A Fast Causal Profiler for Task Parallel Programs

Anonymous Author(s)

ABSTRACT

This paper proposes TASKPROF, a profiler that identifies parallelism bottlenecks in task parallel programs that manifest when the program is executed on a large number of processors. TASKPROF computes this profile by fine-grained attribution of work to parts of the program and by leveraging the structure of a task parallel execution. TASKPROF's profile execution runs in parallel using multi-cores. TASKPROF's use of hardware performance counters to perform fine-grained measurements minimizes perturbation. TASKPROF's causal profile enables users to estimate improvements in parallelism by optimizing a region of the program even when concrete optimizations are not known. We have used TASKPROF to isolate parallelism bottlenecks in twenty three applications that use the Intel Threading Building Blocks library. We have designed parallelization techniques in five applications to increase parallelism by an order of magnitude using TASKPROF. Our user study indicates that developers are able to isolate performance bottlenecks with ease using TASKPROF.

ACM Reference format:

Anonymous Author(s). 2017. A Fast Causal Profiler for Task Parallel Programs. In *Proceedings of 11th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, Paderborn, Germany, 4–8 September, 2017 (ESEC/FSE 2017),* 12 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Task parallelism is one approach to write performance portable parallel code, where the performance increases with the increase in the number of processors [20]. In this model, the programmer specifies fine-grained tasks and the work stealing task parallel runtime maps these tasks to processors while automatically balancing the workload between them. Many task parallelism frameworks have become mainstream (*e.g.*, Intel Threading Building Blocks (TBB) [41], Cilk [17], Microsoft Task Parallel Library [27], Habanero Java [5], X10 [6], and Java Fork/Join tasks [26]).

Asymptotic parallelism measures the potential speedup when the program is executed on a large number of processors. It is constrained by the longest chain of tasks that must be executed sequentially (also known as the `span` or the critical work). Hence, asymptotic parallelism is the ratio of the total work and the critical work performed by the program for a given input. A scalable program must have large asymptotic parallelism.

A task parallel program can have low asymptotic parallelism due to multiple factors. First, coarse-grained tasks can result in serialization and provide few opportunities for the work stealing runtime to balance the workload. Second, scheduling overhead can dominate any benefit from parallel execution when tasks are too fine-grained. Third, the program may not have much parallelism for a given input because the total work performed is limited. Fourth, secondary effects of execution such as contention, low locality, and false sharing can limit the performance of parallel execution.

ESEC/FSE 2017, Paderborn, Germany
2017. 978-x-xxxx-xxxx-x/YY/MM...$15.00
DOI: 10.1145/nnnnnnn.nnnnnnn

Numerous techniques have been proposed to address various bottlenecks in both multithreaded programs [7, 11, 13, 29, 30, 32, 46, 47] and task parallel programs [21, 42]. These technique range from identifying critical paths [22, 35, 39], parallelism [21, 42], synchronization bottlenecks [7, 11, 13, 46, 47], and other performance pathologies [29, 30, 32]. Tools for multithreaded programs identify bottlenecks in a specific execution on a specific machine, which does not provide information about scalability of the program. In contrast, tools that measure asymptotic parallelism in task parallel programs run the program serially [21, 42], which is feasible only when the task parallel model provides serial semantics (*e.g.*, Cilk). They can also perturb execution significantly with high overheads of serial profile execution. Although they identify parallelism bottlenecks, they do not provide information on regions of code that matter in improving asymptotic parallelism.

This paper proposes TASKPROF, a fast and a causal profiler that measures asymptotic parallelism in task parallel programs for a given input. TASKPROF has three major goals in profiling task parallel programs : (1) to minimize perturbation while accurately computing asymptotic parallelism and critical work per spawn site (source code location where a task is created), (2) to run the profiler in parallel (*i.e.*, fast), and (3) to provide feedback on regions of code that matter in increasing asymptotic parallelism (*i.e.*, a causal profile) .

To compute an accurate asymptotic parallelism profile, TASKPROF has to perform fine-grained attribution of work to various parts of the program. Performing this fine-grained attribution while minimizing perturbation and executing the profile execution in parallel is challenging. TASKPROF performs this fine-grained attribution leveraging the structure of a task parallel execution. Task parallel programs are structured parallel programs whose execution can be represented as a tree (specifically Dynamic Program Structure Tree (DPST) [40]), which can be constructed in parallel. Given a task parallel program, TASKPROF constructs this DPST representation in parallel during program execution and attributes work to leaves in the DPST. To minimize perturbation, TASKPROF uses hardware performance counters to measure work performed in regions without any task management constructs (spawns or sync), which correspond to leaves in the DPST. TASKPROF writes the DPST and the work performed by the leaf nodes of the DPST to a profile data file. The profile execution runs in parallel leveraging multi-cores and the measurement using performance counters is thread-safe.

TASKPROF's post-execution analysis tool uses the data file from the profile run, reconstructs the DPST, and computes asymptotic parallelism and critical work at each spawn site in the program using the properties of the DPST (see Section 3.2). TASKPROF maps dynamic execution information to static spawn sites by maintaining information about spawn sites in the DPST. TASKPROF's profile for the sample program in Figure 2 is shown in Figure 3(b).

Apart from reporting where the program performs critical work, TASKPROF enables programmers to identify regions of code that

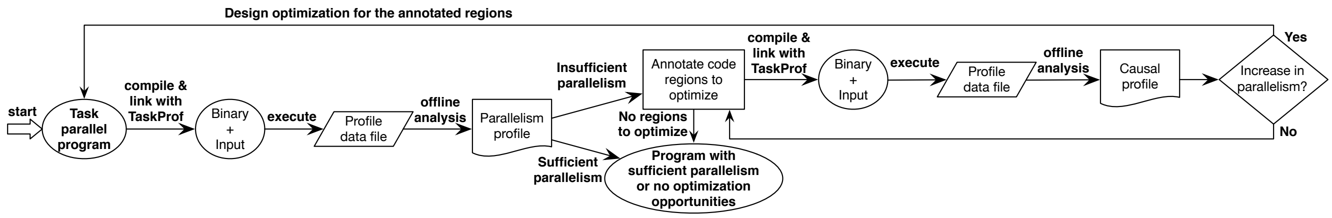


Figure 1: Identifying and diagnosing parallelism bottlenecks in task parallel programs using TASKPROF's parallelism and causal profiles.

matter in increasing asymptotic parallelism. Optimizing regions that perform critical work may not increase asymptotic parallelism when the program has multiple regions that perform similar amount of critical work. Designing a parallelization strategy that reduces critical work requires significant effort. Hence, the programmer would like to know if optimizing a region of code increases asymptotic parallelism even before the specific optimization is designed.

To quantify the impact of optimizing a region of code, the programmer annotates the beginning and the end of the region in the program and the anticipated speedup for the region. TASKPROF generates a causal profile that shows the increase in parallelism with varying amounts of anticipated speedup for the annotated regions (see Figure 3(c)). We designed TASKPROF's causal profile inspired by COZ [10], which provides causal profiles for arbitrary multi-threaded programs but cannot be used with task parallel programs as it is not possible to slow down all active tasks. To generate a causal profile, TASKPROF re-executes the program, generates profile data, and identifies nodes in the DPST that correspond to the annotated regions. Subsequently, TASKPROF recomputes asymptotic parallelism and critical work at each spawn site by reducing the critical work of the annotated region of code by the anticipated improvement. TASKPROF's causal profiling enables the programmer to identify improvements in asymptotic parallelism even before the developer actually designs the optimization. Figure 1 illustrates TASKPROF's usage to generate a parallelism profile and a causal profile.

We have identified asymptotic parallelism bottlenecks in twenty three widely used Intel TBB applications using the TASKPROF prototype. We also designed concrete parallelization techniques for five applications to address parallelism bottlenecks identified using TASKPROF's causal profile. Our concrete optimizations increased asymptotic parallelism in these five applications by an order of magnitude (*e.g.*, from 1.28 to 50.84 for Convex hull). We conducted a user study involving 13 undergraduate and graduate students to evaluate the usability of TASKPROF. Our results show that the participants quickly diagnosed parallelism bottlenecks using TASKPROF. Most participants could not find any bottleneck when they did not use TASKPROF (see Section 5).

In summary, this paper makes the following contributions. (1) A fast and precise profiler with minimal perturbation that computes asymptotic parallelism and serial work at each spawn site of a task parallel program. It is accomplished using fine-grained attribution of work using performance counters and by leveraging the structure of a task parallel execution. (2) A novel approach to generate causal profiles for task parallel programs, which quantifies the impact of optimizing a static region of the program even when the concrete optimization is not known.

2 BACKGROUND

This section provides a quick primer on task parallelism and a tree-based representation of a task parallel execution, which is used by TASKPROF to compute asymptotic parallelism and causal profiles.

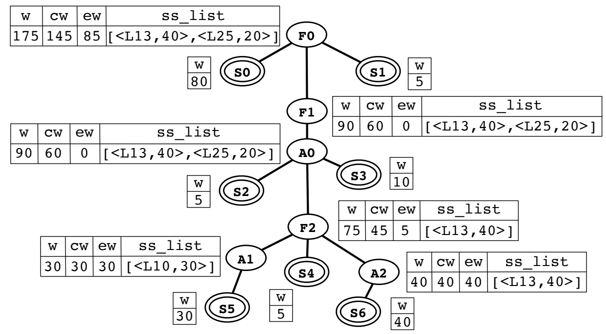
Task parallelism. Task parallelism is a structured parallel programming model that simplifies the job of writing performance portable code. In this model, parallel programs are expressed using a small set of expressive yet structured patterns. In contrast to threads, task creation is inexpensive because a task is always bound to a thread till completion [34]. The work stealing runtime maps dynamic tasks to runtime threads and also balances the workload between threads. Task programming models provide specific constructs to create tasks (*e.g.*, `spawn` keyword in Cilk and `spawn` function in Intel TBB) and to wait for other tasks to complete (*e.g.*, `sync` keyword in Cilk and `wait_for_all()` function in Intel TBB). A sample task parallel program is shown in Figure 2. These models also provide patterns for recursive decomposition of a program (*e.g.*, `parallel_for` and `parallel_reduce`) that are built using the basic constructs. Task parallelism is expressive and widely applicable for writing structured parallel programs. However, task parallelism is a poor fit for event-based coordination patterns, which are unstructured and unpredictable [33].

Structure of a task parallel execution. Unlike an arbitrary multithreaded program, the execution of a task parallel program can be represented either as a directed acyclic graph (DAG) [16] or as a tree [40]. In the DAG representation of a task parallel execution, a node in the DAG represents task management constructs (*e.g.*, `spawn` or `sync`) and an edge represents a collection of instructions executed without any task management constructs [16, 17]. Such a DAG representation can be constructed in a serial depth-first fashion when the program has serial semantics (*e.g.*, Cilk programs that have serial semantics). Many task parallel frameworks (*e.g.*, Intel TBB, X10, Java ForkJoinTasks) do not provide serial semantics. Further, algorithms to construct the DAG representation while executing the program in parallel have not been studied.

Dynamic program structure tree. The dynamic program structure tree (DPST) is an alternative representation that precisely captures the series-parallel relationships between tasks [40]. Further, the DPST can be constructed in parallel. Since our goal in this paper is to profile the program in parallel, we use the DPST representation of a task parallel execution.

```
1  void compute_tree_sum(node* n, int* sum) {
2    if(n->num_nodes <= BASE) {
3      //Compute sum serially
4      __CAUSAL_BEGIN__
5      *sum = serial_tree_sum(n);
6      __CAUSAL_END__
7    } else {
8      int left_sum, right_sum;
9      if(n->left) {
10       spawn compute_tree_sum(n->left, &left_sum);
11     }
12     if(n->right) {
13       spawn compute_tree_sum(n->right, &right_sum);
14     }
15     sync;
16     *sum = left_sum + right_sum;
17   }
18 }
19 int main() {
20   __CAUSAL_BEGIN__
21   node* root = create_tree();
22   __CAUSAL_END__
23   int sum;
24   spawn compute_tree_sum(root, &sum);
25   sync;
26   //print sum;
27   return 0;
28 }
```

Figure 2: An example task parallel program that computes the sum of the values in a binary tree. It creates tasks and waits for tasks to complete using `spawn` and `sync` keywords, respectively. Each node in the tree holds a integer value, number of nodes in the sub-tree rooted at the node, and pointers to the left and right sub-tree. The `create_tree` function builds the tree. The `serial_tree_sum` takes a node `n` as argument and computes the sum of the nodes in the sub-tree under `n`. The BASE is a constant that controls the maximum number of nodes whose sum is computed serially. The user has provided annotations (`__CAUSAL_BEGIN__` and `__CAUSAL_END__`) to specify regions for causal profiling, which are not used in the regular profiling phase.



(a) DPST

	Line number	Work	Span	Parallelism	Critical path percent
★	L20	175	145	1.21	58.62
	L13	40	40	1	27.58
	L25	90	60	1.5	13.79
	L10	30	30	1	0

(b) Parallelism profile

Optimization factor	Work	Span	Parallelism
2X	175	85	2.06
4X	175	55	3.18
8X	175	40	4.38

(c) Causal profile

Figure 3: (a) The DPST for an execution of the program in Figure 2. $F0$, $F1$, and $F2$ are finish nodes. $A0$, $A1$, and $A2$ are async nodes. Step nodes are leaves in the DPST. TaskProf maintains four quantities with each intermediate node in the DPST: work (`w`), critical work (`cw`), exclusive work (`ew`), and the list of spawn sites performing critical work (`ss_list`). Each entry in the spawn site list maintains the line number and the exclusive work done by the spawn site (*e.g.*, $< L20, 40 >$). Step nodes have work data from the profile execution. TaskProf updates these quantities for the intermediate nodes by performing a bottom-up traversal of the DPST. (b) The profile generated by TaskProf reports the work, critical work, parallelism, and percentage of critical work with each spawn site. Line with a "$\star$" in the profile corresponds to the main function and reports the parallelism for the entire program. (c) The causal profile reports the parallelism for the whole program when the annotated regions in Figure 2 are optimized by 2×, 4×, and 8×.

The DPST is a n-ary tree representation of a task parallel execution. There are three kinds of nodes in a DPST: (1) `step`, (2) `async`, and (3) `finish` nodes. The step node represents the sequence of dynamic instructions without any task spawn or sync statements. All computations occur in step nodes. The async node in the DPST represents the creation of a child task by a parent task. The descendants of the newly created task can execute in parallel with the remainder of the parent task. A finish node is created in a DPST when a task spawns a child task and waits for the child (and its descendants) to complete. A finish node is the parent of all async, finish and step nodes directly executed by its children or their descendants.

The DPST, by construction, ensures that two parallel tasks operate on two disjoint sub-trees. DPST's construction also ensures that all internal nodes are either async or finish nodes. The siblings of a particular node in a DPST are ordered left-to-right to reflect the left-to-right sequencing of computation of their parent task. A path from a node to the root and the left-to-right ordering of siblings in a DPST do not change even when nodes are added to the DPST during execution. The DPST was originally proposed for data race detection because it allows a race detector to check if two accesses can occur in parallel. In a DPST, two step nodes $S1$ and $S2$ (assuming $S1$ is to the left of $S2$) can execute in parallel if the least common ancestor of $S1$ and $S2$ in the DPST has an immediate child that is an async node and is also an ancestor of $S1$. In Section 3.2, we will highlight the properties of the DPST that we use to profile programs.

Illustration of the DPST. Figure 3(a) shows the DPST for an execution of the program in Figure 2. Let's consider a scenario where BASE in Figure 2 is equal to half the number of nodes in the tree whose sum is being computed. Given such an input, the program in Figure 2 will execute the spawn calls at lines 10 and 13 once.

We construct the DPST during program execution as follows. When the main function starts, we add a finish node $F0$ as the root of the DPST to represent the fact that main completes after all the tasks spawned by it have completed. We add a step node $S0$ as the child of the root finish node to capture the initial computations being performed in the main function. On a spawn call at line 25 in Figure 2, we create a finish node $F1$ because it is the first spawn performed by the task. We also add an async node $A0$ as the child of $F1$ to represent the spawning of a task. Any computation by the newly created task will be added as nodes in the sub-tree under the async node $A0$. The operations performed in the continuation of the main task will be added to the right of the async node $A0$ under the finish node $F1$. Hence, the continuation of the main task and newly created task can update the DPST in parallel. Step nodes $S5$ and $S6$ can execute in parallel because the least common ancestor of $S5$ and $S6$ is a finish node $F2$ and its immediate child, that is also an ancestor of $S5$, is an async node $A1$. We will use these properties to determine critical work in the next section.

3 PARALLELISM PROFILER

TASKPROF computes the total work, part of the total work done serially (critical work or span), and the asymptotic parallelism at each spawn site in a task parallel program. The key contribution of TASKPROF is the fine-grained attribution of work while ensuring that the profile execution is fast, perturbation-free, and accurate. TASKPROF accomplishes the goal of fast profile execution by using multi-cores. TASKPROF's profile execution itself runs in parallel and leverages the DPST representation to attribute work to various parts of the program. TASKPROF ensures that the profile execution is perturbation-free by using hardware performance counters to obtain information about the computation performed by the step nodes in the DPST. TASKPROF also maintains a very small fraction of the DPST in memory during profile execution to minimize perturbation. TASKPROF ensures that the asymptotic parallelism profile is accurate by capturing spawn sites through compiler instrumentation and by precisely attributing work done in each sequence of instructions without any task management constructs.

TASKPROF computes the parallelism profile in three steps. First, TASKPROF provides a modified task parallel library that captures information about spawn sites. TASKPROF's compiler instrumentation modifies the calls to the task parallel library in the program to provide information about spawn sites. Second, TASKPROF's profile execution runs in parallel on the multi-core processors, constructs the DPST representation of the execution, and collects fine-grained information about the execution using hardware performance counters. TASKPROF writes the profile information to a data file similar to the `grof` profiler for sequential programs [19]. Third, TASKPROF's offline analysis tool analyzes the profile data and aggregates the data for each static spawn site. Finally, it computes asymptotic parallelism and critical work for each spawn site. Figure 1 illustrates the use of the profiler by a programmer to find parallelism bottlenecks.

Static instrumentation. TASKPROF uses static instrumentation to instrument the program with calls to the modified task parallel runtime library. In the final offline analysis phase, TASKPROF needs to map the dynamic execution information about asymptotic parallelism and critical work to static spawn sites in the program. Hence, TASKPROF instruments the spawn sites to capture the line number and the file name of the spawn site. TASKPROF's static instrumentation is currently structured as a rewriter over the abstract syntax tree of the program using the Clang compiler front-end. We have added static instrumentation to the task parallel runtime to construct the DPST when tasks are created or when tasks finish execution. We have also added static instrumentation to the task parallel runtime library to read hardware performance counters at the beginning and the end of every step node. Our instrumented libraries and compiler instrumentation enable the programmer to use TASKPROF without making any changes to the source code.

3.1 Parallel Profile Execution

The goal of the profile execution is to collect fine-grained statistics of the program to enable a subsequent offline computation of asymptotic parallelism. Typically, programs are profiled with representative production inputs that have long execution times. Hence, a fast profile execution is desirable. Our goal is to ensure that the execution time of the program with and without profiling is similar. Hence, TASKPROF profiles in parallel leveraging multi-core processors. To ensure a parallel profile execution, it needs to construct the execution graph in parallel and collect statistics about the program in a thread-safe manner.

The DPST representation for parallel profile execution. We propose to use the DPST representation to capture the execution graph and to attribute work to its nodes because the DPST can be constructed in parallel. TASKPROF constructs the DPST as the program executes the injected static instrumentation and measures the work performed in each step node. The DPST, once constructed, allows TASKPROF to determine the dependencies between tasks. This fine-grained attribution of work to the step nodes in the DPST enables TASKPROF to compute the asymptotic parallelism in the program eventually using an offline analysis.

The complete DPST for a task parallel execution has a large number of nodes. Storing the entire DPST in memory during program execution can cause memory overheads and perturb the execution of the program. To address this issue, TASKPROF does not maintain the entire DPST in memory. In a library based task parallel programming model, a task is always attached to the same thread, which ensures that the task creation operations are inexpensive. We leverage this property to minimize the footprint of the DPST in memory. TASKPROF maintains a small fraction of the nodes that correspond to the tasks currently executing on each thread in memory.

Once a step node of a task completes execution, the work performed in the step node along with the information about its parent node is written to the profile data file. As async nodes do not perform any work, TASKPROF writes the information about its parent in the DPST and the spawn site associated with the async node to the profile data file. In contrast to step and async nodes, only parent node information is written to the profile data file for a finish node. To minimize the number of file operations, TASKPROF buffers the profile data in a fixed size buffer and writes them to the profile data file when the buffer is full. The memory allocated to the intermediate nodes that are no longer active is deallocated.

Measuring work with hardware performance counters. To measure the work performed in each step node without performance overhead, TASKPROF uses hardware performance counters. Performance counters are model specific registers available on almost all modern processors. They count various events performed by the hardware using precise event-based sampling mechanisms [8]. These performance counters can be programmatically accessed. TASKPROF can use both the number of dynamic instructions and the number of execution cycles to measure the work done in a step node. Measuring execution cycles allows TASKPROF to account for latencies due to secondary effects such as locality, sharing, and long latency instructions. Further, the operations on these counters are thread-safe. TASKPROF reads the value of the counter at the beginning and the end of the step node using static instrumentation injected into the program. It calculates the work performed in the step node by computing difference between the two counter values. This fine-grained measurement of work performed in each step node using hardware performance counters along with the construction of the DPST while executing in parallel allows TASKPROF to compute a precise, yet fast profile of the program.

The profile data file from the parallel profile execution contains the work done in each step node. It also contains the information about the parent for each node in the DPST and the spawn site information for each async node.

3.2 Offline Analysis of the Profile Data

TASKPROF's offline analysis reconstructs the DPST using the data from the profile execution and computes the work and critical work (span) for each spawn site in the program. The construction of the DPST from the profile data is fairly straightforward as it contains information about nodes, their parent nodes, and the left-to-right ordering of the nodes. In this section, we describe the computation of work and span for each intermediate node in the DPST given the work performed in the step nodes. We also describe the process of mapping this dynamic information to static spawn sites.

Computing work and critical work for each intermediate node. In the DPST representation, all computation is performed in the step nodes. The step nodes have fine-grained work information from the profile execution. TASKPROF needs to compute the total work and the fraction of that work done serially (critical work) for each intermediate node in the DPST. To provide meaningful feedback to the programmer, TASKPROF also computes the list of spawn sites that perform critical work and the portion of the critical work performed exclusively by each spawn site.

TASKPROF computes the total work and the critical work at each intermediate node by performing a bottom-up traversal of the DPST. The total work performed in the sub-tree at each intermediate node is sum of the work performed by all the step nodes in the sub-tree. In contrast, critical work measures the amount of work that is performed serially. Computing critical work and the set of tasks performing the critical work requires us to leverage the properties of the DPST. Specifically, we leverage the following properties of the DPST to compute the critical work.

- The siblings of a node in a DPST are ordered left-to-right reflecting the left-to-right sequencing in the parent task.
- Given an intermediate node, all the direct step children of the node execute serially.
- All the left step or finish siblings of an async node execute serially with the descendants of the async node.
- All the right siblings (and their descendants) of an async node execute in parallel with the descendants of the async node.

Using the above properties of the DPST, the critical work at an intermediate node will be equal to either (1) the serial work done by all the direct step children and the critical work performed by the finish children or (2) the critical work performed by descendants of an async child and the serial work performed by the left step and finish siblings of the specific async child in consideration. Since any intermediate node in the DPST can have multiple async children, TASKPROF needs to check if any of the async nodes can contribute to the critical work. For example, consider the intermediate node $F2$ in Figure 3(a) that has two async nodes $A1$ and $A2$. The critical work will be the maximum of (1) the work done by the direct step child $S4$ or (2) the critical work by the async child $A1$ (it does not have any left siblings), or (3) the sum of the critical work by the async child $A2$ and the work done by the step node $S4$, which is the left step sibling of $A2$.

Each async node in the DPST corresponds to a spawn site in the program because async nodes are created when a new task is spawned. Hence, TASKPROF computes the list of spawn sites performing critical work by computing the list of async nodes that

function COMPUTEWORKSPAN(T)
  for all intermediate node N in bottom-up traversal of T **do**
    $C_N \leftarrow \text{CHILDREN}(N)$
    $N.work \leftarrow \sum_{C \in C_N} C.work$
    $S_N \leftarrow \text{STEPCHILDREN}(N)$
    $F_N \leftarrow \text{FINISHCHILDREN}(N)$
    $N.c\_work \leftarrow \sum_{S \in S_N} S.work + \sum_{F \in F_N} F.c\_work$
    $N.e\_work \leftarrow \sum_{S \in S_N} S.work + \sum_{F \in F_N} F.e\_work$
    $N.ss\_list \leftarrow \bigcup_{F \in F_N} F.ss\_list$
    for all $A \in \text{ASYNCCHILDREN}(N)$ **do**
      $LS_A \leftarrow \text{LEFTSTEPSIBLINGS}(A)$
      $LF_A \leftarrow \text{LEFTFINISHSIBLINGS}(A)$
      $llw_A \leftarrow \sum_{LS \in LS_A} LS.work + \sum_{LF \in LF_A} LF.c\_work$
      if $llw_A + A.c\_work > N.c\_work$ **then**
        $N.c\_work \leftarrow llw_A + A.c\_work$
        $N.e\_work \leftarrow \sum_{S \in LS_A} S.work + \sum_{F \in LF_A} F.e\_work$
        $N.ss\_list \leftarrow \bigcup_{LF \in LF_A} LF.ss\_list \cup A.ss\_list$
      end if
    end for
    if N is a `async` node **then**
      $N.ss\_list \leftarrow N.ss\_list \cup \langle N.s\_site, N.e\_work \rangle$
    end if
  end for
end function

Figure 4: Algorithm to compute the total work (`work`), critical work (`c_work`), exclusive work (`e_work`), and the spawn sites that perform the critical work (`ss_list`) for each intermediate node in the DPST. The function CHILDREN returns the set of children of the input node. Similarly, functions STEPCHILDREN, FINISHCHILDREN and ASYNC-CHILDREN return the set of `step`, `finish` and `async` child nodes of the input node, respectively. The function LEFTSTEPSIBLINGS returns the set of `step` sibling nodes that occur to the left of the input node in the DPST. Similarly, the LEFTFINISHSIBLINGS returns the set of `finish` sibling nodes to the left of the input node in the DPST.

contribute to the critical work in the sub-tree of the intermediate node.

Algorithm to compute work and critical work. Figure 4 provides the algorithm used by TaskProf to compute the total work, the critical work, and the set of spawn sites contributing to the critical work. The algorithm maintains four quantities with each intermediate node in the DPST: (1) total work performed in the sub-tree under the node (`work`), (2) the critical work performed in the sub-tree (`c_work`), (3) the list of spawn sites that perform the critical work (`ss_list`), and (4) the part of the critical work that is performed exclusively by the direct children of the node (`e_work`). The exclusive work of a node is equal to sum total of the work performed by the direct step children and the exclusive work performed by the finish children. We consider the exclusive work performed by a finish node because it is not yet associated with any spawn site. The exclusive work of the current node will eventually be associated with a spawn site. The algorithm does not consider the exclusive work of the async children because it is already associated with a spawn site.

The algorithm traverses each node in the DPST in a bottom up fashion. All step nodes have work information from the profile data. For any intermediate node, the work performed under the sub-tree is the sum of the work performed by all its children (lines 3-4 in Figure 4). For a given intermediate node, TaskProf initially computes the serial work performed in all the step and finish children as the critical work (lines 5-7 in Figure 4). For each async child of the current node, it checks if the serial work done by the async node and its left siblings is greater than the critical work computed until that point (lines 10-15 in Figure 4).

To compute the set of spawn sites performing critical work, each intermediate node also maintains a list of spawn sites and the exclusive work performed by them. The algorithm initially sets the spawn site list for a node to be the union of spawn site lists of its finish children (lines 8-9 in Figure 4). Whenever an async child contributes to the critical work, the spawn site list of the current node is the union of the spawn site list of the async child and the spawn site lists of the finish children that are to the left of the async child (line 17 in Figure 4). When an async child contributes to the critical work, the exclusive work of the current node is equal to sum of the work performed by the left step siblings and the exclusive work performed by the left finish siblings of the async child (line 16 in Figure 4). The algorithm adds the spawn site and the exclusive work performed by the current node to the spawn site list of the current node when it is an async node (lines 20-22 in Figure 4).

After the algorithm completes traversing the entire DPST, the root of the DPST will contain the list of all spawn sites that perform critical work and their individual contribution to the critical work. The root node also contains information about the total work performed by the program, the work that is computed serially by the program, and the exclusive work performed under the entry function of the program (*i.e.*, `main`).

Aggregating information about a spawn site. A single spawn site may be executed multiple times in a dynamic execution. Hence, TaskProf aggregates information from multiple invocations of the same spawn site. TaskProf computes the aggregate information for each spawn site by performing another bottom-up traversal of the DPST at the end. When it encounters an async node, TaskProf uses a hash table indexed by the spawn site associated with the async node and adds the total work and critical work to the entry. When aggregating this information, TaskProf has to ensure that it does not double count work and critical work when recursive calls are executed. In the presence of recursive calls, a descendant of an async node will have the same spawn site information as the async node. If we naively add descendant's work, it leads to double counting as the work and critical work of the current async node already has the work/critical work of the descendant async node. Hence, when TaskProf encounters an async node in a bottom-up traversal of the DPST, it checks whether the descendants of the async node have the same spawn site information. When a descendant with the same spawn site exists, it subtracts such a descendant's work and critical work from the entry in the hash table corresponding to the spawn site. Subsequently, TaskProf adds the work and critical work of the current async node to the hash table.

Profile reported to the user. For each spawn site in the program, TaskProf presents the work, the critical work, asymptotic parallelism, and the percentage of critical work exclusively done by the spawn site. The asymptotic parallelism of a spawn site is the ratio of the total work and the critical work performed by a spawn site. The spawn sites are ordered by the percentage of critical work exclusively performed by the spawn site. Figure 3(b) illustrates the parallelism profile for the program in Figure 2 that has the DPST shown in Figure 3(a). If a spawn site has low parallelism and performs a significant proportion of the critical work, then optimizing the task spawned by the spawn site may increase the parallelism in the program. This profile information provides a succinct description of the parallelism bottlenecks in the program.

4 CAUSAL PROFILING

TaskProf reports the set of spawn sites performing critical work to the user, which highlights a set of parallelism bottlenecks in the program. A programmer can consider these spawn sites to be initial candidates for optimization to reduce serial computation.

Reducing critical work and the impact on parallelism. Designing a new optimization or a parallelization strategy that reduces the critical work typically requires effort and time. A program may have multiple spawn sites that perform similar amount of serial work. When a set of spawn sites are parallelized to reduce serial work, the resultant execution may have a new set of spawn sites whose critical work is similar to the critical work before the optimization. In such cases, an optimization to a spawn site performing critical work will not improve the asymptotic parallelism in the program. Hence, programmers would benefit from a causal profile of program that identifies the improvement in asymptotic parallelism when certain regions of the code are optimized.

Causal profile with TaskProf. A causal profile provides information on improvements in parallelism when certain parts of the code are parallelized. TaskProf proposes a technique to generate causal profiles for task parallel programs. The programmer can get an accurate estimate of the improvement in asymptotic parallelism by reducing the serial work in a region of the program using TaskProf's causal profile. TaskProf provides such an estimate even before the programmer has designed a concrete strategy to parallelize or reduce the serial work in the region of code under consideration. In summary, a causal profile enables the programmer to identify parts of the program that really matter in increasing the asymptotic parallelism. Figure 3(c) provides the causal profile for the program in Figure 2 where the regions under consideration are demarcated by `__CAUSAL_BEGIN__` and `__CAUSAL_END__`. Next, we describe how TASKPROF generates a causal profile leveraging the DPST and the fine-grained attribution of computation.

Static code annotations. To generate causal profiles, the programmer annotates a static region of code that is considered for parallelization and the expected improvement to the critical work from parallelization. The programmer can provide multiple regions as candidates for optimization. TASKPROF generates a causal profile that accounts for all annotated regions by default. The programmer can use TASKPROF to generate a causal profile for each region in isolation when multiple code regions are annotated. Figure 2 illustrates the regions of code annotated for causal profiling with `__CAUSAL_BEGIN__` and `__CAUSAL_END__` annotations. If the programmer does not specify the amount of expected improvement for the considered region, TASKPROF assumes a default value. If the annotations are nested, the outermost region of code is considered for estimating the benefits.

Profile execution and attribution of work. TASKPROF uses these annotations, profiles the program, constructs the DPST to attribute work to various regions, and provides the estimated improvement in asymptotic parallelism from optimizing the annotated regions. During profile execution, TASKPROF measures work performed in annotated part of the step node and also parts of the step node that have not been annotated. Hence, each step node can have multiple work measurements corresponding to static regions with and without annotation. TASKPROF accomplishes it by reading the performance counter value at the beginning and the end of the each dynamic region. TASKPROF maintains a list of work values for each step node and writes it to the profile data file.

Algorithm to generate causal profiles. The algorithm to compute the causal profile is similar to the work and span algorithm in Figure 4. It takes the DPST as input and a list of anticipated improvements for the annotated regions. The algorithm outputs a causal profile that computes the improvement in asymptotic parallelism of the whole program for the specified improvements for the annotated regions. The causal profile algorithm performs a bottom up traversal of the DPST similar to the work and span algorithm in Figure 4. However, the causal profiling algorithm does not track spawn sites and computes the whole program's work and critical work. The key difference with the causal profiling algorithm is the manner in which it handles the work done by the step nodes, which have regions corresponding to user annotations. Specifically, TASKPROF maintains a list of annotated and non-annotated regions executed with each step node and the amount of work performed in each region. To estimate the effect of optimizing/parallelizing the annotated region, we reduce the critical work contribution of the annotated region by the user-specified optimization factor while keeping the total work performed by the regions unchanged. The output of the causal profiling algorithm is a list that provides the asymptotic parallelism (total work/total span) for each anticipated improvement factor for the regions under consideration.

Illustration. After analyzing the parallelism profile in Figure 3(b) for the program in Figure 2, the programmer has identified two regions of code (lines 4-6 and lines 21-23 in Figure 2) for optimization. The regions are annotated with `__CAUSAL_BEGIN__` and `__CAUSAL_END__` annotations to demarcate the beginning and the end. During execution, the region at lines 21-23 is executed once and is represented by step node $S0$ in Figure 3(a). In contrast, the region at lines 4-6 is executed twice and is represented by step nodes $S5$ and $S6$ in Figure 3(a). In this example, the entire step node corresponds to the annotated region. In general, a step node may have multiple annotated and non-annotated regions within a single step node. To generate a causal profile, the critical work performed by nodes $S0$, $S5$, and $S6$ are decreased by 2×, 4×, and 8× and its impact on whole program parallelism is computed. Figure 3(c) provides the causal profile with the annotated regions, which reports that the asymptotic parallelism in the program increases when those two regions are optimized.

5 EXPERIMENTAL EVALUATION

This section describes our prototype, our experimental setup, and an experimental evaluation to answer the following questions: (1) Is TASKPROF effective in identifying parallelism bottlenecks? (2) How does TASKPROF's parallel profile execution compare to serial profilers? (3) Does TASKPROF introduce minimum amount of perturbation? (4) Is TASKPROF usable by programmers?

Prototype. We have built a TASKPROF prototype to profile task parallel programs using the Intel Threading Building Blocks(TBB) library [41]. The prototype provides a TBB library that has been modified to construct the DPST, measure work done in step nodes using hardware performance counters, and track file name and line information at each spawn site. The prototype also handles algorithms for geometric decomposition such as `parallel_for` and `parallel_reduce`. The prototype also includes a Clang compiler pass that automatically adds line number and file name information to the TBB library calls, which enables the programmer to use the modified library without making any source code changes. Hence, the modified TBB library can be linked to any TBB program. Our prototype adds approximately 2000 lines of code to the Intel TBB library to perform various profiling operations. Our tool is open source.

Applications used for evaluation. We evaluated TASKPROF using a collection of twenty three TBB applications, which include fifteen applications from the problem based benchmark suite (PBBS) [43], all five TBB applications from the Parsec suite [4], and three TBB applications from the structured parallel programming book [34]. The PBBS applications are designed to compare different parallel programming methodologies in terms of performance and code. We conducted all experiments on a 2.1GHz 16-core Intel x86-64 Xeon server with 64 GB of memory running 64-bit Ubuntu 14.04.3. We measured execution time by running each application five times and use the mean of the five executions to report performance. We use the `perf` module in Linux to programmatically access hardware performance counters.

Application	Description	Speedup	Parallel-ism	# of regions	Causal parallelism
blackscholes	Stock option pricing	1.09	1.14	2	59.24
bodytrack	Tracking of a human body	5.96	22.19	1	40.32
fluidanimate	Simulate fluid dynamics	9.39	66.09	1	90.2
streamcluster	Clustering algorithm	7.3	55.13	2	198.93
swaptions	Price a portfolio	8.59	73.45	1	98.73
convexHull	Convex hull	1.3	1.28	4	112.17
delRefine	Delaunay Refinement	2.93	5.5	7	61.28
delTriang	Delaunay triangulation	1.23	1.47	5	78.85
karatsuba	Karatsuba multiplication	5.22	23.69	1	36.9
kmeans	K-means clustering	2.54	4.18	6	69.6
nearestNeigh	K-nearest neighbors	4.54	12.41	2	30.55
rayCast	Triangle intersection	6.62	48.49	2	68.52
sort	Parallel quicksort	3.91	6.33	2	45.04
compSort	Generic sort	4.99	38.97	4	86.23
intSort	Sort key-value pairs	4.71	48.68	2	75.02
removeDup	Remove duplicate value	6.04	54.91	3	98.24
dictionary	Batch dictionary opers	5.13	38.1	4	73.12
suffixArray	Sequence of suffixes	3.75	5.5	1	28.53
bFirstSearch	Breadth first search	6.6	22.45	5	60.55
maxIndSet	Maximal Independent Set	5.48	16.46	5	52.23
maxMatching	Maximal matching	6.73	46.04	0	46.04
minSpanForest	Minimum spanning forest	3.47	7.99	2	49.78
spanForest	Spanning tree or forest	7.46	44.04	1	58.91

Table 1: Applications used to evaluate TaskProf. We provide a short description of the application, the speedup obtained on a 16-core machine when compared to serial execution time, the asymptotic parallelism reported by TaskProf, the number of annotated regions in the program that provides maximum parallelism with causal profiling, and the asymptotic parallelism when the critical work in the annotated regions is optimized by 100×, which we list as causal parallelism.

I. Minimum spanning forest

	File:Line	Parallel -ism	Critical path percent
★	MSTTime.C:77	**7.99**	85.97
	graphIO.h:167	42.11	3.68
	spec_for.h:82	51.79	2.64
	IO.h:71	58.69	1.52

(a) Original parallelism profile

Optimization factor	Parallel -ism
50X	47.28
100X	49.78
200X	51.14
400X	51.84

(b) Causal profile

	File:Line	Parallel -ism	Critical path percent
★	MSTTime.C:77	**33.34**	33.73
	graphIO.h:167	42.87	14.58
	spec_for.h:82	51.54	9.45
	sampleSort.h:81	52.28	8.48

(c) Parallelism profile after optimization

II. Convex hull

	File:Line	Parallel -ism	Critical path percent
★	hullTime.C:55	**1.28**	80.66
	hull.C:209	1.33	10.3
	hullTime.C:46	2.47	5.75
	hull.C:117	1.2	2.67

(a) Original parallelism profile

Optimization factor	Parallel -ism
50X	60.3
100X	112.17
200X	196.82
400X	316.06

(b) Causal profile

	File:Line	Parallel -ism	Critical path percent
	geoIO.h:96	52.71	25.82
★	hullTime.C:55	**50.84**	18.93
	sequence.h:359	41.97	17.06
	IO.h:71	55.46	16.15

(c) Parallelism profile after optimization

III. Delaunay triangulation

	File:Line	Parallel -ism	Critical path percent
★	delTime.C:55	**1.47**	99.12
	geoIO.h:96	58.28	0.09
	delaunay.C:385	56.51	0.07
	IO.h:164	71.41	0.07

(a) Original parallelism profile

Optimization factor	Parallel -ism
50X	63.55
100X	78.45
200X	96.4
400X	111.33

(b) Causal profile

	File:Line	Parallel -ism	Critical path percent
	delaunay.C:284	57.93	47.46
	sequence.h:365	58.65	5.99
	IO.h:179	51.78	5.42
★	delTime.C:55	**53.11**	3.93

(c) Parallelism profile after optimization

IV. Delaunay refinement

	File:Line	Parallel -ism	Critical path percent
★	refineTime.C:59	**5.5**	91.47
	refine.C:249	55.3	4.84
	refine.C:260	54.25	1.73
	topFromT.C:126	49.82	0.24

(a) Original parallelism profile

Optimization factor	Parallel -ism
50X	55.68
100X	61.27
200X	64.51
400X	66.26

(b) Causal profile

	File:Line	Parallel -ism	Critical path percent
	refine.C:246	56.83	25.22
★	refineTime.C:59	**48.08**	16.5
	refine.C:257	58.42	14.37
	sequence.h:365	62.48	6.64

(c) Parallelism profile after optimization

V. Blackscholes

	File:Line	Parallel -ism	Critical path percent
★	bscholes.c:323	**1.14**	99.72
	bscholes.c:274	51.2	0.28

(a) Original parallelism profile

Optimization factor	Parallel -ism
50X	39.1
100X	59.24
200X	79.78
400X	96.52

(b) Causal profile

	File:Line	Parallel -ism	Critical path percent
	bscholes.c:145	1	46.52
	bscholes.c:212	1	22.75
★	bscholes.c:464	**40.03**	21.71
	bscholes.c:415	48.44	9

(c) Parallelism profile after optimization

Figure 5: Figure reports the original parallelism profile, the causal profile for the annotated regions, and final parallelism profile generated by TaskProf after annotated regions were parallelized for each of the five applications. We list the top four spawn sites from TaskProf's parallelism profile. Line with a "⋆" in the profile corresponds to the main function and reports the parallelism for the entire program. The asymptotic parallelism for the entire program is marked bold in the parallelism profile.

RQ1: Is TaskProf effective in identifying parallelism bottlenecks? We used TaskProf to identify parallelism bottlenecks in all the 23 applications. Table 1 provides details on applications used, the speedup of the applications on a 16-core machine compared to serial execution, the asymptotic parallelism reported by TaskProf, the number of regions that we identified using TaskProf to increase asymptotic parallelism, and the resultant asymptotic parallelism from causal profiling when the critical work in the identified regions is decreased by 100×. A rule of wisdom when developing these task parallel applications is that the asymptotic parallelism should be at least 10× or more than the expected speedup on a machine to account for scheduling overheads [17, 41, 42].

TaskProf's profile shows that some applications in Table 1 have reasonable asymptotic parallelism, which accounts for a reasonable speedup on a 16-core machine. For example, `fluidanimate` application has an asymptotic parallelism of 66.09 which is the maximum available speedup when the program is executed on a large number of machines. The `fluidanimate` application exhibits a speedup of 9.39× compared to a serial execution when the program was executed on a 16-core machine.

Table 1 also shows that we were able to identify a small number of code regions which when optimized provides a significant increase in asymptotic parallelism. TaskProf's profile information on spawn sites performing critical work and the causal profiling strategy was instrumental in identifying the specific regions of code as candidates for increasing asymptotic parallelism. The application `maxMatching` already had a large amount of asymptotic parallelism and we could not find any region that increases parallelism. Developers of the PBBS applications were not aware of these specific parallelism bottlenecks even though these applications have been widely used, which emphasizes the need for TaskProf.

In summary, TaskProf enabled us to identify a set of code regions that can increase asymptotic parallelism significantly in almost all our applications. Once we identified code regions that can increase asymptotic parallelism, we designed concrete parallelization strategies to reduce the critical work for 5 applications, which increased the asymptotic parallelism and the speedup of the program. We describe them below.

Improving the speedup of the MinSpanningForest application. This PBBS application computes the minimum spanning forest of the input undirected graph. The program has a speedup of 3.47× over serial execution on a 16-core machine. The parallelism profile generated by TaskProf is shown in Figure 5(I)(a), which reports that the parallelism in the program (main function at `MSTTime.C:77`) is 7.99. The main function also performs 85% of the serial work in the program. We identified two regions of code (by trial and error) using annotations for causal profiling in the main function. Figure 5(I)(b) presents the causal profile generated by TaskProf for optimizing these two regions, which increases the asymptotic parallelism in the program. On further investigation of the code regions, we realized that annotated regions were performing a serial sort. We replaced them with a parallel sort function, which increased the asymptotic parallelism to 33.34 from 7.99. Figure 5(I)(c) reports the profile after our parallel sort optimization. The speedup of the program increased from 3.49× to 6.37×.

Improving the speedup of the Convex Hull application. This PBBS application computes the convex hull of a set of points using a divide and conquer approach [3]. TaskProf's profile shown in Figure 5(II)(a) reveals that the program has an asymptotic parallelism of 1.28 for the whole program. As expected, it did not exhibit any speedup. Figure 5(II)(a) shows that 80% of the critical work is performed by the spawn site at `hullTime.C:55`. We annotated two regions corresponding to that spawn site, which performed sequential read and write operations of the input and output files respectively. TaskProf's causal profile showed that it would increase the parallelism to 6.85. Subsequently, we annotated two additional regions of code corresponding to the spawn site performing the next highest critical work (`hull.C:209`) in Figure 5(II)(a). The causal profile shown in Figure 5(II)(b) shows that asymptotic parallelism increases significantly when all the four regions are optimized. We parallelized a loop at spawn site `hull.C:209` using `parallel_for` and parallelized I/O at spawn site `hullTime.C:55`. These optimizations increased the parallelism to 50.84 (see Figure 5(II)(c)) and the speedup of the whole program increased from 1.3× to 8.14×.

Improving the speedup of Delaunay Triangulation. This PBBS application produces a triangulation given a set of points such that no point lies in the circumcircle of the triangle. The program has an asymptotic parallelism of 1.47 (see Figure 5(III)(a)) for the entire program and exhibits little speedup. The spawn site at `delTime.C:55` performs 99% of the critical work. When we looked at the source code, we found that the program is structured as a collection of `parallel_for` constructs interspersed by serial code. We annotated five regions of code between the invocations of `parallel_for`. The causal profile in Figure 5(III)(b)) shows that the asymptotic parallelism increases significantly by optimizing the annotated regions. We parallelized the annotated regions, which had serial for loops, using `parallel_for` while ensuring they operate on independent data. The profile for the resultant program is shown in Figure 5(III)(c). The parallelism increased to 53.11 and the speedup increased from 1.23× to 5.82×.

Improving the speedup of Delaunay Refinement. This PBBS application takes a set of triangles that form a delaunay triangulation and produces a new triangulation such that no triangle has an angle lesser than a threshold value. TaskProf's profile for this program reports an asymptotic parallelism of 5.5 (see Figure 5(IV)(a)) and it had a speedup of 2.93×. Similar to delaunay triangulation, this program also had a set of serial code fragments in-between `parallel_for` calls. We identified seven regions of such serial

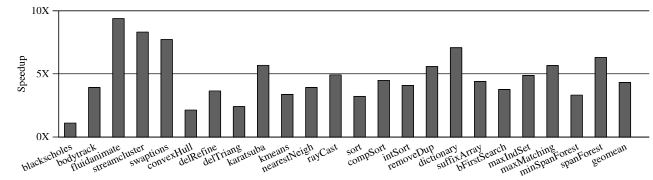


Figure 6: Speedup with TaskProf's parallel profile execution over serial profile execution.

code and annotated them. TaskProf's causal profile shown in Figure 5(IV)(b) indicates that optimizing all these seven regions can increase asymptotic parallelism. We parallelized the serial for loops in these seven regions using `parallel_for`, which increased the asymptotic parallelism to 48.08 (see Figure 5(IV)(c)) and the speedup increased from 2.93× to 6.42×.

Improving the speedup of Blackscholes. This application from the PARSEC suite [4] computes the price of a portfolio of options using partial differential equations. It has low asymptotic parallelism for the entire program (see Figure 5(V)(a)). This program has a single `parallel_for` that has reasonable parallelism of 51.2. However, the spawn site at `bscholes.c:323` is performing 99% of the program critical work. Our examination of the code revealed that it was reading and writing serially. We split the input and output into multiple files and parallelized the input/output operations which increased the parallelism to 40.03 and the speedup increased from 1.09× to 7.7×.

In summary, TaskProf enabled us to quantify asymptotic parallelism in the program and its causal profiling strategy enabled use to identify specific regions of code that can increase parallelism.

RQ2: How does TaskProf's parallel profile execution compare to serial profilers? TaskProf's profile execution executes in parallel compared to prior profilers [42], which execute serially. To quantify the benefits of parallel profile execution, we designed a serial version of TaskProf by pinning the execution of the program to a single core. This is an approximation of serial profiling as TBB programs do not have serial semantics. Figure 6 reports the speedup of a parallel TaskProf profile execution compared to a serial profile execution. On average, TaskProf's parallel profile execution is 4.32× faster than serial profile execution. The speedup from a parallel profile execution is proportional to the amount of parallelism in the application.

RQ3: Is TaskProf's execution perturbation free? TaskProf uses hardware performance counters to perform fine-grain attribution of work and to minimize perturbation. The average performance overhead of TaskProf's profile execution compared to the parallel execution of the program without any profiling instrumentation is 56%. A major fraction of this performance overhead is attributed to system calls to read hardware performance counters. TaskProf's profile execution is an order of magnitude faster than instrumenting each dynamic instruction through compiler instrumentation, which exhibited overheads of 20×-100× for the applications in Table 1. Hence, TaskProf minimizes perturbation even with fine-grained attribution of work.

RQ4: Is TASKPROF usable by programmers? We conducted a user study to evaluate the usability of TASKPROF. The user study had thirteen participants: twelve graduate students and one senior undergraduate student. Among them, two students had 4+ years of experience in parallel programming, five students had some prior experience, four students had passing knowledge, and two students had no prior experience with parallel programming. The total duration of the user study was four hours. To ensure that every student had some knowledge in parallel programming, we provided a 2-hour tutorial on task parallelism, and on writing and debugging task parallel programs using Intel TBB. We gave multiple examples to demonstrate parallelism bottlenecks.

After the tutorial, the participants were given a total of four applications and were asked to identify parallelism bottlenecks without using TASKPROF in a one hour time period. Three applications — minSpanForest, convexHull, and blackscholes — from Table 1 and a treesum application similar to the example in Figure 2. We chose these applications as it had varying levels of difficulty in diagnosing parallelism bottlenecks. We asked the participants to identify the static region of code causing the bottleneck and record the time they spent to analyze each program. They were not required to design any optimization. Some participants used `gprof` and others used fine-grained wall clock based timing for assistance. At the end of the time period, twelve of them did not correctly identify parallelism bottlenecks in any of the four applications. One participant, who had 4+ years of experience in parallel programming, identified the bottleneck in one (minSpanForest) out of the four applications. Some participants were misled by the `gprof` profile.

Subsequently after the first part, we gave a brief tutorial of TASKPROF on a simple example program. The participants were then asked to identify bottlenecks in the four applications using TASKPROF within an hour. Using TASKPROF, seven participants found the parallelism bottleneck in all the four applications, one participant found the bottleneck in three of them, four participant found the bottleneck in two of them, and one participant did not find the bottleneck in any application. Among the participants who identified at least one bottleneck for any application, it took them 12 minutes on average per application to identify the bottleneck using TASKPROF. The participants indicated that once they became familiar with the tool by identifying a bottleneck in one application, subsequent tasks were repetitive. In summary, our user study suggests that TASKPROF can enable both expert and relatively inexperienced programmers identify parallelism bottlenecks quickly.

6 RELATED WORK

There is a large body of work to identify bottlenecks in multithreaded and task parallel programs. These include techniques to address load imbalances [12, 24, 38, 44], scalability bottlenecks [32, 42, 45], visualizing bottlenecks [13–15, 25], synchronization bottlenecks [7, 11, 47], redundant traversals [36, 37], and data locality bottlenecks [1, 28–31]. Data locality and synchronization bottlenecks increase serial work. Hence, TASKPROF will report asymptotic parallelism in their presence. In contrast to prior proposals, TASKPROF is the first proposal that estimates the improvement in parallelism with causal profiling. Next, we focus on the closest related work.

Profiling tools for task parallel programs. Profiling tools such as HPCToolkit [2], and Intel VTune Amplifier [9] can analyze a program's performance on various parameters using hardware performance counters. HPCToolKit also has metrics to quantify idleness and the scheduling overhead [45] in Cilk programs that is specific to a machine. They do not compute the asymptotic parallelism in the program. They also do not identify code that matters with respect to asymptotic parallelism. CilkView [21] computes the whole program asymptotic parallelism. CilkProf [42] computes asymptotic parallelism per spawn site using an online algorithm. However, these profilers execute the program serially, which is only possible with Cilk programs with C-elision. Many task parallelism frameworks including Intel TBB do not have serial semantics, which makes these profilers unsuitable. Further, executing the profiler serially can cause high overheads. Unlike TASKPROF, they also cannot estimate the benefits of optimizing specific regions of code.

Causal Profiling tools. An early causal profiling technique proposed Slack [22], which is a metric that estimates the improvement in execution time through critical path optimizations for a specific machine model. Kremlin [18] identifies regions of code that can be parallelized in serial programs by tracking loops and identifying dependencies between iterations. Kismet [23] builds on Kremlin to estimate speedups for the specific machine on which the serial program is executed. These techniques are tied to a specific machine and cannot estimate asymptotic parallelism improvements.

Our work is inspired by COZ [10], a causal profiler for multithreaded programs that estimates the increase in speedup from optimizing a line or a region of code. COZ slows down all other threads when a region of interest is executed, which has the same effect as speeding up a region of interest. It samples the execution and performs multiple performance experiments to estimate the speedup. In a task parallel context, it is not possible to slow down all active tasks. Further, slowing down threads does not measure the impact of the region as work stealing dynamically balances the load. Further, COZ's speedup estimates are specific to a particular machine. TASKPROF, though similar in spirit, addresses the above challenges and proposes a causal profiler that leverages the dynamic execution structure and estimates improvements in asymptotic parallelism. Hence, TASKPROF's profile is not specific to a single machine and enables the development of performance portable code.

7 CONCLUSION

We have proposed TASKPROF to identify parallelism bottlenecks in task parallel programs that manifest when the program is executed on a large number of processors. TASKPROF identifies these asymptotic parallelism bottlenecks by performing a low-overhead, yet fine-grained attribution of work and by leveraging the dynamic execution structure of a task parallel execution. TASKPROF reports asymptotic parallelism and serial work performed at each spawn site. TASKPROF's causal profile estimates the improvements in parallelism when programmer annotated regions are optimized. We have identified bottlenecks and improved the speedup in numerous Intel TBB applications . Our user study show that TASKPROF helps developers identify parallelism bottlenecks quickly.

REFERENCES

[1] 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.

[2] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent. 2010. HPCTOOLKIT: Tools for Performance Analysis of Optimized Parallel Programs Http://Hpctoolkit.Org. *Concurr. Comput. : Pract. Exper.* (2010), 685–701.

[3] C. Bradford Barber, David P. Dobkin, and Hannu Huhdanpaa. 1996. The Quickhull Algorithm for Convex Hulls. *ACM Trans. Math. Softw.* 22, 4 (Dec. 1996), 469–483.

[4] 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.

[5] Vincent Cavé, Jisheng Zhao, Jun Shirako, and Vivek Sarkar. 2011. Habanero-Java: The New Adventures of Old X10. In *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java (PPPJ)*. 51–61.

[6] 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.

[7] Guancheng Chen and Per Stenstrom. 2012. Critical Lock Analysis: Diagnosing Critical Section Bottlenecks in Multithreaded Applications. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*. 71:1–71:11.

[8] Intel Corporation. 2016. *Intel(R) 64 and IA-32 Architectures Software Developer's Manual, Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C and 3D*.

[9] Intel Corporation. 2017. Intel VTune Amplifier 2017. https://software.intel.com/en-us/intel-vtune-amplifier-xe. (2017). Retrieved 2017-02-23.

[10] 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.

[11] Florian David, Gael Thomas, Julia Lawall, and Gilles Muller. 2014. Continuously Measuring Critical Section Pressure with the Free-lunch Profiler. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA)*. 291–307.

[12] 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.

[13] 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.

[14] 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 & Applications (OOPSLA)*. 355–372.

[15] 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.

[16] Mingdong Feng and Charles E. Leiserson. 1997. Efficient Detection of Determinacy Races in Cilk Programs. In *Proceedings of the 9th ACM Symposium on Parallel Algorithms and Architectures (SPAA)*. 1–11.

[17] 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.

[18] 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.

[19] Susan L. Graham, Peter B. Kessler, and Marshall K. Mckusick. 1982. Gprof: A Call Graph Execution Profiler. In *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction (SIGPLAN)*. 120–126.

[20] Dan Grossman and Ruth E. Anderson. 2012. Introducing Parallelism and Concurrency in the Data Structures Course. In *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education*.

[21] 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.

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

[23] 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.

[24] Melanie Kambadur, Kui Tang, and Martha A. Kim. 2014. ParaShares: Finding the Important Basic Blocks in Multithreaded Programs. In *Proceedings of Euro-Par 2014 Parallel Processing: 20th International Conference (Euro-Par)*. 75–86.

[25] Andreas Knüpfer, Holger Brunst, Jens Doleschal, Matthias Jurenz, Matthias Lieber, Holger Mickler, Matthias S Müller, and Wolfgang E Nagel. 2008. The vampir performance analysis tool-set. In *Tools for High Performance Computing*. 139–155.

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

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] Timothy Mattson, Beverly Sanders, and Berna Massingill. 2004. *Patterns for Parallel Programming* (first ed.). Addison-Wesley Professional.

[34] Michael McCool, Arch Robison, and James Reinders. 2012. *Structured Parallel Programming: Patterns for Efficient Computation*. Morgan Kaufmann.

[35] 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 Trans. Parallel Distrib. Syst.* 1, 2 (April 1990), 206–217.

[36] Adrian Nistor, Po-Chun Chang, Cosmin Radoi, and Shan Lu. 2015. Caramel: Detecting and Fixing Performance Problems That Have Non-intrusive Fixes. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1 (ICSE)*. 902–912.

[37] Adrian Nistor, Linhai Song, Darko Marinov, and Shan Lu. 2013. Toddler: Detecting Performance Problems via Similar Memory-access Patterns. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE)*. 562–571.

[38] 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.

[39] 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.

[40] 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.

[41] James Reinders. 2007. *Intel Threading Building Blocks*. O'Reilly & Associates, Inc.

[42] 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.

[43] 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.

[44] 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.

[45] 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.

[46] Nathan R. Tallent, John M. Mellor-Crummey, and Allan Porterfield. 2010. Analyzing Lock Contention in Multithreaded Applications. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*. 269–280.
[47] Tingting Yu and Michael Pradel. 2016. SyncProf: Detecting, Localizing, and Optimizing Synchronization Bottlenecks. In *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA)*. 389–400.