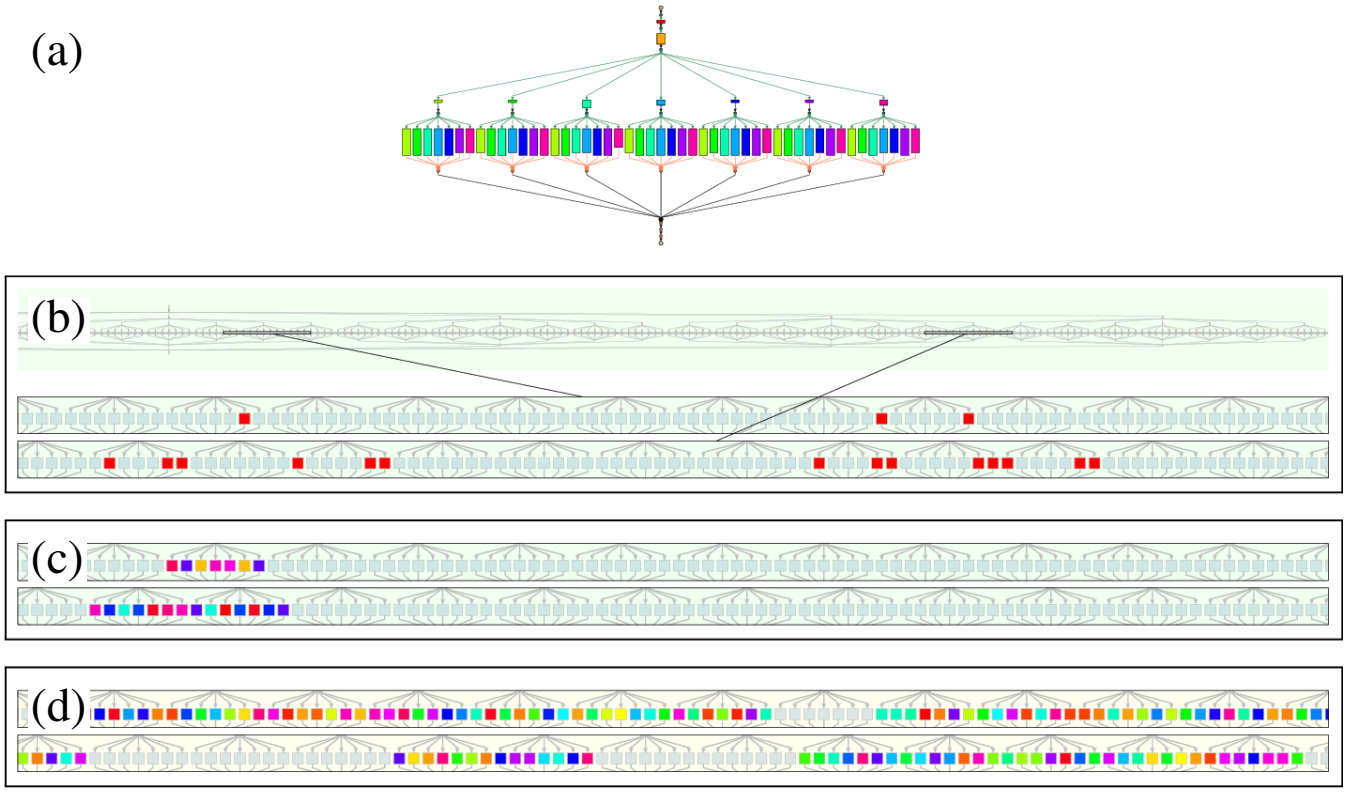

Figure 11: Grain graph of Strassen for small input (2048X2048). (a) Strassen contains a hard-coded cutoff that limits performance since depth is shallow irrespective of program input causing insufficient parallelism to be exposed for our 48-core machine. The graph is limited to 58 grains. Grain colors encode task definition in source code. (b) Many more tasks are seen at recursion leaves, exposing increased parallelism and improving performance when the hard-coded cutoff is disabled. The graph contains 2801 grains. Only the right half of the graph is shown for space reasons. The problem of poor memory hierarchy utilization comes to fore. Grains with poor memory hierarchy utilization are highlighted using a red-to-yellow (red is low) linear color gradient that encodes the relative intensity of memory hierarchy utilization. Non-problematic grains are dimmed. (c) Scheduler choice is crucial for performance. Few grains are scattered with a work-stealing scheduler. Problematic scatter is highlighted using a violet-to-red (rainbow) color gradient that encodes the executing core. Non-problematic grains are dimmed. (d) Many tasks are scattered when a central queue-based scheduler is used.

the region of interest excludes this blocks. Our method shows the problem since the graph represents the whole program.

NQueens (input 14) and 358.botsalgn (input `prot.200.aa`) scale linearly and all metrics indicate good behavior.

Fibonacci is a common example for illustrating task-based OpenMP programming and metrics for input 48 with cutoff 12 indicate problems in work deviation and parallel benefit. The grain graph immediately demonstrates how depth cutoffs control recursion depth and amount of computation performed by leaf grains which commonly do brunt of the work in task-parallel programs.

Speedup less than 20: UTS (test.input) suffers from poor parallel benefit for most of the 4 million grains. Despite the sheer size the graph still captures the imbalance after zooming. UTS can benefit from inlining within the runtime system or depth-based cutoffs.

In Bodytrack, chunks of parallel for-loops in all functions except `ParticleFilterOMP::CalcWeights()` suffer from poor parallel benefit and low memory hierarchy utilization. Loop fusion might improve the scaling and the metrics point out good candidates such as loops in `FlexFilterRowVOMP()` and `FlexFliterColumnVOMP()`. Serial sections are also bottlenecks.

Floorplan is a branch-and-bound optimal solution search that has non-deterministic behavior built-in due to pruning of the search space [14]. This behavior is reflected by the grain graph since the shape of the graph changes for different thread counts.

5. Related Work

Numerous performance visualization techniques and performance analysis tools [17, 25] have been developed since 1997 when OpenMP version 1.0 was released. We focus on techniques and tools that visualize OpenMP execution.

Tools predominantly visualize OpenMP execution using thread time-line and function call graph structures. Task execution is depicted on the structures from the perspective of the runtime system where tasks execute interleaved with other tasks in a runtime-optimized manner. The unfamiliar nature of the interleaving makes connecting performance back to program structure impractical for programmers. Parallel for-loop performance is similarly difficult to analyze since it is aggregated at the thread-level without distinction between chunks. The disconnect between performance visualization and program structure is a long-standing problem that was echoed more than a decade ago by the creators of DMPL [9] and continues to persist in recent tools [37]. We follow with a detailed

discussion on problems with visualizing tasks and chunks on thread time-lines and function call graphs and compare with our method.

Thread Time-line Visualization: The thread time-line shows state transitions of individual threads as they happen. Several tools including Intel VTune Amplifier, Vampir [7], Paraver [32], hpctraceviewer of HPCToolKit [1], Aftermath [12] and GOMP Profiler [2] visualize time-lines. Most tools use built-in profiling infrastructure. Exceptions include Vampir that uses traces from Score-P [22] or Scalasca [19], and Paraver that uses traces from OmpSs-Extrae [15]. The tools differentiate tasks and for-loop execution on the thread time-line as states separate from other thread states with the exception of VTune Amplifier. Time and hardware events spent in individual states can be aggregated, as seen in the histogram feature in Paraver. Task and for-loop load balance can be determined per thread by aggregating time spent in their particular execution states. It can be globally understood that tasks or chunks are too fine grained by checking if aggregated parallelization time exceeds aggregated execution time. Identifying large grained chunks is not possible using visualizations in existing tools.

Parallelism trends such as instantaneous parallelism are not captured by the time-line visualization since it restricts itself to number of the threads and manual inference is prevented since parent-child task connections are typically not shown. Ding et al. [11] show parent-child relationship on the time-line by super-imposing edges between parent and child tasks. Their visualization does not however distinguish task fragments and connects parent and child tasks using cyclic and redundant edges, producing convoluted graph layouts. Servat et al. [38] connect tasks to parent on the time-line without capturing proximity in code.

Task identifiers help in understanding scheduling action and to compare parent and children to infer parallel benefit, but are not shown on the time-line. Exceptions are the GOMP profiler and a research prototype [36] of Vampir that show unique identifiers for tasks. Parents cannot be identified from task identifiers in the Vampir research prototype. The GOMP profiler captures both task and parent identifiers in a table associated with the time-line. No tools provide schedule independent task identifiers which are essential to calculate work deviation per task.

Computing execution time per task from the time-line is impractical because tasks are spread out and interleaved on different time-lines since threads are free to schedule and run other tasks. Fragments have to be individually tracked and aggregated to infer per-instance time – an error-prone and tedious step in the presence of thousands of tasks typically created by task-based OpenMP programs. The time-line aggregation function does not work at a per-task instance level. Aggregating other per-task instance metrics such as hardware performance counter events is similarly impractical for typical programs with thousands of tasks.

None of the time-lines we have seen highlight the critical path through tasks or for-loops. The critical path is an important filter for selecting first-optimization candidates. Sampling-based tools such as VTune and hpctraceviewer show frequently sampled regions of code, *hotspots*, on the time-line but there is no guarantee that hotspots appear on the critical path. Scalasca [19] identifies and visualizes the critical path for MPI programs.

Function Call Graph Visualization: The function call graph is a tree-like visualization that shows how functions call each other. Fewer tools provide call graph visualization in comparison to the widely available time-line visualization. The call graph is useful to understand how functions performed under sequential execution but ill-suited to depict task/parallel for-loop execution since the instantiation and execution of these constructs are decoupled and subject to runtime-optimizations unlike function calls which execute immediately when called.

Call graphs typically show that tasks execute within the context of task synchronization calls to the runtime system. Tasks are typically not differentiated from other functions and tasks executing within the synchronization context are not necessarily children of the caller. Parent-child relationships cannot be inferred from the call graph so a time-line graph is required.

Task execution within synchronization contexts on call graphs are difficult to follow since they embody runtime-optimizations. For example, the VTune call graph requires runtime system knowledge to infer that tasks are inlined and executed immediately or deferred for later execution based on internal cutoffs. In addition, the synchronization context turns into a deeply-nested task tree which is difficult to navigate since OpenMP tasks can create children of their own typically in a recursive manner. Inferring parallelism trends from the call graph is hard.

Task visualization on call graphs is not uniform across tools. The CUBE call graph provided by Scalasca [19] aggregates task computation time to the definition in the program. Individual task instances are not differentiated and average task duration has to be estimated through visit counts. OmpP [18] has a textual call graph with execution time of task definitions but not per instance. The call graph in the Oracle Solaris Studio performance analyzer does not explicitly label tasks, instead functions called by tasks are shown. This complicates calculating exclusive task execution time since tasks and ordinary functions are not differentiated. The VTune call graph does not constantly distinguish tasks from functions due to runtime system inlining optimizations. The call graph visualized by Vampir [7] provides the highest information by aggregating execution time and call counts of task definitions by depth, but does not differentiate individual task instances.

Parallel for-loops are retro-fitted as special functions on the call graph similar to tasks. Computation time, scheduling overhead, and iteration counts of all loop instances are aggregated to the definition in the program with the exception of the latest 2016 version of VTune which separates loop instances. No call graph visualization in existing tools distinguishes chunks or functions called within loops.

Task Graph Visualization: DAGs are useful to visualize dynamic program behavior. Few performance analysis tools visualize tasks in DAGs.

Temanejo [6] visualizes data-flow execution of OmpSs and OpenMP 4.0 tasks in a task dependence graph and uses it as a graphical debugging aid. Tasks on the dependence graph do not have problems highlighted but they can be selected for execution in a step-by-step manner. It is unclear if the dependence graph shows execution time and other performance properties of tasks.

Tareador is an OmpSs tool that visualizes data-flow execution of OmpSs/OpenMP 4.0 tasks in a static task dependence graph [4, 41]. Task execution time and other performance information are not available as annotations and problems are not highlighted. The Flow Graph Designer [44] visualizes compile-time data dependencies between Intel TBB tasks in a graph. Performance of per-task instances created during execution are not available on the Flow Graph Designer.

Paradyn [24] uses dynamic instrumentation and automatically tests hypotheses to narrow down performance problems in long-running programs displayed in multiple DAGs grouped by re-

sources. Paradyn's notion of tasks is different from tasks in OpenMP [3].

The recent OMPT interface [16] is a good step towards enabling per-task performance visualization tools for OpenMP. No tool to the best of our knowledge uses OMPT events to visualize task graphs yet.

Our method complements thread time-lines, function call graphs and existing task graph visualizations by zooming into execution at the grain level. We visualize problems in a single graph that shows performance of all grains. Trends in parallelism can be immediately inferred and grain performance is readily available on the graph along with problem highlights. Programmers can connect execution to program structure since children are depicted close to the parent. The graph structure is robust under runtime system optimizations such as task inlining. We generalize load balance to include tasks and display work deviation per-grain. Metrics such as parallel benefit, instantaneous parallelism, and scatter are absent in other tools.

Cilkprof [35] is a low-overhead profiler for Cilk programs that textually quantifies the parallelism contributed by *call sites* – task definitions. Our method works at the finer resolution of grains, which are individual instances of task definitions, and graphically visualizes problems with several performance metrics including parallelism contribution (instantaneous parallelism metric) .

The disconnect between performance visualization and program structure is not isolated to OpenMP and until recently also troubled programmers of Charm++, a task-based programming model based on asynchronous messaging. Isaacs et al. [21] solve the problem by re-ordering Charm++ event traces to recover and visualize the logical structure familiar to programmers. They also derive and map metrics onto the logical structure visualization to aid performance analysis, similar to our approach.

The starting point for our work was the elegant graph notation used by McCool et al. [23] to explain parallelization patterns.

6. Conclusions

We have presented grain graphs, a method to visualize program execution from a predictable program perspective while retaining essential runtime system execution aspects such as scheduling and memory hierarchy performance. We demonstrated how our method guides programmers to understand and optimize poorly performing, inadequately understood standard programs.

Grain graphs are independent of profiling method and can be adopted to depict any control-flow that creates and synchronizes asynchronously with explicit, implicit or conceptual tasks.

We do not yet visualize OpenMP 4.0 data-flow tasks due to lack of data-dependence resolution support in the MIR profiler. There are no conceptual problems in extending our method to task dependence graphs when the profiler supports data-flow tasks. Similarly there are no conceptual problems to visualize the recently announced task-generating for-loops (version 4.5) once they are supported by the profiler. Large graphs have long rendering times, which is a scalability issue since programmer work flow is iterative. We have encouraging results from early experiments with collapsing collections of nodes and replacing them with a single summary node in the graph.

Acknowledgments

This work was partially funded by the Artemis PaPP Project (nr. 295440).

References

[1] 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*, 22(6):685–701, 2010.

[2] J. M. Arul, G.-J. Hwang, and H.-Y. Ko. GOMP profiler: A profiler for OpenMP task level parallelism. *Computer Science and Engineering*, 3(3):56–66, 2013.

[3] E. Ayguadé, N. Copty, A. Duran, J. Hoeflinger, Y. Lin, F. Massaioli, X. Teruel, P. Unnikrishnan, and G. Zhang. The design of OpenMP tasks. *Parallel and Distributed Systems, IEEE Transactions on*, 20(3): 404–418, 2009.

[4] 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.

[5] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: characterization and architectural implications. In *Proc. of the International Conference on Parallel Architecture and Compilation Techniques (17th PACT'08)*, pages 72–81. ACM, 2008.

[6] S. Brinkmann, J. Gracia, and C. Niethammer. Task debugging with temanejo. In *Tools for High Performance Computing 2012*, pages 13–21. Springer, 2013.

[7] 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*, number 4315 in LNCS, pages 5–14. Springer, 2008.

[8] D. Chase and Y. Lev. Dynamic circular work-stealing deque. In *Proceedings of the Seventeenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA'05, pages 21–28. ACM, 2005.

[9] J. Cownie, J. DelSignore, John, B. de Supinski, and K. Warren. DMPL: An OpenMP DLL debugging interface. In *OpenMP Shared Memory Parallel Programming*, volume 2716 of *LNCS*, pages 137–146. Springer, 2003.

[10] G. Csardi and T. Nepusz. The igraph software package for complex network research. *InterJournal*, Complex Systems:1695, 2006.

[11] Y. Ding, K. Hu, K. Wu, and Z. Zhao. Performance monitoring and analysis of task-based OpenMP. *PLoS ONE*, 8(10):e77742, 2013. doi: 10.1371/journal.pone.0077742.

[12] A. Drebes, A. Pop, K. Heydemann, A. Cohen, and N. Drach-Temam. Aftermath: A graphical tool for performance analysis and debugging of fine-grained task-parallel programs and run-time systems. In *7th Workshop on Programmability Issues for Heterogeneous Multicores (MULTIPROG, associated with HiPEAC), Vienna, Austria*, 2014.

[13] A. Duran, J. Corbalán, and E. Ayguadé. An adaptive cut-off for task parallelism. In *High Performance Computing, Networking, Storage and Analysis. SC'08. International Conference for*, pages 1–11, 2008.

[14] A. Duran, X. Teruel, R. Ferrer, X. Martorell, and E. Ayguade. Barcelona OpenMP tasks suite: A set of benchmarks targeting the exploitation of task parallelism in OpenMP. In *Parallel Processing, 2009. ICPP'09. International Conference on*, pages 124–131, 2009.

[15] A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas. Ompss: a proposal for programming heterogeneous multi-core architectures. *Parallel Processing Letters*, 21(02): 173–193, 2011.

[16] A. E. Eichenberger, J. Mellor-Crummey, M. Schulz, M. Wong, N. Copty, R. Dietrich, X. Liu, E. Loh, and D. Lorenz. OMPT: An OpenMP tools application programming interface for performance analysis. In *OpenMP in the Era of Low Power Devices and Accelerators*, pages 171–185. Springer, 2013.

[17] K. Fürlinger. OpenMP application profiling—state of the art and directions for the future. *Procedia Computer Science*, 1(1):2107–2114, 2010.

[18] K. Fürlinger and D. Skinner. Performance profiling for OpenMP tasks. In *Evolving OpenMP in an Age of Extreme Parallelism*, number 5568 in LNCS, pages 132–139. Springer, Jan. 2009.

[19] 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*, 22(6):702–719, 2010.

[20] Intel Corporation. OpenMP* Runtime to align with Intel Parallel Studio XE 2015 Composer Edition Update 3, 2015. https://www.openmprtl.org/download. Accessed 10 April 2015.

[21] K. E. Isaacs, A. Bhatele, J. Lifflander, D. Böhme, T. Gamblin, M. Schulz, B. Hamann, and P.-T. Bremer. Recovering logical structure from Charm++ event traces. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, ser. SC*, volume 15, 2015.

[22] D. Lorenz, P. Philippen, D. Schmidl, and F. Wolf. Profiling of OpenMP tasks with score-p. In *Parallel Processing Workshops (ICPPW), 2012 41st International Conference on*, pages 444–453, 2012.

[23] M. McCool, J. Reinders, and A. Robison. *Structured Parallel Programming: Patterns for Efficient Computation*. Access Online via Elsevier, 2012.

[24] B. P. Miller, M. D. Callaghan, J. M. Cargille, J. K. Hollingsworth, R. B. Irvin, K. L. Karavanic, K. Kunchithapadam, and T. Newhall. The paradyn parallel performance measurement tool. *Computer*, 28 (11):37–46, 1995.

[25] M. S. Mohsen, R. Abdullah, and Y. M. Teo. A survey on performance tools for OpenMP. *World Academy of Science, Engineering and Technology*, 49, 2009.

[26] P. J. Mucci, S. Browne, C. Deane, and G. Ho. PAPI: A portable interface to hardware performance counters. In *Proceedings of the Department of Defense HPCMP Users Group Conference*, pages 7–10, 1999.

[27] A. Muddukrishna, P. A. Jonsson, V. Vlassov, and M. Brorsson. Locality-aware task scheduling and data distribution on NUMA systems. In *OpenMP in the Era of Low Power Devices and Accelerators*, number 8122 in LNCS, pages 156–170. Springer, 2013.

[28] A. Muddukrishna, P. A. Jonsson, and M. Brorsson. Characterizing task-based OpenMP programs. *PLoS ONE*, 10(4):e0123545, 2015. doi: 10.1371/journal.pone.0123545.

[29] M. S. Müller, J. Baron, W. C. Brantley, H. Feng, D. Hackenberg, R. Henschel, G. Jost, D. Molka, C. Parrott, J. Robichaux, et al. Spec OMP2012—an application benchmark suite for parallel systems using openmp. In *OpenMP in a Heterogeneous World*, pages 223–236. Springer, 2012.

[30] S. L. Olivier, B. R. de Supinski, M. Schulz, and J. F. Prins. Characterizing and mitigating work time inflation in task parallel programs. In *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*, pages 1–12, 2012.

[31] OpenMP Architecture Review Board. OpenMP application program interface version 4.5, 2015. `http://www.openmp.org/mp-documents/openmp-4.5.pdf`.

[32] 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*, volume 44, pages 17–31, 1995.

[33] A. Podobas and M. Brorsson. A comparison of some recent task-based parallel programming models. In *Proceedings of the 3rd Workshop on Programmability Issues for Multi-Core Computers, (MULTIPROG'2010), Pisa*, 2010.

[34] A. Podobas, M. Brorsson, and K.-F. Faxén. A comparative performance study of common and popular task-centric programming frameworks. *Concurrency and Computation: Practice and Experience*, 27(1):1–28, 2015.

[35] T. B. Schardl, B. C. Kuszmaul, I. Lee, W. M. Leiserson, C. E. Leiserson, and others. The Cilkprof Scalability Profiler. In *Proceedings of the 27th ACM on Symposium on Parallelism in Algorithms and Architectures*, pages 89–100. ACM. URL `http://dl.acm.org/citation.cfm?id=2755603`.

[36] D. Schmidl, P. Philippen, D. Lorenz, C. Rössel, M. Geimer, D. a. Mey, B. Mohr, and F. Wolf. Performance analysis techniques for task-based OpenMP applications. In *OpenMP in a Heterogeneous World*, number 7312 in LNCS, pages 196–209. Springer, 2012.

[37] D. Schmidl, C. Terboven, D. a. Mey, and M. S. Müller. Suitability of performance tools for OpenMP task-parallel programs. In *Tools for High Performance Computing 2013*, pages 25–37. Springer, 2014.

[38] H. Servat, X. Teruel, G. Llort, A. Duran, J. Gimenez, X. Martorell, E. Ayguadé, and J. Labarta. On the instrumentation of OpenMP and OmpSs tasking constructs. In *Euro-Par Workshops*, volume 7640 of *LNCS*, pages 414–428. Springer, 2012.

[39] O. Sinnen. *Task scheduling for parallel systems*, volume 60. John Wiley & Sons, 2007.

[40] 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*, 27(3):431–432, 2011.

[41] 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*, 4(6):450 – 456, 2013. doi: http://dx.doi.org/10.1016/j.jocs.2013.01.008.

[42] G. Team. Gecode: Generic constraint development environment, 2006. `http://www.gecode.org`.

[43] M. Thottethodi, S. Chatterjee, and A. R. Lebeck. Tuning Strassen's matrix multiplication for memory efficiency. In *Proceedings of the 1998 ACM/IEEE conference on Supercomputing*, pages 1–14. IEEE Computer Society, 1998.

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

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