Parallelism-Centric What-If and Differential Analyses

Adarsh Yoga
Rutgers University, USA
adarsh.yoga@cs.rutgers.edu

Santosh Nagarakatte
Rutgers University, USA
santosh.nagarakatte@cs.rutgers.edu

Abstract

This paper proposes TaskProf2, a parallelism profiler and an adviser for task parallel programs. As a parallelism profiler, TaskProf2 pinpoints regions with serialization bottlenecks, scheduling overheads, and secondary effects of execution. As an adviser, TaskProf2 identifies regions that matter in improving parallelism. To accomplish these objectives, it uses a performance model that captures series-parallel relationships between various dynamic execution fragments of tasks and includes fine-grained measurement of computation in those fragments. Using this performance model, TaskProf2's what-if analyses identify regions that improve the parallelism of the program while considering tasking overheads. Its differential analyses perform fine-grained differencing of an oracle and the observed performance model to identify static regions experiencing secondary effects. We have used TaskProf2 to identify regions with serialization bottlenecks and secondary effects in many applications.

CCS Concepts • **General and reference → Performance**; • **Software and its engineering → Software performance**.

Keywords Profilers, Parallelism, What-if analyses

ACM Reference Format:
Adarsh Yoga and Santosh Nagarakatte. 2019. Parallelism-Centric What-If and Differential Analyses. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '19), June 22–26, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3314221.3314621

1 Introduction

Given the appeal of performance portable code, many task parallel frameworks have become mainstream (*e.g.*, Intel Threading Building Blocks (TBB) [16], Cilk [23], Microsoft Task Parallel Library [35], X10 [11], OpenMP tasks [52], and Java Fork/Join tasks [34]). Programming with tasks addresses the problem of load imbalance by relying on a runtime to distribute programmer-specified tasks to hardware threads. Beyond load imbalance, a task parallel program can still experience performance issues: limited parallel work, runtime overhead due to fine-grained tasks, and secondary effects of execution such as false sharing. Hence, a programmer will benefit from profilers that pinpoint these bottlenecks.

There is a large body of work on identifying these pathologies. This includes load imbalances [18, 50, 63, 64], tasking overheads [25, 30, 56], serialization bottlenecks [26, 59, 68], scalability bottlenecks [19, 20, 22], memory contention [41, 46], and data locality issues [2, 38–40, 42]. While these techniques are useful in identifying specific bottlenecks, they do not provide useful information on the regions that matter in improving the overall parallelism of the program.

Our work is related to Coz [17] and Intel Advisor [14] in predicting performance improvements. Coz measures the virtual speedup of a progress point by slowing down all other concurrently executing threads and by building an analytical model to estimate the speedup. The improvements suggested by Coz are specific to an execution with a given number of threads. Specifically, it does not indicate whether the program will have scalable speedups when executed on a machine with a different core/thread count. Moreover, Coz's speedup estimates can widely vary based on the choice of progress points. Identifying appropriate progress points to obtain useful feedback from Coz in a large application can be challenging. Similarly, Intel Advisor also does not help in identifying the regions for annotation to estimate speedup improvements.

Our prior work, TaskProf [68], identifies parts of the program that matter in increasing logical parallelism. Logical parallelism quantifies the speedup of the execution for a given input on a large number of processors assuming an ideal runtime (*i.e.*, from Amdahl's law) [23, 26, 59, 68]. It is constrained by the longest chain of work that must be executed serially. To measure logical parallelism for a given input, TaskProf identifies dynamic fragments of tasks (without any task runtime calls) that must be executed serially and those that can be executed in parallel. It also measures the amount of work performed by each

PLDI '19, June 22–26, 2019, Phoenix, AZ, USA

ACM ISBN 978-1-4503-6712-7/19/06…$15.00
https://doi.org/10.1145/3314221.3314621

such fragment. TaskProf uses the dynamic program structure tree (DPST) [54] representation for maintaining series-parallel relationships between various fragments of execution. In the context of measuring logical parallelism, the series-parallel relationship between various fragments of tasks and fine-grained measurements of computation is a performance model of the execution for a given input. TaskProf uses this performance model to identify serialization bottlenecks and to estimate improvements in performance when programmers specify the region they intend to optimize. By measuring and analyzing parallelism, the profiler can identify serialization bottlenecks that would manifest in an execution on a machine with a large core count even when the profile execution is on a machine with a low core count.

Although TaskProf was useful in our initial experiments, we discovered the following limitations: (1) it did not consider the cost of orchestrating parallel execution, which can result in over-optimistic suggestions, (2) it required the programmer to annotate regions to estimate improvements in parallelism, which can be challenging in a large application, (3) it maintained the entire performance model on disk, which restricted its use to small programs, and (4) it did not provide feedback on the secondary effects of execution.

This paper presents TaskProf2: a parallelism profiler and an adviser. It identifies regions with low parallelism and tasking overheads. As an adviser, it automatically identifies a set of static regions that matter in increasing parallelism while accounting for tasking overheads. It includes what-if analyses that identify regions that matter in improving parallelism. It also includes a novel differential analysis that identifies regions experiencing secondary effects of execution. A key enabler for these analyses is a performance model that consists of series-parallel relationships and fine-grained measurements of computation.

TaskProf2 does not maintain the entire performance model in memory. It constructs the performance model during program execution. TaskProf2 has two modes of usage: an offline mode where it maintains the entire performance model on disk (Section 4.1) and an on-the-fly mode where it does not maintain the performance model on disk (Section 4.2). In both modes, TaskProf2 computes the parallelism, serial work on the critical path, the code region performing the highest work on the critical path, and the additional work done to create tasks. Finally, it summarizes this information with each spawn site (Section 4.1). Figure 1(c) shows a sample parallelism profile that highlights serialization bottlenecks and spawn sites with high tasking overhead.

What-if analyses. In its advisory role, TaskProf2's what-if analyses identify parts of the program that must be optimized to improve parallelism. Similar to our prior work [7, 68], what-if analyses estimate improvements in parallelism using the performance model even before the program is concretely optimized. To quantify the effect of addressing a serialization bottleneck, what-if analyses model the parallelization of a region of the program by reducing the serial work while keeping the total work unchanged when the program's overall parallelism is computed. Unlike our prior work, TaskProf2 automatically identifies all regions that need to be optimized to increase the parallelism to a user-specified threshold while taking the cost of creating tasks into account. TaskProf2 does not require programmer annotations. In the on-the-fly mode where the performance model is not available on disk, TaskProf2 identifies all regions that matter in increasing parallelism iteratively. It first executes the program, computes the parallelism profile, and identifies the region with the highest work on the critical path. In the subsequent iterations, it computes parallelism by reducing the serial work for the identified regions while considering the cost of tasking (Section 4.2).

Differential analyses. Apart from serialization bottlenecks and tasking overheads, a low speedup can be attributed to secondary effects of execution. The program typically observes work inflation (or in any other metric of interest) compared to an oracle execution when it experiences secondary effects [51]. Our performance model enables reasoning about work inflation at a fine-granularity rather than just considering inflation for the entire program. We propose differential performance analyses that identify program regions experiencing secondary effects by performing a fine-grained comparison of two performance models: one for the parallel execution and the other for an oracle execution (Section 5). The serial execution of the same parallel program can be considered as an oracle execution. The profiler performs this fine-grained comparison with multiple metrics of interest. The profiler highlights program regions that experience inflation with a metric of interest. Figure 1(e) shows the differential profile reported by TaskProf2.

TaskProf2 is open-source and publicly available [71]. We evaluated TaskProf2 with twenty-three applications that use the Intel Threading Building Blocks library for tasking. TaskProf2 identified bottlenecks in all of them. We designed concrete strategies to improve the speedup with nine applications using the feedback from the profiler.

2 Background on the DPST

TaskProf2 uses the dynamic program structure tree (DPST) representation to encode series-parallel relationship in the performance model. The DPST was originally proposed for explicit async-finish programs [54]. We modify its construction for task parallel programs without explicit finish statements [67, 68, 73]. The DPST can be constructed in parallel during program execution.

The DPST is an ordered tree with three kinds of nodes: step nodes, async nodes, and finish nodes. A step node represents a dynamic execution fragment (longest sequence of instructions without any task runtime calls). All useful work in the program happens in the step nodes. Step nodes are

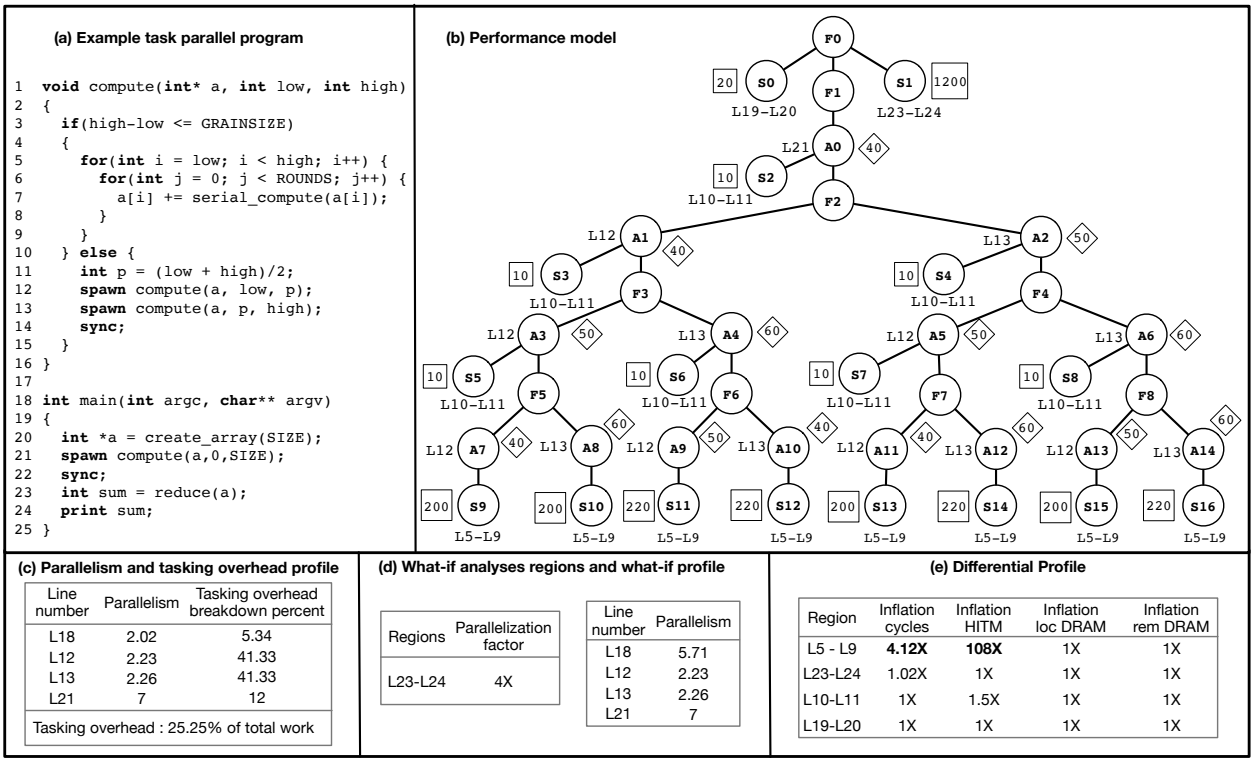


Figure 1. (a) An example task parallel program. (b) The performance model for an execution of the program in (a), where the GRAINSIZE is 1/8th the size of the array. The number in the rectangular box next to each step node represents the work performed in the program segment corresponding to the step node. The number in diamond boxes next to each async node represents the cost of creating the task corresponding to the async node. (c) Parallelism profile reported by our profiler. (d) The regions reported by what-if analyses to achieve a parallelism of 16 and the what-if profile if the reported regions are optimized. (e) Differential profile showing inflation of various metrics in parallel execution over serial execution. We show four hardware performance counter event types: execution cycles, HITM, local DRAM accesses, and remote DRAM accesses.

leaves in the DPST. In contrast, finish and async nodes are intermediate nodes that are used to encode series-parallel relationships between various step nodes. The edges in the DPST represent the parent-child relationship between the nodes. The DPST captures the logical series-parallel relationship between fragments of tasks according to the semantics of the tasking constructs. It is possible that two logically parallel fragments execute serially on the same thread or core in a specific execution.

Properties of the DPST. The formal treatment of the DPST is available in prior work [54]. We highlight the key properties of the DPST that are useful with respect to the profiler. (1) The children of a node in the DPST are ordered left-to-right to reflect serial execution of various fragments of the task represented by that node. (2) All nodes in the sub-tree under an async node can logically execute in parallel with all sibling nodes to the right of the async node and their descendants. (3) All nodes in the sub-tree under a finish node happen in series with all sibling nodes to the right of the finish node and their descendants. (4) Two step nodes can logically execute in parallel if the child of the least common ancestor of the step nodes, which is also an ancestor of the left-most step node, is an async node. (5) The DPST does not change across different schedules for a given input provided the program is free of data races. If the program uses locks or critical sections, the DPST does not change across schedules for Abelian programs [12].

Illustration. Figure 1(b) provides the DPST for an execution of the program in Figure 1(a). Nodes `A0-A14` are the async nodes, `F0-F8` are the finish nodes, and `S0-S16` are the step nodes in the DPST. An async node is created when the program spawns a task. The async nodes `A0`, `A1`, and `A2` are created when the program spawns tasks in line 21, line 12, and line 13 in Figure 1(a), respectively. A finish node represents a block of execution that is in series with the code following the block. In Figure 1(b), the finish node `F1` represents the block of execution starting with the creation of a task in line 21 in Figure 1(a) and terminating with the `sync` statement in line 22 in Figure 1(a). The step node `S0` represents the set of dynamic instructions between lines

19-20 in Figure 1(a). The step nodes S3 to S8 represent different dynamic instances of the region between L10-L11 in Figure 1(a). Similarly, the step nodes S9 to S16 represent different dynamic instances of the region between L5-L9 in Figure 1(a).

We can clearly identify if two step nodes logically execute in parallel, using the DPST. Step nodes S13 and S14 logically execute in parallel since their least common ancestor is F7 and the immediate child of F7 on the path to S13 is an async node (*i.e.* A11). In contrast, nodes S7 and S13 in Figure 1(b) execute serially because the immediate child of their least common ancestor (*i.e.*, A5) on the path to S7 is not an async node. TaskProf2 uses these properties to compute the parallelism profile and to perform what-if analyses.

3 High Level Overview

We provide an overview of what-if and differential analyses using a task parallel program shown in Figure 1(a), which recursively spawns tasks and performs some computation on each element of an array. We use spawn and sync to represent the fork and join of a task, respectively. When executed on a machine with four cores, the speedup of the program is 1.35× over a serial execution. As this program has low speedup, the user may want to profile it.

Performance model. The profiler constructs a performance model by executing the program for a given input. The performance model consists of three components: (1) series-parallel relationship between various dynamic fragments of tasks in the execution encoded as a dynamic program structure tree (DPST) [54], (2) fine-grained measurement data for computation performed by each dynamic execution fragment of tasks, and (3) amount of work performed by the runtime to create tasks. The series-parallel relationship along with this fine-grain measurement of work enables what-if analyses. The measurement of task creation work allows TaskProf2 to pinpoint sources of high tasking overhead in the program (see Section 4). Comparison of computation from two performance models, one from a parallel execution and another from an oracle execution, enables our profiler to identify sources of secondary effects (Section 5). Figure 1(b) illustrates the performance model of the program in Figure 1(a). Each step node has the location information about the source code it represents and the amount of work performed by it (shown next to each step node in Figure 1(b)). Each async node has the location information of the spawn site and the number of cycles spent by the runtime to create the task (shown next to each async node in Figure 1(b)).

Parallelism profile. TaskProf2 generates a parallelism profile that specifies the parallelism and tasking overhead of the entire program and at each spawn site in the program by analyzing the performance model (Section 4). Figure 1(c) presents the parallelism profile for an execution of the program in Figure 1(b). It shows that the program has a low parallelism of 2.02 and a high tasking overhead of 25% (see the tasking overhead percent in the last line of Figure 1(c)). To improve the speedup, we need to improve the parallelism in the program and reduce the tasking overhead.

Identifying cut-offs with recursive decomposition. The parallelism profile in Figure 1(c) reports that the program performs about 25% additional work to create tasks, which indicates that the cut-off for recursive decomposition is too small. The profile also indicates that the two recursive calls (line 12 and line 13 in Figure 1(a)) together account for 82.66% of the task creation overhead. We can reduce tasking overheads by decreasing the parallelism until the work done by the step nodes is reasonably higher than the average task creation overhead. We increased the cut-off point in Figure 1(a). The tasking overhead reduced to 11.78%. The speedup on a 4-core machine increased from 1.35× to 1.58×.

What-if analyses for guidance. What-if analyses over the performance model enable the profiler to estimate the change in parallelism of the program when any region is hypothetically optimized. Using what-if analyses, the profiler iteratively identifies all static regions in the program that need to be optimized to increase the parallelism to a user-specified threshold while accounting for tasking overheads.

Figure 1(d) reports the region identified by TaskProf2 using what-if analyses when the user wants to increase the parallelism of the program to 16. In this example, the average tasking overhead is 50 units per task creation and we assume that each step node has to do at least 5× the tasking overhead (*i.e.*, 250) to amortize the cost of parallel execution. To compute regions that matter, the profiler identifies the step node performing the highest work on the critical path (*i.e.*, S1 performing 1200 units). The profiler's what-if analyses reduces the serial work of S1 by 4× because reducing the serial work any further would make tasking overhead significant. It recomputes the parallelism and chooses the step node with the highest work on the recomputed critical path. In this example, the profiler discovers that it cannot reduce the serial work of any other node because tasking overheads would dominate compared to useful work. The profiler reports the discovered regions and the parallelism of the program and each spawn site if the regions reported in the what-if profile are parallelized. The what-if profile in Figure 1(d) shows that the parallelism of the program can increase to 5.71 when we parallelize the reported region. After concretely increasing grain size to reduce tasking overheads and parallelizing the region reported by the what-if profile, the speedup increased from 1.35× to 2.91×.

Differential analyses to identify secondary effects. Since the speedup of the program in Figure 1(a) is 2.91× after our optimizations on a 4-core machine, the program is likely experiencing secondary effects of execution. The profiler's differential analyses compares the performance model of the parallel execution and that of an oracle execution to identify static regions experiencing secondary effects. The execution

```
1  function ComputeProfile(T)
2      foreach N in bottom-up traversal of T do
3          C_N ← ChildNodes(N)
4          N.w ← Σ_{C∈C_N} C.w
5          ⟨N.s, N.l⟩ ← SerialWork(N)
6          A_N ← AsyncChildNodes(N)
7          F_N ← FinishChildNodes(N)
8          N.t ← Σ_{A∈A_N} A.c + Σ_{C∈C_N} C.t
9          N.et ← Σ_{A∈A_N} A.c + Σ_{F∈F_N} F.et
10     end
11     AggregatePerSpawnSite(T)
12     return ⟨T.w, T.s, T.l, T.t⟩
```

Figure 2. Algorithm to produce the parallelism profile given a performance model T. The work performed by each step node is represented by `w`. Each async node includes the cost of task creation (*i.e.*, variable `c`). The algorithm computes $\langle w, s, l, t\rangle$ for each intermediate node in T, where `w` is the work, `s` is the serial work, `l` is the list of step nodes on the critical path, `t` is the tasking overhead, and `et` is the exclusive tasking overhead. ChildNodes returns all the child nodes of the input node. FinishChildNodes and AsyncChildNodes return the async and finish child nodes of the input node. AggregatePerSpawnSite aggregates the work, serial work and tasking overhead per spawn site.

of a parallel program on a single core with the same tasking structure can serve as an oracle execution. When a program experiences secondary effects, there will be inflation in some metric of interest.

Figure 1(e) presents the differential profile for the program with respect to multiple metrics of interest. One can notice that the region L5-L9 is experiencing inflation in the total number of hardware execution cycles and the number of cache accesses in the modified state (*i.e.*, HITM) when compared to an oracle execution. One possible reason for an increase in HITM hardware counters in the cache is false sharing. This program indeed has false sharing. We fixed the false sharing problem and the speedup of the program increased from 2.91× to 3.83×.

4 Profiling with What-If Analyses

TaskProf2 is both a parallelism profiler and an adviser. As a profiler, it identifies parts of the program with low parallelism. As an adviser, it identifies regions of the program that matter in increasing parallelism. To identify regions that matter, TaskProf2 needs a strategy to quantify the impact of parallelizing a region on the program's overall parallelism. The series-parallel relationships encoded as a DPST along with the fine-grained measurement of work in the step nodes can enable the computation of logical parallelism. Hence, the DPST enriched with fine-grained measurements is a performance model.

The performance model consists of three components: (a) series-parallel relationships encoded as a DPST, (b) fine-grained measurement of computation for each step node, and (c) fine-grained measurement of the cost incurred to create tasks in the program, which is associated with an async node. To provide feedback about recursive decomposition of tasks and the task creation overhead, the performance model also includes the amount of work performed to create tasks. The profiler can measure any metric of interest (*e.g.*, execution time, hardware cycles, or dynamic instructions) corresponding to the step node.

TaskProf2 constructs this performance model in parallel in a distributed fashion when the task parallel runtime creates, executes, waits, or steals tasks. The profiler updates its data structures on task runtime calls and starts the measurement of work with hardware performance counters when the runtime returns to the program. The profiler does not maintain the entire performance model in memory. It maintains a small slice of the performance model, which is proportional to the number of active tasks maintained by the task parallel runtime. TaskProf2 contains two types of analyses: an offline parallelism analysis that maintains the entire performance model on disk and an on-the-fly parallelism analysis that does not maintain the entire performance model either on disk or in memory. The offline analysis enables TaskProf2 to identify regions that matter without re-execution of the program. In contrast, on-the-fly analysis enables the use of TaskProf2 with long running applications but requires re-execution of the program for what-if analyses. First, we describe our algorithm with the assumption that the entire performance model is available. Section 4.2 describes our on-the-fly algorithm.

4.1 Computing Parallelism and Tasking Overhead

Given that the entire performance model is available, the parallelism profile can be computed by performing a bottom-up traversal of the performance model. TaskProf2 computes five quantities for each intermediate node in the bottom-up traversal: (1) the total work under the sub-tree, (2) the total serial work under the sub-tree, (3) the set of step nodes performing the largest amount of serial work, (4) the total tasking overhead under the sub-tree, and (5) the exclusive tasking overhead to spawn the immediate children of the intermediate node. TaskProf2 computes the parallelism in the program using the total work and the total serial work information. The list of step nodes performing the serial work for the root node in the performance model is the critical path of the program. The profiler computes the total and the exclusive tasking overhead to provide feedback about the amount of additional work done to orchestrate parallel

1 **function** SerialWork(N)
2 $S_N \leftarrow$ StepChildNodes(N)
3 $F_N \leftarrow$ FinishChildNodes(N)
4 $s \leftarrow \sum_{S \in S_N} S.w + \sum_{F \in F_N} F.s$
5 $l \leftarrow (\bigcup_{F \in F_N} F.l) \cup S_N$
6 **foreach** $A \in$ AsyncChildNodes(N) **do**
7 $LS_A \leftarrow$ LeftStepSiblings(A)
8 $LF_A \leftarrow$ LeftFinishSiblings(A)
9 $lw \leftarrow \sum_{S \in LS_A} S.w + \sum_{F \in LF_A} F.s$
10 **if** $lw + A.s > s$ **then**
11 $s \leftarrow lw + A.s$
12 $l \leftarrow (\bigcup_{F \in LF_A} F.l) \cup LS_A \cup A.l$
13 **end**
14 **end**
15 **return** $\langle s, l \rangle$

Figure 3. Algorithm to compute the serial work and the list of step nodes on the critical path for a given input node N in the DPST. StepChildNodes, FinishChildNodes, and AsyncChildNodes return the children step, finish, and async nodes of the given input node, respectively. LeftStepNodeSiblings and LeftFinishNodeSiblings return all the left step and finish siblings of the input node, respectively.

execution. Subsequently, this information is aggregated with each spawn site.

Figure 2 presents the algorithm to compute the above five quantities for each intermediate node. The values of the children bubble up to the parent in a bottom-up traversal. The total work at an intermediate node (variable `w` in Figure 2) is the sum of the work performed by all step nodes in the sub-tree under the node. In the context of a bottom-up traversal, the total work is equal to the sum of the work performed by all the immediate children of that node (line 4 in Figure 2).

Computing serial work. We leverage the properties of the DPST to compute the highest serial work in a sub-tree. Among all the step nodes in the sub-tree under an intermediate node, some subset of these step nodes execute serially. We have to compute the chain of step nodes that execute serially. There can be multiple such chains. We have to select the chain that performs the highest serial work.

Figure 3 presents the algorithm to compute both the serial work (variable `s` in Figure 3) and the list of step nodes performing the serial work (variable `l` in Figure 3). All immediate step node children of an intermediate node execute serially. Similarly, any serial work done by the immediate finish children also adds to the serial work. Hence, the algorithm in Figure 3 initially sets the work done by the immediate step children and the serial work done by the immediate finish children as the serial work (lines 2-4 in Figure 3). Similarly, the list of step nodes performing serial work is the union of the immediate step children and the list performing serial work under the immediate finish children (line 5 in Figure 3).

Each async node child of an intermediate node creates a chain of serial work. The descendants of an async node execute in parallel with the siblings of the async node that occur to the right of the async node. The step and finish siblings that occur to the left of the async node occur in series with the descendants of the async node. When an intermediate node has an async child, the highest serial work is the maximum of : (1) serial work done by the step and finish children to the left of the async node and the serial work performed by the sub-tree under async node (lines 9-11 in Figure 3) or (2) serial work done by all immediate step and finish children. Hence, the algorithm examines the serial chain created by each async node and chooses the path with the maximum serial work (lines 6-14 in Figure 3). Similarly, the list of step nodes performing serial work is updated whenever the serial work is updated.

For example, consider the intermediate node `F8` in Figure 1(b). It has two async nodes — `A13` and `A14` — as children. The serial work of `F8` is the maximum of the serial work performed by two chains: (1) serial work done by `A13` as it does not have left step or finish siblings or (2) serial work done by `A14` as it does not have any left step or finish siblings.

Computing tasking overheads. To provide feedback about the cost of parallel execution, our profiler also computes the total tasking overhead incurred in the sub-tree under an intermediate node. It also computes the exclusive tasking overhead incurred to spawn its immediate children. The total tasking overhead enables the user to quantify the amount of extra work done to create tasks in contrast to the useful work in the program. The exclusive tasking overhead enables attribution of the overhead to specific spawn sites. Each async node knows the cost incurred to create that specific task from the profile execution (variable `c` in Figure 2). The total tasking overhead of an intermediate node (variable `t` in Figure 2) is the sum of the task creation costs of all async children and the total tasking overheads under the sub-trees of all children (line 8 in Figure 2). The exclusive tasking overhead (variable `et` in Figure 2) of an intermediate node is sum of the task creation costs of the immediate async children and the exclusive tasking overheads of its finish children (line 9 in Figure 2). We need to propagate the exclusive tasking overhead from the finish children because this overhead is eventually attributed to a spawn site and only async nodes have spawn site information.

Aggregating information per-spawn site. The profiler aggregates the information about parallelism, tasking overheads, and the step nodes performing serial work to static spawn sites in the program and produces the parallelism profile similar to Figure 1(c). TaskProf2 uses the spawn site

```
1  function OnNodeCreation(N, P)
2      N.lw ← P.sw
3  function OnNodeCompletion(N, P)
4      P.w ← P.w + N.w
5      if N.lw + N.s > P.s then
6          P.s ← N.lw + N.s
7      end
8      if N is an ASYNC node then
9          AggregatePerSpawnSite(N)
10         P.t ← P.t + N.c + N.t
11         P.et ← P.et + N.c
12     else
13         P.sw ← P.sw + N.s
14         P.t ← P.t + N.t
15         P.et ← P.et + N.et
16     end
```

Figure 4. On-the-fly algorithm to compute the parallelism profile without the entire performance model. Each node tracks six quantities: work (w), serial work in the sub-tree (s), serial work done by the left siblings (lw), serial work done by its immediate children (sw), total tasking overhead in the sub-tree (t), and exclusive tasking overhead to spawn immediate child tasks (et). This information is updated when the node (N) is added or removed in the performance model. P is the parent of N.

information with the async node in the performance model to perform this aggregation. In the presence of recursive calls, TaskProf2 aggregates information to a spawn site corresponding to an async node if the path from the root to the current async node does not contain another async node with the same spawn site information. Figure 1(c) highlights spawn sites performing significant serial work and experiencing the highest tasking overhead.

4.2 On-the-Fly Parallelism Profile

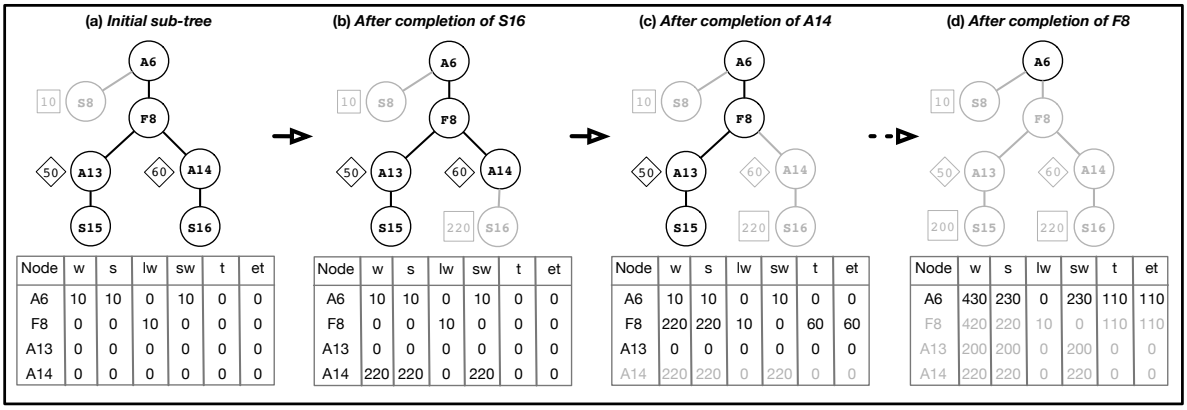
As the performance model for a long-running program can be large, TaskProf2's on-the-fly mode does not maintain the entire performance model either in memory or on disk. It maintains a small slice of the performance model in memory that is proportional to the number of active tasks. In the on-the-fly mode, each node (1) tracks sufficient information to compute the work and serial work on the critical path without the entire performance model, (2) summarizes the information about the computation in the sub-tree under the node to the parent, and (3) deallocates on completion.

Specifically, TaskProf2 tracks six quantities with each node to compute the parallelism profile. First, the total work (w) performed by the sub-tree under the node. Second, the serial work on the critical path (s) performed in the sub-tree under the node. Third, serial work performed by the siblings to the left (lw) of a given node. We need to track the serial work done by the left siblings of a node as we will not have the left siblings in the performance model when the node completes execution. Fourth, the serial work performed by the immediate children (sw) of a given node. This information will be useful to initialize the serial work done by the left siblings for a child of the given node. Fifth, the total tasking overhead (t) under the sub-tree of a given node. Sixth, the exclusive tasking overhead (et) to attribute the cost of spawning child tasks to specific spawn sites.

Figure 4 presents our algorithm to compute the parallelism profile on-the-fly without maintaining the entire performance model either in memory or on-disk. We leverage the property that the series-parallel relationship between the existing nodes are unchanged even when new nodes are added to the performance model. When a node is added to the performance model, the parent initializes the left sibling work (lw) for the node. The parent uses the serial work performed by its immediate children (sw) to initialize the left sibling work for the node (line 2 in Figure 4). It has to be initialized by the parent when the node is created because the parent can create other siblings to the right of the node by the time the entire sub-tree under the node completes.

When a node completes, the node checks if it contributes to the critical path under the parent (lines 5-7 in Figure 4). Hence, it checks if the serial work done by the left siblings (lw) and the serial work in the sub-tree under the node (s) is greater than the serial work on the critical path for the parent, which is similar to offline analysis algorithm (lines 9-10 in Figure 3). Whenever an async node completes, TaskProf2 propagates information in three steps. First, it adds the cost of task creation to the exclusive tasking overhead to the parent (line 11 in Figure 4). Second, it adds the total tasking overhead under the sub-tree of the async node to the tasking overhead of the parent (line 10 in Figure 4). Third, it aggregates the information with respect to the spawn site of the async node. Whenever a step or a finish node completes, TaskProf2 adds the serial work under the node (s) to the parent's serial work performed by the immediate children (sw), which is useful to initialize the left sibling work for future children of the parent node. TaskProf2 also propagates the tasking overheads from the node to the parent. Figure 5 illustrates the updates to the six quantities when a node is created and when a step, finish, and an async node completes.

Additions for what-if analyses. To enable what-if analyses, we also need to track the step node performing the highest work on the critical path. It needs to be updated whenever the critical path is updated. The algorithm in Figure 4 omits this information. It can be accomplished by tracking additional information that identifies the static location and the step node performing the highest work with lw, s, and sw quantities. It needs to be updated when any of the three quantities change for each node.



Node	w	s	lw	sw	t	et
A6	10	10	0	10	0	0
F8	0	0	10	0	0	0
A13	0	0	0	0	0	0
A14	0	0	0	0	0	0

Node	w	s	lw	sw	t	et
A6	10	10	0	10	0	0
F8	0	0	10	0	0	0
A13	0	0	0	0	0	0
A14	220	220	0	220	0	0

Node	w	s	lw	sw	t	et
A6	10	10	0	10	0	0
F8	220	220	10	0	60	60
A13	0	0	0	0	0	0
A14	220	220	0	220	0	0

Node	w	s	lw	sw	t	et
A6	430	230	0	230	110	110
F8	420	220	10	0	110	110
A13	200	200	0	200	0	0
A14	220	220	0	220	0	0

Figure 5. An illustration of on-the-fly parallelism computation for the sub-tree rooted at A6 in Figure 1(b). (a) The sub-tree after the creation of the nodes with the quantities initialized. (b), (c), and (d) show the sub-tree and the six quantities described in Section 4.2 after the completion of step node S16, async node A13, and finish node F8, respectively. The nodes and the quantities for those that have already completed are grayed out.

4.3 Identifying Regions with What-If Analyses

The parallelism profile highlights spawn sites with low parallelism and/or high tasking overheads. The user may want to explore these regions. However, the user will not know whether parallelizing them matters in increasing parallelism. Even after successfully parallelizing a region, the parallelism may not increase when the program has multiple chains of step nodes that perform similar amount of serial work.

TaskProf2's what-if analyses estimate improvements in parallelism using the performance model even before the programmer has concretely parallelized these regions. The what-if analyses mimic the effect of parallelization by reducing the serial work performed by the step nodes in the performance model corresponding to a region of interest by the intended parallelization amount while keeping total work unchanged. Subsequently, we can compute the parallelism of the entire program as described in Section 4.1. A possible use case of what-if analyses is to allow the programmer to iteratively identify regions that matter in increasing parallelism, one region at a time. Our prior work took this approach [68]. However, such an approach can be time consuming for the programmer especially while performance debugging a large application. TaskProf2 automatically identifies all regions that need to be parallelized to increase the parallelism in the program to a user-specified threshold. Although one can theoretically reduce the serial work by any amount, it is practically infeasible because creating and executing a task incurs some overhead. To provide a realistic view of possible parallelization opportunities, our what-if analyses also consider the cost of creating tasks to orchestrate parallel execution.

Our what-if analyses can be used when the entire performance model is available on disk or with the on-the-fly mode. When then entire performance model is available on disk, what-if analyses can identify regions that need to be optimized without re-executing the program. In the on-the-fly mode, the program is executed multiple times to identify regions that matter. In each such re-execution, the serial work for all the identified regions from the previous iterations are reduced during parallelism computation on node completion, which mimics the effect of parallelization.

Figure 6 presents our algorithm to identify all static regions in the program that need to be parallelized. It assumes that the entire performance model is available. The programmer specifies the anticipated parallelism that the program is expected to achieve, the tasking overhead threshold, and the maximum amount of parallelization feasible for a region. TaskProf2 computes the parallelism profile (line 3 in Figure 6), identifies the step node performing the highest work on the critical path, and the static region corresponding to the step node (line 4 in Figure 6). It reduces the serial work of all the step nodes that correspond to the identified region by the maximum amount of parallelization feasible while keeping the total work done by the region unchanged (lines 8-14 in Figure 6). The serial work of a step node is reduced to the maximum of either the threshold tasking overhead or the serial work reduced by the parallelization factor. Subsequently, TaskProf2 recomputes the parallelism (line 16 in Figure 6) and repeats the above process until either the anticipated parallelism is reached or the work performed by every node on the critical path is less than the tasking overhead threshold (line 7 in Figure 6). If the anticipated parallelism is not achieved when our algorithm terminates, it indicates that the program does not have sufficient parallelism for a given input. TaskProf2 outputs the set of regions that were considered in each iteration as the regions that need to be optimized to improve the parallelism of the program (line 20 in Figure 6).

```
1  function WhatIfAnalyses(T, ap, pf, k)
2      regions ← ∅
3      < w, s, l, t > ← ComputeProfile(T)
4      < smax, r > ← MaxStepOnCritPath(l)
5      tt ← k * AvgTaskingOverhead(T)
6      cp ← w/s
7      while (cp < ap) ∧ (smax > tt) do
8          foreach N in bottom-up traversal of T do
9              S_r ← {S|(S ∈ ChildNodes(N) ∧ S ∈ r)}
10             foreach S ∈ S_r do
11                 S.w ← Max(tt, S.w/pf)
12             end
13             ⟨N.s, N.l⟩ ← SerialWork(N)
14         end
15         regions ← regions ∪ r
16         cp ← w/T.s
17         < smax, r > ← MaxStepOnCritPath(T.l)
18     end
19     AggregatePerSpawnSite(T)
20     return regions
```

Figure 6. Algorithm to compute regions that matter in increasing parallelism using what-if analyses on the performance model T. The user specifies the anticipated parallelism (ap), parallelization possible for the regions (pf), and a task overhead threshold (k). The function MaxStepOnCritPath returns the step node that performs the highest work on the critical path and the corresponding static program region. Here, *smax* refers to the step node and the work performed by it. AvgTaskingOverhead computes the average tasking overhead– ratio of the total tasking overhead and the number of async nodes in T. The function ChildNodes returns the immediate children of the input node N.

5 Differential Performance Analysis

Even when the program has sufficient parallelism and low tasking overheads, a parallel program execution can fail to achieve the maximum possible speedup on a given machine. Typically, this behavior is attributed to secondary effects of parallel execution. In contrast to logical parallelism, secondary effects can be specific to a hardware configuration. Moreover, it can change across executions on the same machine. For example, when multiple parallel tasks simultaneously access distinct data in the same cache block (*i.e.* false sharing), cache invalidations will reduce the speedup. To observe this behavior, two such tasks have to be scheduled to execute at the same time in the execution.

Differential analysis using work inflation. We observe that our performance model can be used to identify static regions of the program experiencing secondary effects. When a region is experiencing secondary effects, it performs more work in the parallel execution compared to an oracle execution [3, 43, 51]. The performance model for a task parallel execution on a multi-core machine captures fine-grained work information. If we construct a performance model for an oracle execution, then we can perform a fine-grained comparison of the two performance models.

The execution of a task parallel program on a single core can be considered as an oracle execution because each task would execute without any interference from other tasks. In a race-free program, the DPST and the series-parallel relationships encoded by the DPST in the performance model remain unchanged irrespective of the number of cores used for execution. Only the work information associated with the step nodes will differ between the performance model of the execution on multiple cores and that of the execution on a single core. Hence, our differential analyses performs fine-grained comparison of two performance models to isolate regions likely experiencing secondary effects.

Use of multiple metrics. Apart from measuring work done in hardware execution cycles, TaskProf2 can measure any metric that is supported on the machine. For instance, it can identify regions experiencing true or false sharing by measuring the number of cache hits in the modified state with each step node (*e.g.*, using the HITM performance counter) [21, 44]. It can identify regions experiencing reduced locality in shared caches due to parallel execution by measuring last level cache misses. Some regions can perform a large number of remote memory accesses due to the lack of affinity between the processor producing the data and the one using it. TaskProf2 can identify such regions by measuring accesses to remote DRAM.

TaskProf2's differential analysis constructs two performance models — one for the oracle execution and other for the parallel execution — for multiple metrics of interest. It has to execute the program twice for each metric of interest to construct the performance model. It aggregates the information about inflation in the metric of interest to static program regions. Figure 1(e) presents the differential analysis profile for the program in Figure 1(a). In Figure 1(e), our profiler has compared the performance model for the parallel execution with the oracle execution on four distinct metrics of interest. Figure 1(e) clearly shows that program is experiencing significant work and HITM inflation with static program region `L5-L9`, which provides information about concrete regions to explore to address secondary effects. The programmer can also guide our differential analysis by choosing appropriate metrics of interest and by tailoring the configuration (*e.g.*, by choosing the number for threads).

Which regions should one focus on? The programmer should focus on regions that experience significant inflation with a metric of interest and perform a reasonable fraction of the total work or the work on the critical path. If a region experiences significant inflation with some metric but performs very little work or serial work, then it likely will not affect

Table 1. Applications used in our evaluation. We list the initial speedup on a 16-core machine, the logical parallelism, total tasking overhead in the program in contrast to total useful work, the number of regions reported by our what-if analyses to increase the parallelism to 128, and the inflation in work in the entire program due to parallel execution.

Application	Initial speedup	Parallelism	Tasking overhead	# of regions reported	Work inflation in cycles
MILCmk	2.20X	44.21	41.90%	3	3.00X
LULESH	4.17X	43.01	7.37%	2	4.19X
compSort	5.80X	29.11	2.87%	2	1.28X
integerSort	4.93X	34.27	8.95%	2	1.16X
remDups	8.59X	48.86	2.04%	1	1.08X
dictionary	8.54X	41.94	2.49%	1	1.19X
suffixArray	2.15X	6.13	6.80%	3	1.15X
BFS	6.77X	24.19	3.08%	3	1.22X
maxIndSet	8.52X	27.24	7.17%	2	1.27X
maxMatching	9.39X	46.57	7.31%	1	1.23X
minSpanForest	6.43X	29.02	2.19%	1	1.28X
spanForest	7.17X	36.31	6.06%	1	1.41X
convexHull	8.14X	78.90	3.71%	0	1.59X
nearestNeigh	4.54X	17.83	7.09%	4	1.45X
delTriang	5.95X	57.55	7.92%	1	1.38X
delRefine	7.06X	51.50	8.83%	1	1.42X
rayCast	9.36X	51.44	20.83%	1	1.22X
nBody	12.26X	126.69	16.60%	1	2.12X
blackscholes	7.70X	40.03	1.02%	2	1.02X
bodytrack	6.32X	31.95	2.71%	2	1.16X
fluidanimate	10.41X	64.97	2.41%	1	1.02X
streamcluster	12.28X	76.28	9.17%	5	1.47X
swaptions	12.24X	74.17	47.27%	0	1.39X

the speedup of the program. Hence, TaskProf2's differential analysis also reports the percentage of the total work and the percentage fraction of the serial work performed by the region experiencing inflation with a metric of interest.

6 Experimental Evaluation

We evaluate TaskProf2's effectiveness in identifying various bottlenecks (serialization, tasking overheads, and secondary effects), and regions that matter in improving parallelism. We also describe our experience with three other tools: Coz [17], Intel Advisor [14], and Intel Vtune [15].

6.1 Prototype and Methodology

TaskProf2 supports task parallel programs that use the Intel Threading Building Blocks (TBB) library [16]. It consists of a profiler runtime that performs both offline and on-the-fly parallelism computation, an extended TBB library that calls the profiler runtime, and analysis tools to perform what-if and differential analyses. It uses hardware performance counters to perform fine-grained measurement of various metrics. The prototype can use any available hardware performance counter: dynamic instructions, execution cycles, HITM events, local and remote DRAM accesses, last level cache misses, and floating point operation cycles.

Methodology and setup. We performed all experiments on a 16-core dual-socket Intel x86-64 2.1Ghz Xeon server with 64 GB RAM and hyper-threading disabled. It has a 32KB data cache, 32KB instruction cache, 256KB L2 cache, and 20MB L3 cache. Each cache line is 64 bytes. We run each application five times while performing speedup experiments and consider the average execution time. To perform what-if analyses, we use 128 for both anticipated parallelism and possible parallelization because we want the application to have large enough parallelism to obtain scalable speedup on a machine with a large number of cores. We use 10× the average tasking overhead ($k = 10$ in Figure 6) as the tasking overhead threshold [59]. TaskProf2's profile execution is 2.69× slower on average when compared to parallel execution without any profiling. System calls that read hardware performance counters are the major component of this overhead. The resident memory overhead is 10%.

Applications. We used twenty-three applications to evaluate the prototype: `MILCmk` and `LULESH` applications from the Coral benchmark suite [1], sixteen applications from the Problem Based Benchmark Suite (PBBS) [62], and five TBB applications from the Parsec benchmark suite [5]. `MILCmk` and `LULESH` applications were originally written in OpenMP. The PBBS applications were written in Cilk. We converted them to use the Intel TBB library. The prototype, applications, and evaluation data are available open source [71].

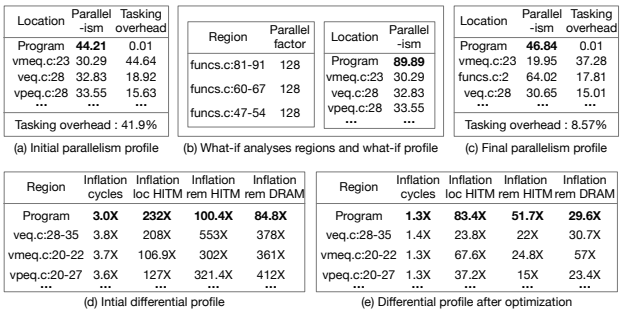
6.2 Effectiveness in Identifying Bottlenecks

Table 1 provides a summary of our results when we used the profiler with twenty-three applications.

Regions identified by what-if analyses. The logical parallelism in all these applications is less than 128, which is the anticipated parallelism for our experiments. The prototype's what-if analyses identified regions that need to be parallelized to increase the parallelism to 128 in all except two applications. The prototype did not identify any region in *convexHull* and *swaptions* since parallelizing regions on the critical path would increase tasking overhead of the program beyond the threshold. Hence, these two applications do not have sufficient parallelism for execution on a 128-core machine for the given input.

Programs with tasking overheads. Among the twenty three applications, we can observe that four applications have relatively high tasking overheads (> 10%) in Table 1. TaskProf2 helped us pinpoint the exact spawn sites experiencing tasking overhead in all four applications. We designed changes to reduce the tasking overhead in three of them, which we describe below in application case studies. We were unable to reduce the tasking overhead in *rayCast* because the computation to determine the cut-off was closely intertwined with actual computation and it was not straightforward to reduce the tasking overheads.

Programs with secondary effects. Table 1 also reports the inflation in total work. `MILCmk` and `LULESH` are performing 3× and 4.19× more work in parallel execution compared



Location	Parallel-ism	Tasking overhead
Program	**44.21**	0.01
vmeq.c:23	30.29	44.64
veq.c:28	32.83	18.92
vpeq.c:28	33.55	15.63
...	...	...
Tasking overhead : 41.9%		

(a) Initial parallelism profile

Region	Parallel factor
funcs.c:81-91	128
funcs.c:60-67	128
funcs.c:47-54	128

Location	Parallel-ism
Program	**89.89**
vmeq.c:23	30.29
veq.c:28	32.83
vpeq.c:28	33.55
...	...

(b) What-if analyses regions and what-if profile

Location	Parallel-ism	Tasking overhead
Program	**46.84**	0.01
vmeq.c:23	19.95	37.28
funcs.c:2	64.02	17.81
veq.c:28	30.65	15.01
...	...	...
Tasking overhead : 8.57%		

(c) Final parallelism profile

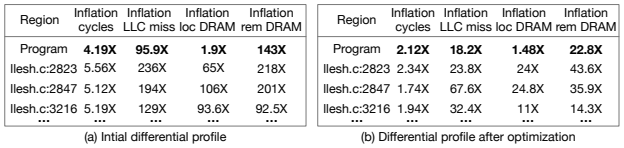
Region	Inflation cycles	Inflation loc HITM	Inflation rem HITM	Inflation rem DRAM
Program	**3.0X**	**232X**	**100.4X**	**84.8X**
veq.c:28-35	3.8X	208X	553X	378X
vmeq.c:20-22	3.7X	106.9X	302X	361X
vpeq.c:20-27	3.6X	127X	321.4X	412X
...	...	...	...	...

(d) Intial differential profile

Region	Inflation cycles	Inflation loc HITM	Inflation rem HITM	Inflation rem DRAM
Program	**1.3X**	**83.4X**	**51.7X**	**29.6X**
veq.c:28-35	1.4X	23.8X	22X	30.7X
vmeq.c:20-22	1.3X	67.6X	24.8X	57X
vpeq.c:20-27	1.3X	37.2X	15X	23.4X
...	...	...	...	...

(e) Differential profile after optimization

Figure 7. Profiles for MILCmk. (a) Initial parallelism profile. (b) Regions identified and the what-if profile. (c) Parallelism profile after concretely parallelizing the reported regions and reducing the tasking overhead. (d) Initial differential profile that reports the inflation in cycles, local HITM, remote HITM, and remote DRAM accesses. (e) Differential profile after addressing secondary effects using TBB's affinity partitioner. Here, we report only the top three spawn sites.



Region	Inflation cycles	Inflation LLC miss	Inflation loc DRAM	Inflation rem DRAM
Program	**4.19X**	**95.9X**	**1.9X**	**143X**
llesh.c:2823	5.56X	236X	65X	218X
llesh.c:2847	5.12X	194X	106X	201X
llesh.c:3216	5.19X	129X	93.6X	92.5X
...	...	...	...	...

(a) Intial differential profile

Region	Inflation cycles	Inflation LLC miss	Inflation loc DRAM	Inflation rem DRAM
Program	**2.12X**	**18.2X**	**1.48X**	**22.8X**
llesh.c:2823	2.34X	23.8X	24X	43.6X
llesh.c:2847	1.74X	67.6X	24.8X	35.9X
llesh.c:3216	1.94X	32.4X	11X	14.3X
...	...	...	...	...

(b) Differential profile after optimization

Figure 8. (a) The differential profile for LULESH showing the inflation in cycles, last level cache misses, local DRAM, and remote DRAM accesses. (b) The profile after reducing the secondary effects at `lulesh.c:2823` and `lulesh.c:2847`.

to an oracle execution, respectively. Using differential analysis, we were able to precisely locate regions causing high work inflation. Subsequently, we designed optimizations to reduce this inflation (Section 6.3).

Overall, we increased the speedup in nine applications. TaskProf2 helped us identify bottlenecks and regions that would help in improving parallelism. TaskProf2 does not perform automatic parallelization. It does not automatically address these bottlenecks. The programmer has to concretely optimize the regions reported by TaskProf2. It may not be feasible to optimize all regions reported by TaskProf2 (*e.g.*, due to dependencies). We designed concrete techniques to optimize the identified regions in nine applications and increase the speedup. We also ensured that the optimizations improved the speedup for all the available inputs.

6.3 Improving the Speedup of Applications

We describe our experience addressing serialization bottlenecks, tasking overheads, and secondary effects.

Increasing the speedup of `MILCmk`. The `MILCmk` program is a scientific application from the LLNL Coral benchmark suite with 5000 lines of optimized code. This application had a speedup of 2.2× on a 16-core machine. Figure 7(a) shows the parallelism profile reported by our profiler, which reports that the program has high tasking overheads (42% of total useful work) and sufficient parallelism (44.21). The program is spending one-third of the execution time orchestrating parallel execution. The profiler identifies six `parallel_for` calls that together account for almost 98% of the tasking overhead. Figure 7(a) shows top three calls due to space constraints. We carefully analyzed the program and increased the cut-off points for these six `parallel_for` calls until the tasking overhead was less than 10%. As a result, the speedup increased from 2.2× to 5.51×.

We subsequently used TaskProf2's what-if analyses to increase the parallelism to 128. Figure 7(b) presents the three regions reported by the profiler's what-if analyses and the what-if parallelism after optimizing these regions. When we carefully examined these reported regions, a part of each reported region was serially computing the sum of a large array of numbers. We parallelized a part of these reported regions with the `parallel_reduce` function in the Intel TBB library, which increased the parallelism to 46.84 and the speedup to 5.76×. Figure 7(c) reports the parallelism profile after parallelizing these three regions.

Subsequently, we used the profiler's differential analyses to check if the application is experiencing secondary effects. Figure 7(d) shows the differential profile, which reports significant inflation with four events (cycles, local HITM events, remote HITM events, and remote DRAM accesses) in the parallel execution when compared to an oracle execution. The differential analysis profile in Figure 7(d) shows the inflation in top three `parallel_for` regions. On examining them, we found that all the `parallel_for` calls were being made multiple times in a sequential loop. An inflation in remote DRAM accesses in the differential analysis profile made us suspect that TBB's work stealing scheduler was likely mapping tasks operating on the same data items from multiple invocations of the `parallel_for` to different processors. We explored techniques to maintain affinity between the data and the processor performing computations on the same data over multiple invocations of a `parallel_for` call. We used TBB's affinity partitioner that provides best-effort affinity by mapping the iterations of `parallel_for` to the same thread that executed it previously. We changed the six `parallel_for` calls to use TBB's affinity partitioner. Figure 7(e) shows the differential profile after this optimization. It shows a significant decrease for all the four performance counter events. The profile still shows some inflation because the affinity partitioner is a best-effort technique. After this optimization, the speedup of the program improved from 5.76× to 5.98×. In summary, the profiler's what-if analyses and differential analyses helped us increase the speedup from 2.2× to 5.98×.

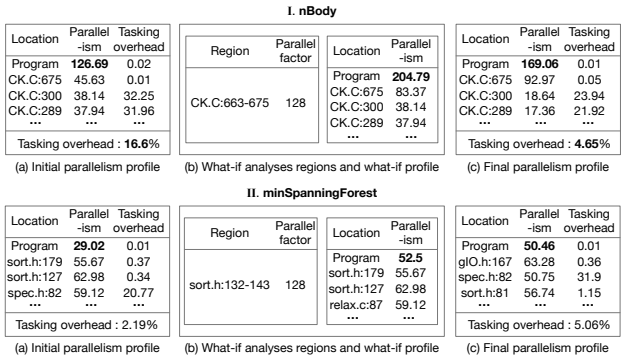


Figure 9. The initial parallelism profile with tasking overheads, the regions identified using the what-if analyses, and the parallelism profile after parallelizing the regions reported for two applications: `nBody`, and `minSpanningForest`.

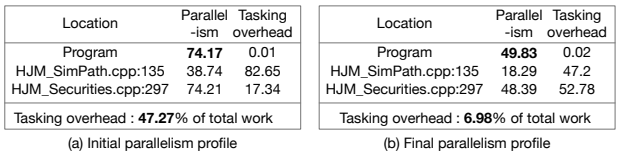


Figure 10. (a) The parallelism profile for `swaptions` that highlights high tasking overhead. (b) The profile after reducing the tasking overhead by increasing the grain size at `HJM_SimPath.cpp:135`.

Increasing the speedup of `LULESH`. LULESH [33] is an application from LLNL that is widely used to model hydrodynamics in scientific applications. The program had a speedup of 4.17× on a 16-core machine. The parallelism of the program was 43.01. We wanted to understand the reason behind low speedup even when the program has a parallelism of 43.01. We profiled the program with TaskProf2's differential analysis. Figure 8(a) shows the differential profile generated by it. Overall, the program has 4.19× inflation in cycles when compared to serial execution. The profile also has significant inflation in last level cache misses and remote memory accesses. The profile highlights two `parallel_for` calls (llesh.c:2823 and llesh.c:2847 in Figure 8(a)) having high inflation in last level cache (LLC) misses and remote DRAM accesses. Further, these `parallel_for` calls were performing almost 30% of the work on the critical path. Since these regions had high inflation in LLC misses, we checked whether the working set of the program was larger than the LLC during parallel execution. We noticed that both the `parallel_for` regions were performing computations on two large arrays. Further investigation revealed that accessing these arrays in parallel is the root cause of significant LLC misses. We rearranged the computation to reduce the working set size while ensuring that the transformation was correct. Figure 8(b) shows the reduction in inflation for all events after this optimization. The optimization improved the speedup of the program to 5.86×.

Increasing the speedup of `nBody`. The nBody application takes an array of 3-D points as input and computes the gravitational force vector of each point due to all other points. The initial speedup on a 16-core machine was 12.26×. Unsurprisingly, our prototype showed that the program has high parallelism (126.69 in Figure 9(I)(a)). However, our prototype also reports a relatively high tasking overhead (16.6% of total useful work in Figure 9(I)(a)) and three spawn sites corresponding to `parallel_for` calls accounting for 90% of this tasking overhead. The profile in Figure 9(I)(a) shows two such `parallel_for` calls at `CK.C:300`, `CK.C:289` (we omit the third one due to space constraints). On careful examination of the code, we observed that these `parallel_for` calls were nested within other parallel tasks and they were using TBB's default partitioner, which was partitioning the iteration sub-optimally. We changed the code to use a simple partitioner and increased the cut-off until there was reduction in the tasking overhead. Eventually, we reduced the tasking overhead to 4% and the speedup increased to 13.88×.

Subsequently, the prototype's what-if analyses identified a region of the code that when parallelized can increase the parallelism to 204.79 (see Figure 9(I)(b)). It corresponded to the body of a `parallel_for` call at `CK.C:675`, which performs 80% of the work on the critical path (see Figure 9(I)(a)). We decreased the cut-off to reduce the serial work done by the body of the `parallel_for` call at `CK.C:675`. Figure 9(I)(c) reports that the parallelism of the program improved to 169.06 and the tasking overhead reduced to 4.65% (Figure 9(I)(c)). The speedup of the program improved from 12.26× to 14.11×. In summary, we had to increase the cut-off for some spawn sites and decrease the cut-off with a few others to improve speedup. TaskProf2 helped us resolve the trade-off between parallelism and tasking overhead.

Increasing the speedup of `minSpanningForest`. This program in the PBBS suite is a parallel implementation of Kruskal's minimum spanning tree algorithm. The initial speedup of the program is 6.43×. Figure 9(II)(a) presents the parallelism profile that reports the program has a parallelism of 29.02. Our what-if analyses identified one region that when parallelized can increase parallelism to 52.5 (Figure 9(II)(b)) without increasing tasking overheads. The reported region was partitioning edges into multiple blocks sequentially. Since there were no data dependencies, we determined that the partitioning could be done in parallel. We parallelized it by recursively spawning tasks in parallel. After this optimization, the parallelism of the program increased to 50.46 (Figure 9(II)(c)) and the speedup of the program increased to 9.8×.

Increasing the speedup of `swaptions`. This program from the Parsec benchmark suite has an initial speedup

of 12.25× on our 16-core machine. Although this program has relatively higher speedup, the profiler reported that the program has a tasking overhead of 47.27% (Figure 10(a)). The parallelism profile highlights the `parallel_for` call at `HJM_SimPath.cpp:135`, which accounts for 80% of the tasking overhead. On examining the code, we found that the cut-off for recursive decomposition was too small. We increased the cut-off with the help of the profiler until the overall tasking overhead reduced to less than 10%. Figure 10(b) presents the parallelism profile after reducing the tasking overhead. The speedup of the program increased to 14.16×. The prototype's what-if analyses subsequently reported that the program cannot be parallelized any further without increasing the tasking overhead.

Other applications. Similar to `minSpanningForest`, our profiler's what-if analyses identified serialization bottlenecks in four other applications. We designed concrete parallelization strategies for the regions reported and increased the speedup from 6.77× to 8.82× in `breadthFirstSearch`, from 7.17× to 8.27× in `spanningForest`, from 2.15× to 7.58× in `suffixArray`, and from 4.9× to 6.32× in `comparisonSort`. Overall, in these four applications, the regions reported by our what-if analyses were precisely the regions that we eventually parallelized.

6.4 Evaluation with Other Profilers

To highlight the effectiveness of TaskProf2, we also evaluated all nine applications that we sped up with three other profilers: Coz [17], Intel Advisor [14], and Intel Vtune Amplifier [15]. Our evaluation data is available online [72].

Evaluation with Coz. Coz quantifies the amount of possible speedup when a line of code is optimized. The programmer is expected to annotate the code with progress points for throughput or latency. It provides three modes: (1) end-to-end sampling, (2) latency profiling, and (3) throughput profiling. Typically, a programmer does not a priori know the bottlenecks. Hence, we ran all nine applications with Coz without any progress points. After we ran each application numerous times (at least 20 to 80 times), Coz identified a progress point that will either improve the speedup or cause slowdowns by 1-6% except with `swaptions` where it reported a maximum speedup of 30%. When we annotated these regions with latency and throughput progress points, Coz also reported very little speedup. It was also not useful feedback as we could not optimize those regions. Moreover, we observed that the speedup estimates from Coz can vary depending on the choice of progress points. We found it difficult to identify appropriate progress points to obtain consistent speedup estimates with Coz [69]. As we had already identified regions that matter with TaskProf2 and we are interested in reducing the execution time of the program, we used latency progress points in Coz for the regions identified by our profiler. Strangely, the latency profile reported a slowdown for all the annotated regions [70]. Finally, we used throughput progress points with the regions reported by our profiler. Out of the nine applications, Coz with throughput progress points reports a possible speedup with five applications (*i.e.*, `breadthFirstSearch`, `minSpanningForest`, `comparisonSort`, `suffixArray`, and `nBody`). The speedup reported was too pessimistic with `suffixArray`. Coz reported a speedup of 20% in the best case. However, we improved the speedup by 352% when we concretely parallelized it. The speedup estimates by Coz appear reasonable when the progress point is within loops and is frequently executed and also matters in improving whole-program performance (*e.g.*, `nBody`) . Coz did not report any throughput profile for the `spanningForest` application even after executing it numerous times. Coz did not provide useful information with speedup estimates for addressing secondary effects (`MILCmk`, `LULESH`, and `swaptions`). We also had to run the program numerous times to get throughput profiles, which can be slow and time-consuming.

Evaluation with Intel Advisor. Intel Advisor aims to provide feedback on threading bottlenecks. It consists of a survey analysis that identifies where the program spends its time. Subsequently, programmers can use annotations to check the suitability of a region of code for optimization to address load imbalance, lock contention, or runtime overheads. Similar to our Coz experiments, we first ran the survey analysis for each of the nine applications and identified the regions reported by it. We used Intel Advisor's annotations to mark these regions for optimization. Intel Advisor highlighted serial and parallel loops in the program. We optimized some of these regions but it did not increase the speedup. We could not identify a way to parallelize other loops. Also, these were not the regions reported by TaskProf2.

Subsequently, we used the regions identified by TaskProf2 for annotations. We used Intel Advisor's "check suitability" analysis, which estimates the improvement in speedup of the region and the entire program. Of the seven applications that had parallelism bottlenecks, Intel Advisor reports a speedup in the range 28%-65% for four applications [72]. If the annotations are within loops and are hot-spots, Intel Advisor predicts a reasonable speedup (*e.g.*, `nBody` application). For the remaining three applications Intel Advisor reports a nominal speedup in the range of 1%-6% [72]. These programs contained recursive functions in the annotated regions. Overall, we found that Intel Advisor's speedup prediction is useful if the annotated regions are within loops and are also hot-spots in the program. Its use is limited in identifying regions that one should explore.

Evaluation with Intel VTune. Intel VTune Amplifier is a comprehensive commercial profiler. It provides feedback on frequently executed regions and regions experiencing scheduling overheads or secondary effects. VTune identified that six of our nine applications have low parallelism based on CPU utilization and highlighted the hot-spots. Those regions were primarily `parallel_for` parts of the code and

were not useful in improving the speedup. We used VTune's micro-architectural analysis which reported that five applications were experiencing secondary effects of execution. While the analysis highlighted various causes for the secondary effects in the programs, it did not provide specific code regions that were experiencing them. VTune identified scheduling overheads in two of the three applications identified by TaskProf2. It did not identify the scheduling overhead in nBody. Overall, we found VTune is a good tool to identify hot-spots, secondary effects, and scheduling overheads. However, it does not report whether optimizing these hot-spots matter in improving parallelism.

7 Related Work

Profiling parallel programs is a widely researched topic. In this section, we compare the most closely related work.

Serialization and tasking overheads tools. Prior research on profiling task parallel programs has explored quantifying the parallelism in the program to identify serialization bottlenecks. CilkView [26], CilkProf [59] and TaskProf [68] compute the parallelism of task parallel programs and attribute the parallelism to the static spawn sites in the program. Our on-the-fly algorithm resembles CilkProf's algorithm. Unlike CilkProf that requires serial execution, our profile execution runs in parallel. OMP-WhIP [7] computes the parallelism in OpenMP programs. Our prior work, TaskProf and OMP-WhIP, can estimate the improvement in parallelism when the programmer annotates a region. However, they do not consider tasking overheads. Numerous approaches have explored profiling and compiler techniques to identify task granularity cut-off points in isolation [25, 30, 56, 57].

Improving parallelism or addressing runtime overheads in isolation is not sufficient to improve the speedup. The user has to carefully balance parallelism and the cost of creating tasks to obtain good performance.

Performance estimation tools. Some early tools identify functions that execute on the critical path [6, 48, 53]. Other early tools have also proposed metrics to estimate improvements in execution time on optimizing certain functions [27, 48]. Kremlin [24] and Kismet [32] quantify the parallelism in a sequential program and estimate the speedup that can be obtained from parallelization. Coz [17] quantifies the effect of optimizing a line of code by inserting delays in concurrently executing threads. It can be considered as a kind of what-if analysis. We found that Coz reported pessimistic estimates of speedup with task parallel programs and it is challenging to identify where to optimize. Intel Advisor [14] is a useful commercial tool in estimating the speedup of a piece of annotated code and the entire program with multiple programming models. Similar to Coz, figuring out where to annotate is challenging with Intel Advisor.

Tools identifying secondary effects. Techniques to identify secondary effects in parallel program execution have been widely studied. They include tools to detect cache contention [8, 10, 21, 31, 36, 37, 44, 49, 65, 74], identify data locality issues [38–40, 42], and find bottlenecks due to NUMA [41, 46]. Unlike our profiler, these tools are tailored to detect specific kinds of secondary effects. Further, they either use binary instrumentation or sampling. TaskProf2 does not explicitly sample while performing measurements (PMU units employ precise event sampling).

Differential profiling. Prior profilers for parallel programs have also compared profiles from multiple executions using expectations [28, 61]. They classify anything that differs from expected values as bottlenecks or performance issues [13, 47]. SCAAnalyzer [43] performs sampling and compares latencies of data objects from two executions to identify memory scalability bottlenecks. They do not provide feedback on specific regions experiencing secondary effects.

Work inflation. There are tools that measure work inflation to identify if the program is experiencing secondary effects [3, 51, 58]. But these tools neither pinpoint parts of the program with secondary effects nor its relation to program parallelism. In contrast, our differential analyses can highlight regions that are experiencing any type of secondary effect and probable reasons for them by examining inflation in various metrics of interest.

Performance modeling. Profilers have explored performance modeling to predict performance on a large number of cores [9, 55] and identify critical paths in parallel programs [4, 60]. The Roofline model is a visual performance model to identify the upper bound of a kernel's performance on a given machine by measuring metrics such as floating point performance, off-chip and on-chip memory traffic [29, 45, 66]. However, these approaches do not provide feedback on concrete program regions to focus on.

8 Conclusion

TaskProf2 identifies program regions with serialization bottlenecks, tasking overheads, and significant inflation in work due to secondary effects with the help of a novel performance model of a task parallel execution. This performance model consists of series-parallel relationships between task fragments and fine-grained measurement of computation. It enables TaskProf2 to report regions that matter in increasing parallelism using what-if analyses and identify regions that experience secondary effects of execution using differential analyses. TaskProf2 can enable effective performance debugging of task parallel programs. We would like to see the integration of TaskProf2 into widely used profilers.

Acknowledgments

We thank our shepherd Ayal Zaks and the anonymous reviewers for their feedback. This paper is based on work supported in part by NSF CAREER Award CCF–1453086.

References

[1] [n. d.]. Coral benchmarks. https://asc.llnl.gov/CORAL-benchmarks/.

[2] Umut A. Acar, Guy E. Blelloch, and Robert D. Blumofe. 2000. The Data Locality of Work Stealing. In *Proceedings of the Twelfth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*. 1–12.

[3] U. A. Acar, A. Charguéraud, and M. Rainey. 2017. Parallel Work Inflation, Memory Effects, and their Empirical Analysis. *ArXiv e-prints* (2017).

[4] Cedell Alexander, Donna Reese, and James C. Harden. 1994. Near-Critical Path Analysis of Program Activity Graphs. In *Proceedings of the Second International Workshop on Modeling, Analysis, and Simulation On Computer and Telecommunication Systems (MASCOTS)*. 308–317.

[5] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 72–81.

[6] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. 1995. Cilk: An Efficient Multithreaded Runtime System. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)*. 207–216.

[7] Nader Boushehrinejadmoradi, Adarsh Yoga, and Santosh Nagarakatte. 2018. A Parallelism Profiler with What-if Analyses for OpenMP Programs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC)*. 16:1–16:14.

[8] Bevin Brett, Pranith Kumar, Minjang Kim, and Hyesoon Kim. 2013. CHiP: A Profiler to Measure the Effect of Cache Contention on Scalability. In *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing Workshops and PhD Forum (IPDPSW)*. 1565–1574.

[9] Alexandru Calotoiu, Torsten Hoefler, Marius Poke, and Felix Wolf. 2013. Using Automated Performance Modeling to Find Scalability Bugs in Complex Codes. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*. 45:1–45:12.

[10] Milind Chabbi, Shasha Wen, and Xu Liu. 2018. Featherlight On-the-fly False-sharing Detection. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*. 152–167.

[11] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. 2005. X10: An Object-oriented Approach to Non-uniform Cluster Computing. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 519–538.

[12] Guang-Ien Cheng, Mingdong Feng, Charles E. Leiserson, Keith H. Randall, and Andrew F. Stark. 1998. Detecting Data Races in Cilk Programs That Use Locks. In *Proceedings of the 10th ACM Symposium on Parallel Algorithms and Architectures (SPAA)*. 298–309.

[13] Cristian Coarfa, John Mellor-Crummey, Nathan Froyd, and Yuri Dotsenko. 2007. Scalability Analysis of SPMD Codes Using Expectations. In *Proceedings of the 21st Annual International Conference on Supercomputing (ICS '07)*. 13–22.

[14] Intel Corporation. 2019. Intel Advisor. Retrieved March 20, 2019 from https://software.intel.com/en-us/advisor

[15] Intel Corporation. 2019. Intel VTune Amplifier. Retrieved March 20, 2019 from https://software.intel.com/en-us/intel-vtune-amplifier-xe

[16] Intel Corporation. 2019. Official Intel(R) Threading Building Blocks (Intel TBB) GitHub repository. Retrieved Apr 5, 2019 from https://github.com/01org/tbb

[17] Charlie Curtsinger and Emery D. Berger. 2015. Coz: Finding Code That Counts with Causal Profiling. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP)*. 184–197.

[18] Luiz DeRose, Bill Homer, and Dean Johnson. 2007. Detecting Application Load Imbalance on High End Massively Parallel Systems. In *Proceedings of the 13th International Euro-Par Conference on Parallel Processing (Euro-Par)*. 150–159.

[19] Kristof Du Bois, Stijn Eyerman, Jennifer B. Sartor, and Lieven Eeckhout. 2013. Criticality Stacks: Identifying Critical Threads in Parallel Programs Using Synchronization Behavior. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA)*. 511–522.

[20] Kristof Du Bois, Jennifer B. Sartor, Stijn Eyerman, and Lieven Eeckhout. 2013. Bottle Graphs: Visualizing Scalability Bottlenecks in Multi-threaded Applications. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*. 355–372.

[21] Ariel Eizenberg, Shiliang Hu, Gilles Pokam, and Joseph Devietti. 2016. Remix: Online Detection and Repair of Cache Contention for the JVM. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 251–265.

[22] Stijn Eyerman, Kristof Du Bois, and Lieven Eeckhout. 2012. Speedup Stacks: Identifying Scaling Bottlenecks in Multi-threaded Applications. In *Proceedings of the 2012 IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS)*. 145–155.

[23] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. 1998. The Implementation of the Cilk-5 Multithreaded Language. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation (PLDI)*. 212–223.

[24] Saturnino Garcia, Donghwan Jeon, Christopher M. Louie, and Michael Bedford Taylor. 2011. Kremlin: Rethinking and Rebooting Gprof for the Multicore Age. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 458–469.

[25] Suyash Gupta, Rahul Shrivastava, and V Krishna Nandivada. 2017. Optimizing Recursive Task Parallel Programs. In *Proceedings of the International Conference on Supercomputing (ICS)*. 11:1–11:11.

[26] Yuxiong He, Charles E. Leiserson, and William M. Leiserson. 2010. The Cilkview Scalability Analyzer. In *Proceedings of the Twenty-second Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 145–156.

[27] Jeffrey K. Hollingsworth and Barton P. Miller. 1994. *Slack: A New Performance Metric for Parallel Programs.* Technical Report. University of Wisconsin-Madison.

[28] R. Hood, H. Jin, P. Mehrotra, J. Chang, J. Djomehri, S. Gavali, D. Jespersen, K. Taylor, and R. Biswas. 2010. Performance impact of resource contention in multicore systems. In *2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS)*. 1–12.

[29] Aleksandar Ilic, Frederico Pratas, and Leonel Sousa. 2014. Cache-aware Roofline Model: Upgrading the Loft. *IEEE Computer Architecture Letters* (2014), 21–24.

[30] Shintaro Iwasaki and Kenjiro Taura. 2016. A Static Cut-off for Task Parallel Programs. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation (PACT)*. 139–150.

[31] Sanath Jayasena, Saman Amarasinghe, Asanka Abeyweera, Gayashan Amarasinghe, Himeshi De Silva, Sunimal Rathnayake, Xiaoqiao Meng, and Yanbin Liu. 2013. Detection of False Sharing Using Machine Learning. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*. 30:1–30:9.

[32] Donghwan Jeon, Saturnino Garcia, Chris Louie, and Michael Bedford Taylor. 2011. Kismet: Parallel Speedup Estimates for Serial Programs. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*. 519–536.

[33] Lawrence Livermore National Labs. 2018. Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics (LULESH). Retrieved November 17, 2018 from https://computation.llnl.gov/projects/co-design/lulesh

[34] Doug Lea. 2000. A Java Fork/Join Framework. In *Proceedings of the ACM 2000 Conference on Java Grande (JAVA)*. 36–43.

[35] Daan Leijen, Wolfram Schulte, and Sebastian Burckhardt. 2009. The Design of a Task Parallel Library. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*. 227–242.

[36] Tongping Liu and Emery D. Berger. 2011. SHERIFF: Precise Detection and Automatic Mitigation of False Sharing. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*. 3–18.

[37] Tongping Liu, Chen Tian, Ziang Hu, and Emery D. Berger. 2014. PREDATOR: Predictive False Sharing Detection. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*. 3–14.

[38] Xu Liu and John Mellor-Crummey. 2011. Pinpointing Data Locality Problems Using Data-centric Analysis. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 171–180.

[39] Xu Liu and John Mellor-Crummey. 2013. A Data-centric Profiler for Parallel Programs. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*. 28:1–28:12.

[40] Xu Liu and John Mellor-Crummey. 2013. Pinpointing data locality bottlenecks with low overhead. In *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 183–193.

[41] Xu Liu and John Mellor-Crummey. 2014. A Tool to Analyze the Performance of Multithreaded Programs on NUMA Architectures. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*. 259–272.

[42] Xu Liu, Kamal Sharma, and John Mellor-Crummey. 2014. ArrayTool: A Lightweight Profiler to Guide Array Regrouping. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation (PACT)*. 405–416.

[43] Xu Liu and Bo Wu. 2015. ScaAnalyzer: A Tool to Identify Memory Scalability Bottlenecks in Parallel Programs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. 47:1–47:12.

[44] Liang Luo, Akshitha Sriraman, Brooke Fugate, Shiliang Hu, Gilles Pokam, Chris J. Newburn, and Joseph Devietti. 2016. LASER: Light, Accurate Sharing dEtection and Repair. In *2016 IEEE International Symposium on High Performance Computer Architecture, HPCA 2016, Barcelona, Spain, March 12-16, 2016*. 261–273.

[45] D. Marques, H. Duarte, A. Ilic, L. Sousa, R. Belenov, P. Thierry, and Z. A. Matveev. 2017. Performance Analysis with Cache-Aware Roofline Model in Intel Advisor. In *2017 International Conference on High Performance Computing Simulation (HPCS)*. 898–907.

[46] Collin McCurdy and Jeffrey Vetter. 2010. Memphis: Finding and fixing NUMA-related performance problems on multi-core platforms. In *2010 IEEE International Symposium on Performance Analysis of Systems Software (ISPASS)*. 87–96.

[47] Paul E. McKenney. 1999. Differential Profiling. *Software - Practice & Experience* (1999), 219–234.

[48] B. P. Miller, M. Clark, J. Hollingsworth, S. Kierstead, S. S. Lim, and T. Torzewski. 1990. IPS-2: The Second Generation of a Parallel Program Measurement System. *IEEE Transactions on Parallel and Distributed Systems* (1990), 206–217.

[49] Mihir Nanavati, Mark Spear, Nathan Taylor, Shriram Rajagopalan, Dutch T. Meyer, William Aiello, and Andrew Warfield. 2013. Whose Cache Line is It Anyway?: Operating System Support for Live Detection and Repair of False Sharing. In *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys)*. 141–154.

[50] Jungju Oh, Christopher J. Hughes, Guru Venkataramani, and Milos Prvulovic. 2011. LIME: A Framework for Debugging Load Imbalance in Multi-threaded Execution. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE)*. 201–210.

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

[52] OpenMP Architecture Review Board. 2015. OpenMP 4.5 Complete Specification. http://www.openmp.org/wp-content/uploads/openmp-4.5.pdf

[53] Yoshihiro Oyama, Kenjiro Taura, and Akinori Yonezawa. 2000. Online Computation of Critical Paths for Multithreaded Languages. In *Proceedings of the 15 IPDPS 2000 Workshops on Parallel and Distributed Processing (IPDPS)*. 301–313.

[54] Raghavan Raman, Jisheng Zhao, Vivek Sarkar, Martin Vechev, and Eran Yahav. 2012. Scalable and Precise Dynamic Datarace Detection for Structured Parallelism. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 531–542.

[55] Patrick Reisert, Alexandru Calotoiu, Sergei Shudler, and Felix Wolf. 2017. Following the Blind Seer – Creating Better Performance Models Using Less Information. In *Euro-Par 2017: Parallel Processing*.

[56] Andrea Rosà, Eduardo Rosales, and Walter Binder. 2018. Analyzing and Optimizing Task Granularity on the JVM. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization (CGO)*. 27–37.

[57] E. Rosales, A. Rosà, and W. Binder. 2017. tgp: A Task-Granularity Profiler for the Java Virtual Machine. In *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*. 570–575.

[58] Mark Roth, Micah Best, Craig Mustard, and Alexandra Fedorova. 2012. Deconstructing the overhead in parallel applications. In *Proceedings - 2012 IEEE International Symposium on Workload Characterization, IISWC 2012 (IISWC)*. 59–68.

[59] Tao B. Schardl, Bradley C. Kuszmaul, I-Ting Angelina Lee, William M. Leiserson, and Charles E. Leiserson. 2015. The Cilkprof Scalability Profiler. In *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 89–100.

[60] M. Schulz. 2005. Extracting Critical Path Graphs from MPI Applications. In *2005 IEEE International Conference on Cluster Computing*. 1–10.

[61] Martin Schulz and Bronis R. de Supinski. 2007. Practical Differential Profiling. In *Euro-Par 2007 Parallel Processing: 13th International Euro-Par Conference, Rennes ,France , August 28-31, 2007. Proceedings*. 97–106.

[62] Julian Shun, Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, Aapo Kyrola, Harsha Vardhan Simhadri, and Kanat Tangwongsan. 2012. Brief Announcement: The Problem Based Benchmark Suite. In *Proceedings of the 24th Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 68–70.

[63] Nathan R. Tallent, Laksono Adhianto, and John M. Mellor-Crummey. 2010. Scalable Identification of Load Imbalance in Parallel Executions Using Call Path Profiles. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. 1–11.

[64] Nathan R. Tallent and John M. Mellor-Crummey. 2009. Effective Performance Measurement and Analysis of Multithreaded Applications. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*. 229–240.

[65] Shasha Wen, Xu Liu, John Byrne, and Milind Chabbi. 2018. Watching for Software Inefficiencies with Witch. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 332–347.

[66] Samuel Williams, Andrew Waterman, and David Patterson. 2009. Roofline: An Insightful Visual Performance Model for Multicore Architectures. *Commun. ACM* (2009), 65–76.

[67] Adarsh Yoga and Santosh Nagarakatte. 2016. Atomicity Violation Checker for Task Parallel Programs. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization (CGO)*. 239–249.

[68] Adarsh Yoga and Santosh Nagarakatte. 2017. A Fast Causal Profiler for Task Parallel Programs. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*. 15–26.

[69] Adarsh Yoga and Santosh Nagarakatte. 2019. Issue 103:Where to place progress points. Retrieved Mar 28, 2019 from https://github.com/plasma-umass/coz/issues/103

[70] Adarsh Yoga and Santosh Nagarakatte. 2019. Issue 104:Inconsistent results for throughput and latency profiling. Retrieved Mar 28, 2019 from https://github.com/plasma-umass/coz/issues/104

[71] Adarsh Yoga and Santosh Nagarakatte. 2019. TaskProf2. Retrieved Apr 5, 2019 from https://github.com/rutgers-apl/TaskProf2.git

[72] Adarsh Yoga and Santosh Nagarakatte. 2019. TaskProf2-Evaluation data. Retrieved Apr 5, 2019 from https://github.com/rutgers-apl/TaskProf2/tree/master/pldi_comparison_results

[73] Adarsh Yoga, Santosh Nagarakatte, and Aarti Gupta. 2016. Parallel Data Race Detection for Task Parallel Programs with Locks. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*. 833–845.

[74] Qin Zhao, David Koh, Syed Raza, Derek Bruening, Weng-Fai Wong, and Saman Amarasinghe. 2011. Dynamic Cache Contention Detection in Multi-threaded Applications. In *Proceedings of the 7th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)*. 27–38.