
# A Parallelism Profiler with What-If Analyses for OpenMP Programs

Nader Boushehrinejadmoradi
*Department of Computer Science*
*Rutgers University*
Piscataway, USA
naderb@cs.rutgers.edu

Adarsh Yoga
*Department of Computer Science*
*Rutgers University*
Piscataway, USA
adarsh.yoga@cs.rutgers.edu

Santosh Nagarakatte
*Department of Computer Science*
*Rutgers University*
Piscataway, USA
santosh.nagarakatte@cs.rutgers.edu

***Abstract*—This paper proposes OMP-WHIP, a profiler that measures inherent parallelism in the program for a given input and provides what-if analyses to estimate improvements in parallelism. We propose a novel OpenMP series-parallel graph representation (OSPG) that precisely captures series-parallel relations induced by various directives between different fragments of dynamic execution. OMP-WHIP constructs the OSPG and measures the computation performed by each dynamic fragment using hardware performance counters. This series-parallel representation along with measurement of computation is a performance model of the program for a given input, which enables computation of inherent parallelism. This novel performance model also enables what-if analyses where a programmer can estimate improvements in parallelism when bottlenecks are addressed. We have used OMP-WHIP to identify parallelism bottlenecks in more than forty applications and then designed strategies to improve the speedup in seven applications.**

***Index Terms*—Parallel programming, Performance analysis**

## I. Introduction

OpenMP [1], [2] is an application programming interface to incrementally add parallelism to an application. OpenMP consists of a set of compiler directives and a runtime that orchestrates parallel execution. A programmer specifies code regions that can be executed in parallel using directives. The compiler translates these directives into outlined functions, which are executed by the runtime in parallel. The OpenMP specification includes both work-sharing and tasking directives [1], [2].

Although the incremental addition of parallelism with OpenMP enables easier adoption, the program can have serialization bottlenecks. A program that provides reasonable speedup with a small number of cores may not have scalable speedup with a large number of cores. To address this problem, a large number of OpenMP performance tools have been proposed [3]–[14]. Some techniques have been integrated into commercial tools (*e.g.*, Intel Vtune [5], Oracle's Developer studio [15]). These tools primarily indicate where the program spends its time [3] or contends for a resource [16]. Addressing regions where the program spends significant time may not improve the speedup because the bottleneck may shift to some other code region. Further, these tools provide a thread-based view of the program execution, which is dependent on the number of threads and their scheduling. It may not reveal serialization bottlenecks when the program is executed on a machine with a large number of cores.

This paper makes a case for measuring inherent parallelism in the OpenMP program for a given input to identify serialization bottlenecks and to estimate improvements when such bottlenecks are removed. Parallelism represents the speedup that the program will exhibit when it is executed on an infinite number of processors (assuming no scheduling overheads and secondary effects). Inherent parallelism is an upper bound on the speedup for the given input on any machine and is constrained by the longest serial computation in the program (*i.e.*, Amdahl's law).

This paper proposes OMP-WHIP, a profiler for OpenMP programs that measures the inherent parallelism in the entire program and for each OpenMP directive. The inherent parallelism reported by OMP-WHIP is not dependent on the number of threads and the specific scheduling of threads during execution. OMP-WHIP's what-if analyses enable the user to estimate the increase in parallelism even before concrete strategies to address serial bottlenecks have been designed. OMP-WHIP handles both work-sharing and tasking constructs. To compute parallelism, OMP-WHIP has to measure work performed by various fragments of dynamic execution without directives and identify those fragments that execute in series.

We propose OpenMP Series Parallel Graph (OSPG), a novel directed acyclic graph representation that precisely captures series-parallel relations between fragments of dynamic execution without OpenMP directives. Each leaf in the OSPG represents a fragment of execution without OpenMP directives. Intermediate nodes in the OSPG capture series-parallel relations induced by the OpenMP directives. The series-parallel relationship between existing nodes in the OSPG does not change when new nodes are added. OSPG is inspired by previous series-parallel graphs for task parallel programs [17]–[20]. In contrast to them, OSPG accurately models the semantics of OpenMP's directives and it can be constructed in parallel. Section III-B presents our novel algorithm to construct the OSPG during program execution. We can determine whether two leaf nodes execute in parallel by looking at the least common ancestor of the two nodes in the OSPG (see Section III). OMP-WHIP uses the OSPG to compute inherent parallelism and perform what-if analyses.

OMP-WHIP constructs this OSPG representation by intercepting the program during the execution of the OpenMP directive using OMPT callbacks [21]. Apart from constructing the OSPG, it also measures the total computation performed by

SC18, November 11-16, 2018, Dallas, Texas, USA
978-1-5386-8384-2/18/$31.00 

each dynamic execution fragment without OpenMP directives, using hardware performance counters. The OSPG along with this fine-grained measurement of computation in its leaves is an accurate performance model of the program for a given input, which can be used to compute the parallelism in the program and perform what-if analyses. OMP-WHIP maintains a small slice of the OSPG in memory at any point in time during profile execution. The profile reported to the user contains the inherent parallelism and the serial work exclusively performed by each OpenMP directive (see Section IV).

OpenMP directives with low parallelism are potential candidates to improve performance. However, optimizing one such region may not increase the parallelism when the program has multiple parallel paths that perform similar amounts of work. OMP-WHIP's what-if analyses estimate improvements in parallelism when certain regions of the code are hypothetically optimized. Programmers will annotate static program regions that they want to optimize. OMP-WHIP executes the program, identifies the dynamic regions that correspond to the annotated regions, and reconstructs the performance model by executing the program. OMP-WHIP's what-if analyses recompute the parallelism by reducing the serial work for the annotated regions by a user-defined threshold while keeping the total work exactly as measured. Reducing serial work while keeping the total work unchanged mimics the effect of parallelizing a region of code.

The OMP-WHIP prototype supports the common core of the OpenMP specification, tasking directives, and task dependencies. We have used OMP-WHIP with a set of 43 applications from Sequoia, Coral, BOTS, PBBS, NAS and Kastors benchmark suites. We were able to use OMP-WHIP's what-if analyses to identify bottlenecks in all of them. We improved the speedup of 7 applications by addressing the identified bottlenecks. For instance, OMP-WHIP's what-if analyses helped us identify two serial regions in AMGmk that would increase parallelism. Subsequently, parallelizing them improved the speedup of AMGmk from 5.39× to 9.13× on a 16-core machine. OMP-WHIP is open source [22].

## II. Overview of Profiling and What-If Analyses

We provide a brief overview of profiling an application with OMP-WHIP, the OSPG representation, what-if analyses, and the profiles reported to the user. We illustrate our ideas with an OpenMP program in Figure 1(a) that identifies all prime numbers up to a given bound ($N$). The program uses the `parallel` directive to create a parallel region (line 3 in Figure 1(a)). The runtime creates a team of threads that execute the code in the region in parallel. The `omp for` loop within the parallel region executes the iterations of the loop in parallel. The programmer has balanced the work in the loop using dynamic scheduling because checking the primality of a small number is much faster than a large number. With dynamic scheduling, the runtime divides the iteration space into chunks of size 100 in Figure 1(a).

When the program in Figure 1(a) is executed on a machine with two cores and for an input where `N` is 600, the speedup of the parallel execution compared to the serial execution is 1.37×. This speedup can vary across executions depending on the scheduling of threads on the same machine. Moreover, the speedup on a 2-thread execution provides little information about the speedup on a large number of processors.

**The OMP-WHIP parallelism profiler.** OMP-WHIP is a profiler that measures inherent parallelism in a program for a given input and also enables what-if analyses. To measure inherent parallelism and perform what-if analyses, OMP-WHIP has to identify the series-parallel relationship between program execution fragments that is not dependent on the number of threads. OMP-WHIP lifts the level of abstraction from a thread-based view of the execution to an inherent parallelism view using a novel OpenMP series-parallel graph (OSPG) representation. OMP-WHIP constructs this OSPG representation during program execution and measures computation in any fragment without OpenMP directives.

**OpenMP Series Parallel Graph (OSPG).** OSPG is a directed acyclic graph that precisely encodes the series-parallel relationship between various fragments of dynamic execution. Section III provides formal definitions, properties, and construction of the OSPG. The dynamic execution of a program can be divided into fragments without any OpenMP directives. Each such fragment executes serially and is called a work-node (`W-node`). These are leaves in the OSPG. To maintain the series-parallel relationship between these `W-nodes`, OSPG contains other intermediate nodes — a parallel node (`P-node`) and a sync node (`S-node`). The descendants of two sibling `P-nodes` logically execute in parallel. An `S-node` encodes the fact that a block of code executes in series with another block of code. Given any two work nodes `W1` and `W2` where `W1` is to the left of `W2` in the OSPG, we can determine if they logically execute in parallel by finding the least common ancestor (LCA) of `W1` and `W2` in the OSPG and checking if the child of the LCA on the path to `W1` is a `P-node`. Otherwise, they execute in series.

Figure 1(b) provides the OSPG for the execution of the program in Figure 1(a) on a machine with two cores (2 threads in the team). All useful work (ignoring OpenMP directives) happens in the `W-nodes` (*i.e.*, `W0-W8`). In Figure 1(b), `S1` and `S2` are `S-nodes` added to represent the series-parallel relations induced by `omp parallel` and `omp for` directives, respectively. `P0` and `P1` are `P-nodes` added to model parallel execution by the threads in the team. The chunks of the loop executed by different threads in the team are represented by `P2, P3, P4, P5, P6`, and `P7`. Although different chunks may execute on the same core in a given execution, all these chunks logically execute in parallel from the perspective of measuring inherent parallelism. Two work nodes `W1` and `W2` in Figure 1(b) logically execute in parallel because their least common ancestor is `S3`, whose left child on the path to `W1` is a `P-node` (*i.e.*, `P2`). Using the OSPG, we can identify if any two `W-nodes` execute in series or not.

**Parallelism Profile.** OMP-WHIP's parallelism profile reports the parallelism and the percentage of serial work on the critical path for each directive in the program. Section IV provides a detailed description of our algorithm to compute

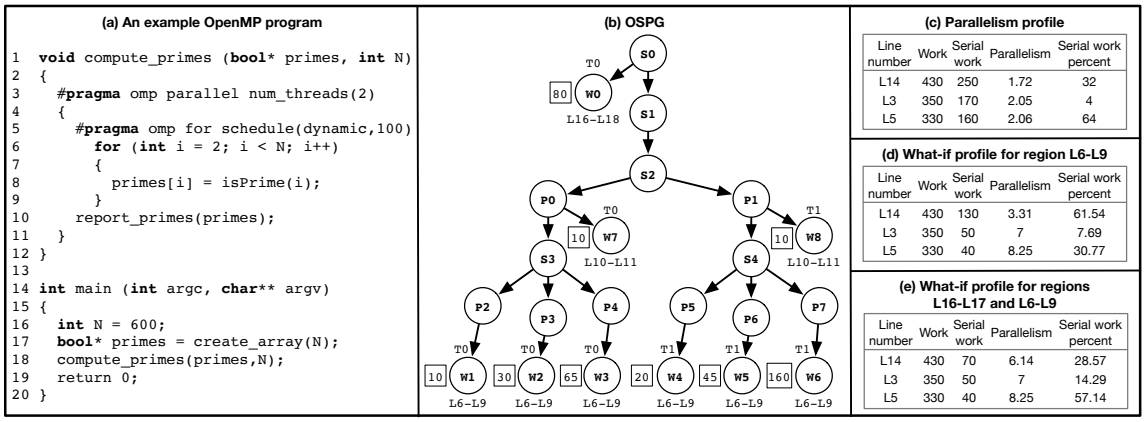


Fig. 1. (a) A simple C++ program to compute prime numbers for a given bound $N$. It creates an array of Boolean values (`primes`) to record if the number is prime. The parallel for loop iterates over the range and checks the number is prime. (b) The performance model for an execution with two threads. It consists of the OpenMP series parallel graph along with work measurement for the leaf nodes (numbers in boxes). We also show the static regions of code corresponding to each leaf node (*e.g.*, L6-L9 for node `W1`). (c) The inherent parallelism profile reported to the user. (d) What-if profile estimating parallelism improvements when region L6-L9 is hypothetically optimized by 4×. (e)What-if profile when regions L6-L9 and L16-L17 are optimized by 4×.

the profile. Figure 1(c) reports the parallelism profile for the OpenMP program in Figure 1(a). It reports that the parallelism for the entire program (line 14 corresponding to the main function) is 1.72, which is the maximum speedup that the program is going to attain irrespective of the number of processors. Similarly the parallelism for the `parallel` directive (line 3 in Figure 1(c)) is 2.05 and the parallelism for the `omp for` directive is 2.06. To improve parallelism, the programmer has to reduce serial work on the critical path in the program. The `omp for` directive, which performs 64% of the serial work on the critical path, is a good place to start.

**What-If Analyses.** OMP-WHIP's performance model allows programmers to estimate increases in parallelism when certain regions of the program are parallelized, which we call what-if analyses. To use them, programmers will demarcate the regions of code that they plan to parallelize with annotations (*i.e.*, `__WHATIF_BEGIN__` and `__WHATIF_END__`) and specify the expected parallelism for the region. OMP-WHIP performs what-if analyses over the performance model and generates a what-if profile (see Section IV-C for details). Figure 1(d) reports the what-if profile when the code region L6-L9 (*i.e.*, lines 6-9 in Figure 1(a)) is optimized by 4×. It shows that the parallelism for `omp for` increases much more than the parallelism of the program. Figure 1(e) reports the profile when both regions L6-L9 and L16-L17 are optimized by 4× using what-if analyses. From the profile in Figure 1(e), one can infer that the programmer has to parallelize both regions to increase parallelism in the program. Now, the programmer can focus on concrete strategies to parallelize them.

## III. OpenMP Series Parallel Graph

OpenMP series parallel graph (OSPG) is inspired by prior series-parallel representations for task parallel programs [17]–[20], [23]. However, the OSPG is designed with two goals: (1) it accurately models the semantics of OpenMP directives to enable the measurement of inherent parallelism and (2) it can be constructed in parallel during program execution.

### A. OSPG and its Properties

OSPG is a directed acyclic graph (DAG) representation of an execution for a given input. Formally, OSPG $G = (V, E)$ is a DAG, where $V$ is the set of nodes and $E$ is the set of directed edges. OSPG consists of three types of nodes: work nodes (`W-node`), parallel nodes (`P-node`), and sync nodes (`S-node`). Hence, $V = V_w \cup V_p \cup V_s$ where $V_w$ represents the set of `W-nodes`, $V_p$ represents the set of `P-nodes`, and $V_s$ represents the set of `S-nodes` in $G$. The edges ($E$) in OSPG ($G$) consists of two sets of directed edges: the set of parent-child edges ($E_{pc}$) and the set of task dependency edges ($E_{dep}$), which models the programmer specified task dependencies between sibling OpenMP tasks. Hence, $E = E_{pc} \cup E_{dep}$.

**Work nodes (W-nodes).** A `W-node` represents the longest sequence of instructions without any OpenMP directive. In a program without any OpenMP directives (*i.e.*, a serial program), the entire execution will be one single `W-node`. In a parallel execution, a `W-node` captures the execution fragment between two successive OpenMP directives. All computation in the program's execution is performed by the `W-nodes`. Hence, the entire execution can be viewed as a collection of `W-nodes`, some of which may execute in parallel.

**Parallel node (P-node).** A `P-node` represents execution fragments that can logically execute in parallel. The dynamic execution fragments represented by the subtree under the `P-node` may execute in parallel with the execution fragments represented by the siblings to the right of the `P-node` or their descendants. For example, when the program creates a new task, a `P-node` will be added to the OSPG to capture the fact that the newly created task will execute in parallel with the continuation. Any computation that occurs within the newly created task will be in the subtree under the `P-node`.

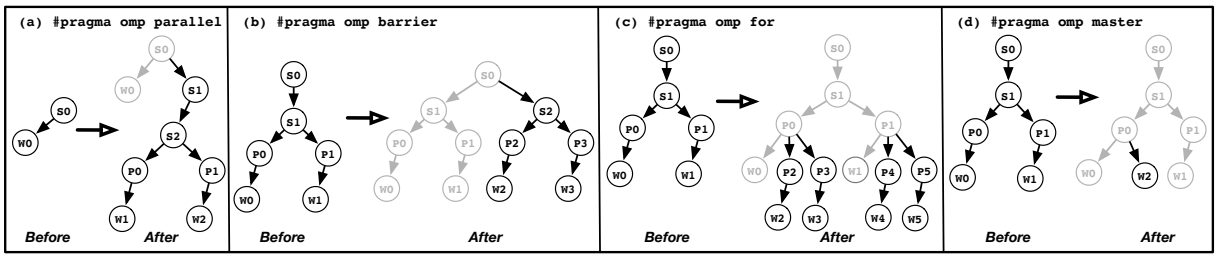


Fig. 2. The OSPG construction for the commonly used directives. The nodes in the OSPG *before* a directive are greyed out in the OSPG *after* the directive.

**Sync nodes (S-nodes).** An `S-node` represents the serial execution of a region of code in relation to the code after the region. All computation and OpenMP directives within the region will be descendants of the `S-node`. For example, the execution of the code block succeeding a `omp parallel` directive will be represented by an `S-node` because the code within the `omp parallel` block executes in series with the code after the `omp parallel` block.

**Properties.** We highlight the main properties of the OSPG that will be useful in computing the parallelism profile:

- The OSPG without the task dependency edges is a tree. Given an OSPG $G = (V, E)$, $G' = (V, E')$ is a tree where $E' = E \setminus E_{dep}$.
- The left to right ordering of sibling nodes represents the logical sequencing of operations in the program. A node $U$ is to the left of node $V$ if $U$ occurs before $V$ in the depth-first traversal of the OSPG without the task dependency edges ($E_{dep}$).
- The descendants of a `P-node` logically execute in parallel with its siblings (and their descendants) to the right of the `P-node`. Similarly, the descendants of an `S-node` execute in series with its siblings (and their descendants) to the right of the `S-node`.
- Two `W-nodes` $U$ and $V$, where $U$ is to the left of $V$, logically execute in parallel if the child of the least common ancestor (LCA) of $U$ and $V$ that occurs on the path from the LCA to $U$, is a `P-node`. Similarly, two `W-nodes` $U$ and $V$, where $U$ is to the left of $V$, execute in series if the child of the LCA of $U$ and $V$ that occurs on the path from the LCA to $U$, is an `S-node`.

### *B. OSPG Construction during Program Execution*

The semantics of the OpenMP directive determines the specific nodes added to the OSPG during execution. When the program encounters a directive during execution, we may not know whether the directive ends with a barrier or not. We have to add `S-nodes` and `P-nodes` appropriately to maintain the series-parallel relationship and the properties of the OSPG. Further, our goal is to measure inherent parallelism even when the program uses work-sharing directives. Hence, we need to capture fragments that can logically execute in parallel even though some chunks of execution are serialized in a given thread (*e.g.*, with `omp for`).

**Per-thread OSPG stack.** The construction of the OSPG happens in parallel as threads execute various OpenMP directives. A thread needs to know where it should add new nodes in the OSPG. Hence, each thread maintains a stack of intermediate nodes that provides information about the slice of the OSPG being executed by it. A stack is needed because multiple nodes can execute in the context of the thread. For example, lets consider the case when a thread executing an OpenMP task waits for its children to complete (*e.g.*, using taskwait). When there are children still waiting to execute, this thread can execute a child task. Eventually, the thread resumes the execution of the original task. Hence, this per-thread stack provides information about where to add nodes in the OSPG.

**Program start.** When a program begins its execution, the thread executing it creates an `S-node` as the root of the OSPG and pushes it to its per-thread stack. It also adds a `W-node` as the child of the root `S-node`, which captures the computation performed before encountering an OpenMP directive.

**The `omp parallel` directive.** This is a work sharing directive. A thread reaching this directive creates a team of one or more threads including itself that execute the following block of code in parallel. The threads in the team synchronize at the end of the parallel region. The thread encountering the directive creates an `S-node` and adds it to the OSPG. It adds an `S-node` because the code within the parallel region executes in series with the code after the parallel region. Figure 2(a) shows the changes to the OSPG on encountering a `parallel` directive, where `S1` is the `S-node` added as part of this step. The parallel region itself can contain other OpenMP directives. When the execution observes a `parallel` directive, we do not know whether the parallel region has barriers within it. We anticipate that such barriers can potentially occur. The master thread executing the `parallel` directive captures it by adding another `S-node` to the OSPG. It is added as the child of the `S-node` corresponding to the `parallel` directive. In Figure 2(a), `S2` corresponds to the `S-node` added in this step.

The code block following `parallel` directive is executed by a team of threads. We capture the parallel execution of the code region by a team of threads as follows. The master thread creates `P-nodes` in the OSPG for each member of the team. These `P-nodes` become the children of the `S-node` that were created in the previous step. In Figure 2(a), `P0` and `P1` represent parallel execution of the region by two threads in the team. Any computation in the subtree under `P0` will happen in

parallel with the computation under the subtree of P1 as the left child of their LCA will be a P-node. The master thread informs other threads in the team where to update the OSPG by pushing the two S-nodes and the P-node corresponding to the thread to their per-thread stack. Finally, the master also adds a W-node as the child of its P-node in the OSPG to capture the computation by it in the parallel region. Other members of the team add W-nodes as the child of their respective P-nodes. In Figure 2(a), W1 and W2 correspond to the computation performed by the two threads in the team.

**Implicit and explicit barriers.** Barriers in OpenMP can be implicit (*e.g.*, a barrier at the end of omp parallel) or explicit when the programmer specifies it using the omp barrier directive. The code before the barrier executes in series with code after it. We have already anticipated this barrier (then in the future) and added an S-node to the OSPG. This barrier signals the end of the subtree under that S-node. At the end of the barrier, the team of threads executes the region following the barrier in parallel.

The first thread reaching the barrier identifies the S-node that was added to the OSPG in anticipation of the barrier and creates a new S-node as its sibling. Since the region following the barrier is executed by the team of threads in parallel, it also creates a P-node for each thread in the team and adds it as the child of the newly created S-node.

Eventually when a thread completes the barrier, it pops the previous P-node and S-node on its per-thread OSPG stack as the subtree under that S-node is now completed. It pushes the newly created S-node and P-node corresponding to the thread on its per-thread OSPG stack, which is the OSPG slice being executed by that thread. It also adds a W-node as the child of the P-node corresponding to the thread to represent the computation done in parallel after the barrier.

In Figure 2(b), when a thread executing either W0 or W1 encounters a barrier, it identifies the S-node corresponding to the region (*i.e.*, S1). The first thread to reach the barrier creates a new S-node (*i.e.*, S2) as the sibling of S1. As there are two threads in the team, there are two P-nodes (P2 and P3) that are children of S2. They represent the computation after the barrier. Any computation done in the subtree under S2 occurs in series with the computation under S1.

**The omp for directive.** This directive creates a parallel for loop using the threads in the team corresponding to the parallel region. The omp for directive, by default, has a barrier at the end of the loop. The programmer can specify a nowait clause to remove the barrier. On encountering this directive, the OpenMP runtime divides the iteration space into chunks and assigns the chunks to the threads in the team. The iteration space can be divided statically or dynamically. Two different chunks can be executed serially by the same thread in an execution even though they are logically parallel.

Our goal is to measure inherent parallelism in the program. We have to measure work performed by each chunk where the chunk size is specified by the user. We also need to add nodes to the OSPG to accurately capture series-parallel relationships between chunks. The omp for directive is encountered within

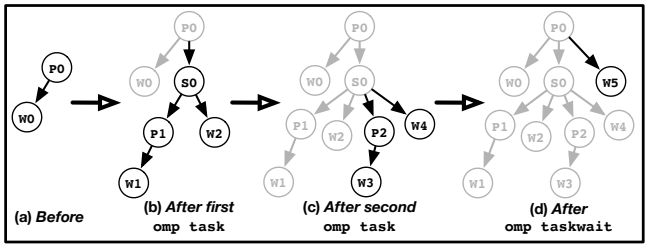


Fig. 3. The OSPG construction for the following sequence of tasking directives: omp task, another omp task following the first task, and an omp taskwait.

the parallel region. The OSPG at that moment already contains a P-node for each thread in the team. For each chunk of the parallel loop, the thread executing the chunk creates a new P-node and adds the newly created P-node as the child of the P-node corresponding to the parallel region. The thread also creates a W-node and adds it as the child of the newly created P-node. The intuition is that we want all chunks that may have executed serially (*i.e.*, on the same thread) in a given execution to be logically parallel with respect to the OSPG. In Figure 2(c), the omp for is executed with two threads (let's say T1 and T2) in the team and each thread executes two chunks. We create two P-nodes (P2 and P3) to represent the chunks executed by thread T1. Each chunk has a work node added to represent the computation performed by the chunk (*e.g.*, W3). Similarly, P4 and P5 represent the chunks executed by thread T2.

**The omp master directive.** The structured code block following the master directive is executed by the master thread of the team. We create a W-node to clearly attribute the serial work performed in the code block. Since the master thread is already executing within the parallel region, it adds the newly created W-node as the child of its P-node (corresponding to the parallel region) in the OSPG. Figure 2(d) presents the OSPG before and after executing the master directive.

**The omp critical directive.** This directive specifies a region of code that must be executed by only one thread in the team at a time. Our goal is to measure inherent parallelism. Hence, we should not measure the waiting times experienced in a specific execution. We split the W-node executing this directive into three W-nodes: one representing the work before the critical directive, one representing the work in the critical, and one for the computation after the critical section.

**The omp task directive.** This directive is used to specify a new task. The code block following the omp task directive executes in parallel with the code after the code block. To measure inherent parallelism, we have to identify the series-parallel relationships between various fragments of tasks.

When the program encounters a omp task directive, we do not know whether the program executes a taskwait directive in the future execution (similar to barriers within parallel regions). The thread executing the omp task directive creates an S-node in the OSPG if it is the first task in the

program or is the first task after the `taskwait` directive. The subtree under this `S-node` represents a set of parallel tasks that end with a `taskwait`. The thread also adds a `P-node` as the child of this `S-node` to represent the newly created task and a `W-node` as the immediate right sibling of the `P-node` to represent the computation in the continuation. When any thread executes the newly created task, it adds a `W-node` as the child of the `P-node` to represent the computation performed by the new task. Figure 3(b) presents the OSPG after the first `omp task` directive has been encountered.

On any subsequent `omp task` directive before the next `omp taskwait` directive, there is no need to create an `S-node`. We just add a `P-node`, a `W-node` as the child of the `P-node`, and a `W-node` as the immediate right sibling of the `P-node` representing the task. Figure 3(c) presents the OSPG after encountering a second `omp task` directive.

**The `omp taskwait` directive.** The task executing the `omp taskwait` directive will wait for all its children to complete execution before it proceeds with the current task's execution. When the program executes the `omp taskwait` directive, the code region after this directive executes in series with the code before it. We have already anticipated a `taskwait` (then in the future) on the first task and added an `S-node`. This directive finishes the computation under the subtree of that `S-node`. We add a `W-node` as the immediate right sibling of this `S-node` to capture the work performed after `taskwait`. Figure 3(d) presents the OSPG after executing the `taskwait` directive.

## IV. Parallelism Profile and What-If Analyses

OMP-WHIP constructs the OSPG as the program executes on a multi-core machine for a given input. The profiler gets invoked via callbacks from the OpenMP runtime when the program executes a specific OpenMP directive. Apart from constructing the OSPG, OMP-WHIP also measures the total amount of computation performed in each `W-node` by programmatically using hardware performance counters. The OSPG and the fine-grained measurement of work constitutes an accurate performance model of the execution for a given input, which enables what-if analyses (Section IV-C).

OMP-WHIP does not maintain the entire OSPG in memory. Instead, it maintains a small slice of the OSPG in memory at any point in time during profile execution (*i.e.*, whatever information is maintained by the per-thread stacks), which ensures low memory overhead. Further, OMP-WHIP's profile execution has two modes, each with its own set of benefits. The first mode writes the OSPG data to a file after the node has finished and is popped from the per-thread stack, which is subsequently used by offline analyses. Although memory overhead is low, the log files generated can be large for long running applications. However, the presence of the entire OSPG enables what-if analyses for some programs without re-execution. The second mode directly computes the parallelism profile on-the-fly without requiring any offline analyses. It is attractive for long-running applications. Next, we describe the generation of parallelism and what-if profile with an offline analysis. We describe the on-the-fly mode in Section IV-D.

```
1  function WhatIfParallelismProfile(G, R, f)
2      foreach N in bottom-up traversal of G do
3          if what-if-mode then
4              C_N ← ChildNodes(N)
5              work ← Σ_{C∈C_N} C.w
6              W_R ← {W | W ∈ C_N ∧ W ∈ R}
7              foreach W ∈ W_R do
8                  W.w ← W.w/f
9              end
10             ⟨w, N.s, N.l⟩ ← ComputeNode(N)
11             N.w ← work
12         end
13         else
14             ⟨N.w, N.s, N.l⟩ ← ComputeNode(N)
15         end
16     end
17     AggregatePerStaticDirective(G)
18     return
```

Fig. 4. Algorithm to produce the parallelism profile and the what-if profile given input OSPG $G$, regions annotated for what-if analyses $R$ and optimization factor $f$. ChildNodes returns all the child nodes of the input node. AggregatePerStaticDirective aggregates the work and serial work to produce the parallelism profile and the what-if profile similar in format to Figure 1(c), (d), and (e).

### A. Offline Analysis to Compute the Parallelism Profile

The performance model has work information with each `W-node` and the OSPG accurately encodes the series-parallel relationship between various nodes. The task of the offline analysis is to compute parallelism for each intermediate node, which in turn requires it to compute total work and serial work for each node. We want to accurately attribute serial work to various nodes that contribute to the critical path under the subtree. The critical path of an intermediate node is the longest chain of nodes that have to be executed in series and perform the highest serial work under the subtree. Hence, the offline analysis also computes the list of nodes that contribute to the critical path in the subtree under the intermediate node.

The offline analysis performs a bottom-up traversal of the OSPG to compute the parallelism for each intermediate node. Figure 4 and Figure 5 present the algorithm to compute the parallelism for a program. The algorithm computes three quantities with each intermediate node: (1) work (*w*), (2) serial work (*s*), and (3) list of `W-nodes` that perform serial work on the critical path (*l*). These quantities are computed using the series-parallel relationship encoded by the OSPG and the information of its children, which would have been computed earlier in the bottom-up traversal. Figure 5 presents the algorithm to compute work (`w`), serial work (`s`), and the list of `W-nodes` performing serial work on the critical path (`l`) for each intermediate node. The total work under the subtree at an intermediate node is equal to the sum total of the work performed by the children (line 3 in Figure 5).

**Computing serial work.** To compute serial work for an intermediate node, we have to identify the chain of `W-nodes`

```
1  function ComputeNode(N)
2      C_N ← ChildNodes(N)
3      w ← Σ_{C∈C_N} C.w
4      W_N ← WChildNodes(N)
5      S_N ← SChildNodes(N)
6      s ← Σ_{W∈W_N} W.w + Σ_{S∈S_N} S.s
7      l ← (⋃_{S∈S_N} S.l) ∪ W_N
8      foreach P ∈ PChildNodes(N) do
9          LP_D ← LongestLeftDepChain(P)
10         LW_P ← LeftWNodeSiblings(P)
11         LS_P ← LeftSNodeSiblings(P)
12         cw ← Σ_{D∈LP_D} D.s + Σ_{W∈LW_P} W.w + Σ_{S∈LS_P} S.s + P.s
13         if cw > s then
14             s ← cw
15             l ← LW_P ∪ (⋃_{S∈LS_P} S.l) ∪ (⋃_{D∈LP_D} D.l) ∪ P.l
16         end
17     end
18     return ⟨w, s, l⟩
```

Fig. 5. Algorithm to compute work, serial work, and the list of `W-nodes` on the critical path for the input node $N$ in the OSPG. WChildNodes, SChildNodes, and PChildNodes return the children `W-nodes`, `S-nodes`, and `P-nodes` of the input node, respectively. LeftWNodeSiblings and LeftSNodeSiblings return all the left `W-node` and `S-node` siblings of the input node, respectively. LongestLeftDepChain returns the set of `P-nodes` that are the left siblings of the input node, dependent on the input node through task dependency edges, and perform the highest serial work.

that execute serially and perform the highest amount of serial work in the subtree. We use the properties of the OSPG to identify children that execute serially. In an OSPG without task dependency edges, descendants of a `P-node` will execute in parallel with the siblings to the right of the `P-node` and their descendants. Similarly, the descendants of the `P-node` execute in series with the `W-nodes` and the descendants of `S-nodes` to the left of the `P-node`. In the absence of `P-nodes`, the `W-nodes` and the `S-nodes` that are the children of a given node execute serially. Each `P-node` creates a separate chain of nodes in the subtree. All nodes in a single chain execute serially. However, two such chains in the subtree logically execute in parallel. Hence, there could be multiple such chains of nodes that perform serial work for a given intermediate node. Any of these chains of nodes can perform the highest serial work and constitute the critical path. The task dependency edges between sibling `P-nodes` serializes the subtrees under the two `P-nodes`. These dependencies can be transitive (*e.g.*, `P3` depends on `P2` and `P2` depends on `P1`). These dependency edges also need to be considered while computing serial work.

Figure 5 illustrates the computation of serial work for the subtree under a given node. It first considers the serial work of the chain of `W-nodes` and `S-nodes` that are direct children of the node under consideration (lines 4-6 in Figure 5). As each `P-node` creates a chain of nodes, the algorithm computes the serial work of that chain (lines 8-17 in Figure 5). The serial work of the chain created by the `P-node` is equal to the sum of: (1) work performed by `W-nodes` to the left (line 10) , (2) serial work performed by `S-nodes` to the left (line 11), (3) total serial work performed by the transitively dependent `P-nodes` to the left, and (4) the serial work performed by the subtree under the current `P-node`. If this chain's serial work is greater than the serial work seen till now, then the chain of nodes including the current `P-node` becomes the new critical path (lines 13-16 in Figure 5).

**Computing the list of `W-nodes` on the critical path.** Each intermediate node maintains the list of `W-nodes` that are on the critical path. Initially, the list of nodes on the critical path is the union of all children `W-nodes` and the union of lists of children `S-nodes` as it constitutes the critical path (line 7 in Figure 5). When the algorithm identifies a new critical path, the list of `W-nodes` on the critical path needs to be updated. When a `P-node` becomes part of the critical path, the list of nodes contributing to the critical path is updated to be the union of the following sets:(1) set of `W-nodes` to the left of the `P-node`, (2) union of the lists of nodes on the critical path within the subtree under the `S-nodes` to the left, (3) union of the list of nodes on the critical path within the subtree under dependent `P-nodes` to the left, and (4) list of nodes on the critical path under the current `P-node` (line 15 in Figure 5). At the end of the bottom-up traversal, each internal node will have work, serial work, and list of `W-nodes` on the critical path.

### *B. Summarizing the Profile for each Static Directive*

Although the dynamic information computed above on the OSPG highlights the parallelism in the program, the programmer can perform concrete optimizations if we attribute this information to static OpenMP directives. There are multiple granularities for aggregating dynamic information from the OSPG to static locations: aggregation per static location of the directive or aggregation with respect to dynamic calling contexts per static location (*e.g.*, using libunwind). OMP-WHIP, by default, restricts itself to aggregation per static location. As part of its construction, each internal node in the OSPG maintains static location information of the directive (*e.g.*, line number and filename). To aggregate work and serial work information, the offline analysis maintains a map with location information as the key and work/serial work as the values. It updates the values in the map by performing another bottom-up traversal of the OSPG. When performing this aggregation with a bottom-up traversal, we should be careful not to double count work and serial work when the subtree under the current node has another node with the same location information. Such scenarios can arise with recursive decomposition using tasks. To measure work and serial work accurately, the offline analysis has to subtract the values of the nodes in the subtree with same location information and then, add the values of the current node.

To help programmer identify specific directives exclusively performing serial work on the critical path, the analysis

aggregates this information using the list of `W-nodes` on the critical path (associated with the root node). The final profile reports the parallelism and the percentage of serial work on the critical path performed by each directive.

### C. What-If Analyses over the Performance Model

OMP-WHIP's parallelism profile highlights the OpenMP directives that perform significant serial work on the critical path and have low parallelism. These directives are likely candidates for optimization. However, the program can have multiple parallel chains of nodes that perform similar amounts of serial work. Optimizing one of these chains may not increase the parallelism. Designing a concrete parallelization strategy takes time as it may require significant changes to the program. A programmer would like to know whether the parallelism increases on changes to a particular region.

We propose what-if analyses to estimate the improvement in parallelism when a region of the program is parallelized. It enables the user to identify changes in parallelism even before concrete parallelization strategies have been designed. The key enabler for these analyses is the performance model that encodes series-parallel relationship between various `W-nodes` and the measurement of work in these `W-nodes`. Parallelizing a region effectively reduces the serial work performed by it. Hence, our what-if analyses estimate the improvement in parallelism by reducing the serial work of the regions of interest while keeping the total work exactly as before, which mimics the effect of parallelization.

To use what-if analyses, programmers annotate the region of code that they plan to optimize using OMP-WHIP's annotations. OMP-WHIP re-executes the program to accurately measure the amount of computation performed solely in the annotated regions. Each static annotation can correspond to multiple `W-nodes` in the dynamic execution. If the annotated regions exactly correspond to `W-nodes` in the performance model before annotations, a re-execution is not necessary. Otherwise, when OMP-WHIP re-executes the annotated program, it measures the total amount of computation in the `W-node` before, within, and after the annotated region.

In the offline analysis, OMP-WHIP produces a what-if profile that reports the parallelism and the serial work on the critical path for each directive when the annotated regions are optimized by the user-specified threshold. Figure 4 presents the algorithm to perform what-if analyses (lines 4-11 in Figure 4). When OMP-WHIP performs its what-if analyses, it performs a bottom-up traversal of the OSPG. For each intermediate node, it computes the total work under the subtree as before (lines 5 and 11 in Figure 4). Then it computes the serial work and the list of `W-nodes` on the critical path after it has reduced the serial work of the `W-nodes` corresponding to the annotated regions by the user-specified threshold (lines 6-9 in Figure 4).

### D. On-the-fly Parallelism Profile

OMP-WHIP's second mode computes the profile on-the-fly and does not generate OSPG log files. To compute profiles on-the-fly, OMP-WHIP maintains two additional quantities (along with `w`, `s`, and `l` as before) with each node on the per-thread stack : (1) total serial work performed by the sibling `W-nodes` and `S-nodes` to the left, and (2) the list of `W-nodes` performing the highest serial work among the left `W-node` siblings and the left `S-node` siblings and their descendants. When a new node is pushed to the per-thread stack, the serial work performed by the siblings to the left has already been measured and these two additional quantities are the initialized by the parent. When the node is popped, OMP-WHIP checks if the node is part of the critical path and updates the parent's work, serial work, and list of `W-nodes` on the critical path (similar to lines 12-16 in Figure 5).

## V. Experimental Evaluation

**Prototype.** The OMP-WHIP prototype profiles C/C++ OpenMP programs and consists of two modules: (1) a static library that is linked with the application during compilation, which supports both offline and on-the-fly profiling modes, and (2) a standalone analyzer to perform offline analysis. The profile execution library uses OMPT [21] to intercept calls to the OpenMP runtime to construct the OSPG and to perform fine-grained work measurement. The current OMP-WHIP prototype works with LLVM+Clang-5.0 [1]. We modified the Clang-5.0 OpenMP runtime to add new OMPT callbacks to support parallel for loops with dynamic scheduling and tasks to measure inherent parallelism. OMP-WHIP uses the `perf events` module in Linux to read hardware performance counters to measure work. OMP-WHIP can measure work with execution cycles and dynamic instructions, which estimates work ignoring secondary effects of execution (*e.g.*, cache misses). OMP-WHIP supports OpenMP-4.5's common core including work-sharing and tasking directives. It does not support offloading and nested parallelism yet. Our prototype is open source [22].

To test the effectiveness of the profiler, we profiled 43 OpenMP applications from Sequoia [24], Coral [25], PBBS [26], BOTS [27], Kastors [28], and NAS [29] benchmark suites. All experiments were performed on a 16-core 2.1 GHz Intel Xeon machine. The profiler with post-mortem analysis is 80% slower on average when compared to parallel execution without any profiling code. The time taken for post-mortem analysis depends on the size of the OSPG (ranges from seconds to hours). In contrast, the on-the-fly profiler is 62% slower on average compared to an execution without profiling. The resident memory overhead is on average 28% in both modes compared to the baseline.

**Profiling work-sharing applications with OMP-WHIP.** We used OMP-WHIP to measure inherent parallelism and perform what-if analyses with 32 applications (8 Coral/Sequoia applications, 16 PBBS programs ported to OpenMP, and 8 NAS applications) with work-sharing directives. We found bottlenecks in all of them. On average, we identified 2 or more regions in these applications through what-if analyses. Subsequently, we parallelized one PBBS and two Coral (AMGmk and Quicksilver) applications.

[1] We are working on the OMP-WHIP prototype for LLVM+Clang-6.0 [22], which partially implements the latest specification of OpenMP [2].

**I. AMGmk**

| Directive Location | Parallel -ism | Serial work percent |
|---|---|---|
| Program | **6.93** | 47.95 |
| relax.c:91 | 13.63 | 29.17 |
| csr.c:172 | 10.57 | 20.36 |
| relax.c:87 | 11.53 | 1.58 |

(a) Initial parallelism profile

| Directive Location | Parallel -ism | Serial work percent |
|---|---|---|
| Program | **11.62** | 20.47 |
| relax.c:91 | 13.11 | 52.32 |
| csr.c:172 | 10.56 | 21.43 |
| relax.c:87 | 11.4 | 2.75 |

(b) What-if profile

| Directive Location | Parallel -ism | Serial work percent |
|---|---|---|
| Program | **11.43** | 17.44 |
| relax.c:91 | 13.11 | 51.87 |
| csr.c:179 | 15.79 | 21.31 |
| vect.c:383 | 9.14 | 3.52 |

(c) Final parallelism profile

**II. Quicksilver**

| Directive Location | Parallel -ism | Serial work percent |
|---|---|---|
| Program | **13** | 8.85 |
| mc.hh:4 | 14.16 | 91.15 |

(a) Initial parallelism profile

| Directive Location | Parallel -ism | Serial work percent |
|---|---|---|
| Program | **86.79** | 59.12 |
| mc.hh:4 | 210.9 | 40.88 |

(b) What-if profile

| Directive Location | Parallel -ism | Serial work percent |
|---|---|---|
| Program | **85.37** | 55.59 |
| mc.hh:4 | 190.98 | 44.41 |

(c) Final parallelism profile

**III. MinSpanningForest**

| Directive Location | Parallel -ism | Serial work percent |
|---|---|---|
| Program | **2.94** | 97.82 |
| seq.h:403 | 5.87 | 1.44 |
| seq.h:429 | 20.61 | 0.48 |
| spec.h:72 | 693.1 | 0.12 |

(a) Initial parallelism profile

| Directive Location | Parallel -ism | Serial work percent |
|---|---|---|
| Program | **89.41** | 14.62 |
| sort.h:126 | 51.78 | 31.41 |
| sort.h:100 | 59.83 | 28.53 |
| seq.h:443 | 20.59 | 13.47 |

(b) What-if profile

| Directive Location | Parallel -ism | Serial work percent |
|---|---|---|
| Program | **95.83** | 9.48 |
| sort.h:126 | 52.09 | 33.24 |
| sort.h:100 | 59.76 | 30.2 |
| seq.h:443 | 20.57 | 14.23 |

(c) Final parallelism profile

**IV. Nbody**

| Directive Location | Parallel -ism | Serial work percent |
|---|---|---|
| Program | **1.13** | 98.96 |
| seq.h:123 | 1.38 | 0.44 |
| seq.h:403 | 1.86 | 0.19 |
| CK.C:226 | 11.26 | 0.14 |

(a) Initial parallelism profile

| Directive Location | Parallel -ism | Serial work percent |
|---|---|---|
| Program | **15.2** | 85.79 |
| seq.h:123 | 1.37 | 5.99 |
| seq.h:403 | 1.86 | 2.55 |
| CK.C:227 | 11.07 | 1.88 |

(b) What-if profile

| Directive Location | Parallel -ism | Serial work percent |
|---|---|---|
| Program | **85.54** | 17.32 |
| seq.h:123 | 1.39 | 32.53 |
| seq.h:403 | 1.86 | 14.29 |
| CK.C:227 | 11.17 | 10.37 |

(c) Final parallelism profile

Fig. 6. The original parallelism profile, the what-if profile , and the final parallelism profile after the annotated regions were parallelized for a sample of the applications profiled using OMP-WHIP. The topmost line in each profile reports the inherent parallelism of the program and the percentage of serial work performed in the program. The rest, list the parallelism and the percentage of serial work for an OpenMP directive. We list only the top three directives ordered according to the percentage of serial work.

**Speedup with AMGmk.** The AMGmk program from the LLNL Coral benchmark suite [25] is an algebraic multigrid solver that uses OpenMP for parallelization. The program is made of three main kernel functions - a compressed sparse matrix multiply, an algebraic mesh relaxation, and a vector operation. Initially, the speedup over serial execution on a 16-core machine was 5.39×. To understand bottlenecks, we generated OMP-WHIP's inherent parallelism profile, which is shown in Figure 6(I)(a). The entire program has a parallelism of 6.93 and regions that are not associated with any directive exclusively perform 47.95% of the serial work (line corresponding to Program in Figure 6(I)(a)). On examining the source code, we observed that parallel regions were interspersed with serial code spanning multiple files. We annotated the first such region and used OMP-WHIP's what-if analyses to obtain an estimate of the increase in parallelism when this region is optimized by 16×. The what-if profile showed a slight increase in parallelism from 6.93 to 8.02. To further increase the parallelism, we annotated one more serial region and generated the what-if profile when both the regions are hypothetically parallelized on 16-cores. Figure 6(I)(b) shows the what-if parallelism profile where the parallelism of the program increases to 11.62. If we optimize these two regions, the parallelism almost doubles.

Subsequently, we examined these regions closely to parallelize them. The first region in the sparse matrix multiply kernel contained for loops that initialized a vector and performed the algebraic operation on each element in the vector. Similarly, the second region contained the vector operation kernel with for loops. We parallelized both regions with the `omp parallel for` directive. The parallelism profile after concrete parallelization of the program is shown in Figure 6(I)(c), which almost attains the same parallelism as the estimate from what-if analyses. There is a small difference between the estimate and the final measured parallelism because the regions for what-if analyses contained other code along with the parallelized loops. Moreover, the speedup of the program increased from 5.39× to 9.13× on a 16-core machine.

**Speedup with Quicksilver.** Quicksilver [30] is a Coral application that solves a simplified dynamic Monte Carlo particle transport problem. We configured it to run on a single node machine and used the Coral2_P_1 input to profile the program. The program has a parallelism of 12.99 (see Figure 6(II)(a)). It has a single `omp parallel for` loop that iterates over the number of particles being simulated. We annotated the body of the loop and used what-if analyses to check if the parallelism increases when it is optimized by 16×. Figure 6(II)(b) presents the what-if profile that shows the increase in parallelism for both the directive (to 210.9) and the program (to 86.79). We noticed that the loop was using static scheduling and there was load imbalance. We changed the loop to use dynamic scheduling. Figure 6(II)(c) shows the final parallelism profile after these optimizations. The speedup also increased from 11.83× to 12.8× on a 16-core machine.

**Speedup with Delaunay Triangulation.** This PBBS application [26] computes the delaunay triangulation of a given set points. It had a speedup of 1.08×. We identified a loop that matters using OMP-WHIP's what-if analyses. Subsequent parallelization increased the speedup from 1.08× to 9.23× [22].

**Profiling applications with tasking.** Apart from the applications that use work-sharing directives, we identified bottlenecks in 11 applications using tasking directives from the BOTS [27] and the Kastors [28] benchmark suites. We increased the speedup of five applications using OMP-WHIP's what-if analyses: four applications from PBBS and one application from the Kastors suite. The Strassen benchmark, which uses tasking and task dependencies, performs parallel recursive matrix multiplication. OMP-WHIP's what-if analyses highlighted that reducing the work in the base case can improve parallelism. When we reduced the work in the base case, the speedup increased from 14× to 15.6× on a 16-core machine.

**Speedup with Minimum Spanning Forest.** This program computes the minimum spanning forest of a graph using the parallel Kruskal algorithm. It initially had a speedup of 1.95× over serial execution. It had low parallelism (*i.e.*, 2.94 in Figure 6(III)(a)). OMP-WHIP's what-if analyses helped us find two regions which when hypothetically optimized by 16× can increase the parallelism to 35.7. We noticed that these regions were calling the serial sort function and we replaced them with the parallel sort function already present in the program. The speedup increased from 1.95× to 7.6×. When we ran the modified program with OMP-WHIP's what-if analyses, it showed that optimizing regions in the

functions called by parallel sort can improve parallelism to 89.41 (see Figure 6(III)(b)). We parallelized the recursive block transpose function within this region using OpenMP tasks. The program's parallelism increased to 95.83 (see Figure 6(III)(c)). The parallelism in the final parallelism profile is higher than the estimate from what-if analyses because what-if analyses hypothetically optimized the region by 16× (assuming parallel execution on 16 cores). However, our parallelization created more tasks. Our changes increased the speedup from initial 1.95× to 8.9×.

**Speedup with NBody.** NBody computes the gravitational force vector of each point due to all other points. The OpenMP port of this PBBS application primarily uses `omp for` directives. Hence, the speedup was 1.08× and parallelism was 1.13 (see Figure 6(IV)(a)). Our what-if analyses suggested that parallelizing code in the non-parallel regions by 16× can increase the parallelism to 15.2 (see Figure 6(IV)(b)). We identified a recursive function that was invoked on a disjoint set of vertices in the non-parallel region. We transformed it to recursively spawn tasks, which increased the parallelism to 85.54 (see Figure 6(IV)(c)). These changes increased the speedup from 1.08× to 14.77×.

**Speedup with ConvexHull.** This PBBS application computes the convex hull of points in a 2D space. It is also parallelized with work-sharing directives. It had low parallelism and low speedup. Our what-if analyses helped us find two regions that can improve parallelism. We optimized one region using the `omp for` loop and another using recursive decomposition with tasks. The parallelism increased from 2.51 to 220. The speedup increased from 2.11× to 11.1×.

**Comparison with other OpenMP tools**. As a point of comparison, we also profiled the above 7 applications with three existing tools: Intel VTune Amplifier [5], HPCToolkit [10] and Scalasca [14], [31]. These tools either measure work or a set of metrics to characterize performance and attribute it to the dynamic call-sites and OpenMP regions. Although these tools highlight the OpenMP regions with load imbalance, they do not identify all the serialization bottlenecks that were identified by OMP-WHIP. For example, HPCToolkit and Scalasca both identify two `omp parallel` regions in AMGmk and ConvexHull as the root cause of load imbalance. When we parallelized these regions, the speedup and the parallelism did not change. Further, these regions are different from the serialization bottlenecks identified by OMP-WHIP. Similarly, VTune does not highlight the bottlenecks in AMGmk but is able to highlight only one out of the two regions in ConvexHull.

## VI. Related Work

A large body of work exists on profiling OpenMP programs [3], [5]–[8], [10], [11], [13], [14], [32]. Profiling tools like Intel VTune Amplifier [5], TAU [9] and HPCToolkit [10] use hardware event-based sampling to compute performance metrics and associate the metrics to the source code calling context. Trace-based performance analysis tools [11], [13], [14], [33]–[35] provide a way to visualize the execution time behavior of an application, which can be used to identify load imbalances. To determine the root cause of performance problems, tools that identify functions on the critical path have been proposed [31], [36]–[40]. Numerous performance modeling tools that can be used to identify critical paths and possible scalability bottlenecks have also been explored [41]–[45]. These techniques present a thread-based view of execution, which provides little information about parallelism, varies across schedules, and does not enable what-if analyses.

**Parallelism estimates.** There are numerous efforts to measure either inherent parallelism or provide a speedup estimate [20], [46]–[51]. One can provide an estimate of execution by comparing performance metrics from executions using expectations [52], work-time inflation [53], difference of profiles [54], and/or memory access time expectations [45]. *Parallel Prophet* [50] and Kismet [48] provide an estimate of potential speedup from parallelizing sequential programs. Cilk tools [46], [47] measure inherent parallelism in an online fashion without maintaining an explicit series-parallel representation. However, these techniques require serial execution. Dimemas [55], [56] is a simulation based performance analysis tool for MPI programs that estimates performance improvement by optimizing application functions and network communication. Unfortunately, it primarily highlights improvements for bottlenecks observed in a specific execution.

**Causal profilers.** Our work is closely related to causal profiling for parallel programs [20], [57]. COZ [57] measures the benefit of optimizing a line of the program by virtually speeding up the line of code. It slows down all other parallel threads and develops an analytical model to compute the speedup. Coz estimates the speedup in a particular execution. It cannot estimate performance benefits in task parallel environments in the presence of work stealing. TASKPROF [20] measures inherent parallelism, builds a series-parallel relationship for task parallel programs, and can also estimate the improvements in parallelism. OMP-WHIP is inspired by TASKPROF. In contrast, OMP-WHIP includes a novel series-parallel graph abstraction that handles both work-sharing and tasking constructs in OpenMP and a novel algorithm to construct the OSPG.

## VII. Conclusion and Future Work

We make a case for measuring inherent parallelism and built the OMP-WHIP profiler to measure it. The novel OSPG representation of the execution enables OMP-WHIP to precisely identify the series-parallel relationship between various program fragments. The fine-grained measurements and the OSPG together constitute a performance model of an execution, which enables us to effectively identify bottlenecks that matter. As future work, we will extend OMP-WHIP to identify regions experiencing secondary effects, identify appropriate grain sizes to minimize OpenMP runtime overhead, and support offloading and hybrid MPI+OpenMP programs.

## Acknowledgments

We thank the anonymous reviewers for their feedback. This paper is based on work supported in part by NSF CAREER Award CCF-1453086 and NSF Award CNS-1441724.

## REFERENCES


[1] OpenMP Architecture Review Board, "OpenMP 4.5 complete specification," Nov. 2015. [Online]. Available: http://www.openmp.org/wp-content/uploads/openmp-4.5.pdf

[2] ——, "OpenMP 5.0 public comment draft," Jul. 2018. [Online]. Available: http://www.openmp.org/wp-content/uploads/openmp-TR7.pdf

[3] K. Fürlinger and M. Gerndt, "ompP: A profiling tool for OpenMP," in *Proceedings of the 2005 and 2006 International Conference on OpenMP Shared Memory Parallel Programming*, ser. IWOMP'05/IWOMP'06, 2008, pp. 15–23.

[4] B. Mohr, A. D. Malony, S. Shende, and F. Wolf, "Design and prototype of a performance tool interface for OpenMP," *Journal of Supercomputing*, pp. 105–128, 2002.

[5] Intel. Corporation. (2018) Intel VTune Amplifier 2018. [Online]. Available: https://software.intel.com/en-us/intel-vtune-amplifier-xe

[6] Cray Inc. (2015) Using Cray Performance Measurement and Analysis Tools. [Online]. Available: http://docs.cray.com/PDF/Cray_Performance_Measurement_and_Analysis_Tools_User_Guide_640.pdf

[7] (2018) Arm MAP. [Online]. Available: https://developer.arm.com/products/software-development-tools/hpc/arm-forge/arm-map

[8] The Portland Group, "PGI profiler user's guide - PGI compilers," 2017. [Online]. Available: https://www.pgroup.com/doc/pgprofug.pdf

[9] S. S. Shende and A. D. Malony, "The Tau parallel performance system," *International Journal of High Performance Computing Applications*, pp. 287–311, 2006.

[10] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent, "HPCToolkit: Tools for performance analysis of optimized parallel programs http://hpctoolkit.org," *Concurrency and Computation : Practice & Experience - Scalable Tools for High-End Computing*, pp. 685–701, 2010.

[11] (2018) Paraver. [Online]. Available: https://tools.bsc.es/

[12] (2018) ParaFormance. [Online]. Available: https://www.paraformance.com/

[13] A. Knüpfer, H. Brunst, J. Doleschal, M. Jurenz, M. Lieber, H. Mickler, M. S. Müller, and W. E. Nagel, "The Vampir performance analysis tool-set," in *Tools for High Performance Computing*, 2008, pp. 139–155.

[14] F. Wolf, B. J. N. Wylie, E. Ábrahám, D. Becker, W. Frings, K. Fürlinger, M. Geimer, M.-A. Hermanns, B. Mohr, S. Moore, M. Pfeifer, and Z. Szebenyi, *Usage of the SCALASCA toolset for scalable performance analysis of large-scale parallel applications*, 2008, pp. 157–167.

[15] Oracle. (2017) Oracle Developer Studio. [Online]. Available: https://www.oracle.com/tools/developerstudio/index.html

[16] X. Liu, J. Mellor-Crummey, and M. Fagan, "A new approach for performance analysis of OpenMP programs," in *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ser. ICS, 2013, pp. 69–80.

[17] R. Raman, J. Zhao, V. Sarkar, M. Vechev, and E. Yahav, "Scalable and precise dynamic datarace detection for structured parallelism," in *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI, 2012, pp. 531–542.

[18] A. Yoga, S. Nagarakatte, and A. Gupta, "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*, ser. FSE, 2016, pp. 833–845.

[19] M. Feng and C. E. Leiserson, "Efficient detection of determinacy races in Cilk programs," in *Proceedings of the 9th ACM Symposium on Parallel Algorithms and Architectures*, ser. SPAA, 1997, pp. 1–11.

[20] A. Yoga and S. Nagarakatte, "A fast causal profiler for task parallel programs," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2017, 2017, pp. 15–26.

[21] A. Eichenberger, J. Mellor-Crummy, M. Schulz, N. Copty, J. Cownie, R. Dietrich, X. Liu, E. Loh, and D. Lorenz, "OpenMP technical report 2 on the ompt interface," Mar. 2014. [Online]. Available: http://www.openmp.org/wp-content/uploads/ompt-tr2.pdf

[22] N. Boushehrinejadmoradi, A. Yoga, and S. Nagarakatte. (2018) OMP-WHIP. [Online]. Available: https://github.com/rutgers-apl/omp-whip

[23] R. Raman, "Dynamic data race detection for structured parallelism," Ph.D. dissertation, Rice University, 2012.

[24] Lawrence Livermore National Laboratory. (2018) LLNL sequoia benchmarks. [Online]. Available: https://asc.llnl.gov/sequoia/benchmarks

[25] CORAL benchmarks. [Online]. Available: https://asc.llnl.gov/CORAL-benchmarks/

[26] J. Shun, G. E. Blelloch, J. T. Fineman, P. B. Gibbons, A. Kyrola, H. V. Simhadri, and K. Tangwongsan, "Brief announcement: The problem based benchmark suite," in *Proceedings of the 24th Annual ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA, 2012, pp. 68–70.

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

[28] P. Virouleau, P. Brunet, F. Broquedis, N. Furmento, S. Thibault, O. Aumage, and T. Gautier, "Evaluation of OpenMP dependent tasks with the KASTORS benchmark suite," in *Using and Improving OpenMP for Devices, Tasks, and More*, 2014, pp. 16–29.

[29] NAS parallel benchmarks. [Online]. Available: https://www.nas.nasa.gov/publications/npb.html

[30] (2017) A proxy app for the monte carlo transport code, mercury. llnl-code-684037. [Online]. Available: https://github.com/LLNL/Quicksilver

[31] D. Böhme, F. Wolf, B. R. de Supinski, M. Schulz, and M. Geimer, "Scalable critical-path based performance analysis," in *2012 IEEE 26th International Parallel and Distributed Processing Symposium*, 2012, pp. 1330–1340.

[32] H. Servat, G. Llort, K. Huck, J. GiméNez, and J. Labarta, "Framework for a productive performance optimization," *Parallel Comput.*, vol. 39, no. 8, pp. 336–353, 2013.

[33] M. Schulz, J. Galarowicz, D. Maghrak, W. Hachfeld, D. Montoya, and S. Cranford, "Open | SpeedShop: An open source infrastructure for parallel performance analysis," *Sci. Program.*, pp. 105–121, 2008.

[34] Y. Ding, K. Hu, K. Wu, and Z. Zhao, "Performance monitoring and analysis of task-based OpenMP," *PLOS ONE*, pp. 1–12, 2013.

[35] A. Drebes, A. Pop, K. Heydemann, A. Cohen, and N. Drach, "Aftermath: A graphical tool for performance analysis and debugging of fine-grained task-parallel programs and run-time systems," in *Seventh Workshop on Programmability Issues for Heterogeneous Multicores (MULTIPROG-2014)*, Jan 2014.

[36] Y. Oyama, K. Taura, and A. Yonezawa, "Online computation of critical paths for multithreaded languages," in *Proceedings of the 15 IPDPS 2000 Workshops on Parallel and Distributed Processing*, ser. IPDPS, 2000, pp. 301–313.

[37] B. P. Miller, M. Clark, J. Hollingsworth, S. Kierstead, S. S. Lim, and T. Torzewski, "IPS-2: The second generation of a parallel program measurement system," *IEEE Transactions on Parallel and Distributed Systems*, vol. 1, no. 2, pp. 206–217, Apr. 1990.

[38] J. K. Hollingsworth and B. P. Miller, "Slack: A new performance metric for parallel programs," University of Wisconsin-Madison, Tech. Rep., 1994.

[39] F. Schmitt, J. Stolle, and R. Dietrich, "CASITA: A tool for identifying critical optimization targets in distributed heterogeneous applications," in *2014 43rd International Conference on Parallel Processing Workshops*, 2014, pp. 186–195.

[40] C. Alexander, D. Reese, and J. C. Harden, "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*, ser. MASCOTS '94, 1994, pp. 308–317.

[41] C. Q. Yang and B. P. Miller, "Critical path analysis for the execution of parallel and distributed programs," in *[1988] Proceedings. The 8th International Conference on Distributed*, 1988, pp. 366–373.

[42] A. Calotoiu, T. Hoefler, M. Poke, and F. Wolf, "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*, ser. SC '13, 2013, pp. 45:1–45:12.

[43] P. Reisert, A. Calotoiu, S. Shudler, and F. Wolf, "Following the blind seer – creating better performance models using less information," in *Euro-Par 2017: Parallel Processing*, 2017.

[44] M. Schulz, "Extracting critical path graphs from mpi applications," in *2005 IEEE International Conference on Cluster Computing*, 2005, pp. 1–10.

[45] X. Liu and B. Wu, "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*, ser. SC, 2015, pp. 47:1–47:12.

[46] T. B. Schardl, B. C. Kuszmaul, I.-T. A. Lee, W. M. Leiserson, and C. E. Leiserson, "The Cilkprof scalability profiler," in *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA, 2015, pp. 89–100.

[47] Y. He, C. E. Leiserson, and W. M. Leiserson, "The Cilkview scalability analyzer," in *Proceedings of the Twenty-second Annual ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA, 2010, pp. 145–156.

[48] D. Jeon, S. Garcia, C. Louie, and M. B. Taylor, "Kismet: Parallel speedup estimates for serial programs," in *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, ser. OOPSLA, 2011, pp. 519–536.

[49] A. Muddukrishna, P. A. Jonsson, A. Podobas, and M. Brorsson, "Grain graphs: OpenMP performance analysis made easy," in *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '16, 2016, pp. 28:1–28:13.

[50] M. Kim, P. Kumar, H. Kim, and B. Brett, "Predicting potential speedup of serial code via lightweight profiling and emulations with memory performance model," in *2012 IEEE 26th International Parallel and Distributed Processing Symposium*, 2012, pp. 1318–1329.

[51] V. S. Adve and M. K. Vernon, "Parallel program performance prediction using deterministic task graph analysis," *ACM Trans. Comput. Syst.*, vol. 22, no. 1, pp. 94–136, 2004.

[52] C. Coarfa, J. Mellor-Crummey, N. Froyd, and Y. Dotsenko, "Scalability analysis of SPMD codes using expectations," in *Proceedings of the 21st Annual International Conference on Supercomputing*, ser. ICS '07, 2007, pp. 13–22.

[53] S. L. Olivier, B. R. de Supinski, M. Schulz, and J. F. Prins, "Characterizing and mitigating work time inflation in task parallel programs," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '12, 2012, pp. 65:1–65:12.

[54] M. Schulz and B. R. de Supinski, "Practical differential profiling," in *Euro-Par 2007 Parallel Processing: 13th International Euro-Par Conference, Rennes ,France , August 28-31, 2007. Proceedings*, 2007, pp. 97–106.

[55] V. Subotic, J. C. Sancho, J. Labarta, and M. Valero, "A simulation framework to automatically analyze the communication-computation overlap in scientific applications," in *Proceedings of the 2010 IEEE International Conference on Cluster Computing*, ser. CLUSTER '10, 2010, pp. 275–283.

[56] C. Rosas, J. Giménez, and J. Labarta, "Scalability prediction for fundamental performance factors," *Supercomput. Front. Innov.: Int. J.*, vol. 1, no. 2, pp. 4–19, 2014.

[57] C. Curtsinger and E. D. Berger, "Coz: Finding code that counts with causal profiling," in *Proceedings of the 25th Symposium on Operating Systems Principles*, ser. SOSP, 2015, pp. 184–197.

## APPENDIX

### A. Abstract

This artifact description contains the information to access the source code for OMP-WHIP. The artifact includes the sources and the applications for which we could increase the speedup as described in the evaluation section of the paper. It also includes instructions to build OMP-WHIP, build various applications with various configurations, and generate the results as described in the paper.

### B. Description

*1) Check-list (artifact meta information):*

- **Algorithm**: The paper describes the algorithm to construct the novel OpenMP Series Parallel Graph (OSPG) and to generate parallelism and what-if profiles in the offline analysis.
- **Program:** C++ source code, C and C++ libraries
- **Compilation:** llvm clang++ 5.0 with C++11 flag. -g flag for case studies
- **Binary:** C++ and OpenMP programs
- **Data set:** Included in the applications
- **Run-time environment:** Ubuntu 16.04 LTS with llvm 5.0 compiled with a modified llvm OpenMP runtime. We also require `perf events` module in Linux.
- **Hardware:** Any Intel CPU with perf access to Hardware performance counters (Intel Xeon Gold 6130 @ 2.10GHz as tested)
  the
- **Execution:** Command line execution as described in the respective README files.
- **Output:** A parallelism profile and what-if profile in csv format
- **Experiment workflow:** Compile modified OpenMP runtime. Compile profiler to generate profiler library. Compile provided applications with the profiler. Execute the program. If on-the-fly mode is chosen, the profiles are generated after the program completes execution. In the offline-mode, run the offline analyzer provided with the profiler to produce parallelism profiles and what-if profiles.
- **Publicly available?:** Yes. It is publicly available on GitHub [22].

*2) How software can be obtained:* The source code of the profiler, offline analyzer, and modifications to the OpenMP runtime are available on GitHub at this URL: https://github.com/rutgers-apl/omp-whip.

*3) Hardware dependencies:* To reproduce the program profiles reported in the evaluation, an Intel CPU with hardware performance counters is necessary. All experiments were performed on a 16-core Intel Xeon Gold 6130 @ 2.10GHz machine. The tool only works on native hardware. To check if the machine supports hardware performance counters, use the command,

```
$ dmesg | grep PMU
```

If the output is "Performance Events: Unsupported...", then the machine does not support performance counters and OMP-WHIP cannot be executed on the machine.

*4) Software dependencies:* There are three main software dependencies.

- Linux Ubuntu 16.04 LTS distribution with `perf events` module installed.
- LLVM+Clang-5.0 compiled with the provided OpenMP runtime with OMPT support.
- Common packages such as Git, Cmake, and Python2.7.

*5) Datasets:* The inputs to various applications to reproduce the results are either included in the artifact or can be generated by the programs provided in the artifact.

### C. Installation

Clone the OMP-WHIP source code from https://github.com/rutgers-apl/omp-whip. We provide two bash scripts to automate the installation of OMP-WHIP and all the dependencies. To build OMP-WHIP and its dependencies run,

```
$ source setup-llvm.sh
$ source setup-ompwhip.sh
```

A successful build will compile LLVM/Clang, create the static libraries for OMP-WHIP and also setup the appropriate environment variables. Note, the bash script has to be sourced at the command-line. The installation will fail if it is run as an executable (i.e, with ./). The README file in the base directory of the package also provides detailed instructions on installation for both the on-the-fly mode and the offline-mode. To use the on-the-fly profiler mode, run source setup-ompwhip-online.sh.

### D. Experiment workflow

To download the applications for which we could improve speedups, use the link provided in README.md. Once downloaded use,

```
$tar -xvf applications.tar
```

There is a README file in each application directory providing information to reproduce our results. Here are the steps to check performance results: (1) build the baseline serial version following the instructions in the README of that application, (2) build the parallel version with our optimizations, and (3) follow the instructions in the README to execute them and measure speedup.

To profile an application, the general strategy is described below:

- Follow the instructions in the README to obtain inputs. Each application either has a pre-packaged input or it is generated through an auxiliary program.
- Build the application with the profiler. README provides detailed instructions. A simple make command will build the application with the profiler linked in for all applications.
- Execute the application with the inputs as specified.
- Run the offline analyzer using the instructions in the README. Follow the instructions in the README to understand the output of the profiler. The offline analysis step can be skipped if the on-the-fly profiling mode is selected.
- Either add annotations to the source or use the already packaged application with annotations and use the profiler to perform what-if analyses. Rerun the offline analyzer to see the results of what-if analyses.

### E. Evaluation and expected result

Each application has a README in its directory that provides detailed instructions on how to run the application, profile it, and the expected results. To automate the build process, execution, generation of speedup information, and omp-whip profiles for each application, we provide two python scripts in the directory of each application. The script generate_results.py generates results for offline profiling mode. Similarly, the script generate_results_online.py generates results for on-the-fly profiling mode.