PERFORMANCE PROFILERS AND DEBUGGING TOOLS FOR OPENMP APPLICATIONS

by

NADER BOUSHEHRINEJAD MORADI

A dissertation submitted to the

School of Graduate Studies

Rutgers, The State University of New Jersey

In partial fulfillment of the requirements

For the degree of

Doctor of Philosophy

Graduate Program in Computer Science

Written under the direction of

Santosh Nagarakatte

and approved by

____

____

____

____

New Brunswick, New Jersey

January, 2021

ABSTRACT OF THE DISSERTATION

Performance Profilers and Debugging Tools for OpenMP Applications

By Nader Boushehrinejad Moradi

Dissertation Director:

Santosh Nagarakatte

OpenMP is a popular application programming interface (API) used to write shared-memory parallel programs. It supports a wide range of parallel constructs to express different types of parallelism, including fork-join and task-based parallelism. Using OpenMP, developers can incrementally parallelize a program by adding parallelism to it until their performance goals are met.

In this dissertation, we address the problem of assisting developers in meeting the two primary goals of writing parallel programs in OpenMP: performance and correctness. First, writing OpenMP programs that achieve scalable performance is challenging. An OpenMP program that achieves reasonable speedup on a low core count system may not achieve scalable speedup when ran on a system with a larger number of cores. Traditional profilers report program regions where significant serial work is performed. In a parallel program, optimizing such regions may not improve performance since it may not improve its parallelism. To address this problem, we introduce OMP-ADVISER, a parallelism-centric performance analysis tool for OpenMP programs with what-if analyses. We propose a novel OpenMP series-parallel graph (OSPG) that precisely captures the series-parallel relations between different fragments of the program's execution. The OSPG, along with fine-grained measurements, constitute OMP-ADVISER's

performance model. OMP-ADVISER identifies serialization bottlenecks by measuring inherent parallelism in the program and its OpenMP constructs. OMP-ADVISER's what-if analysis assists developers to identify regions that have to be optimized first for the program to achieve scalable speedup. While a lack of inherent parallelism is a sufficient condition for a program not to have scalable speedup, too much parallelism can lead to excessive runtime and scheduling overheads, resulting in lower performance. We address this issue by extending OMP-ADVISER to measure tasking overheads. By attributing this additional information to different OpenMP constructs in the program, OMP-ADVISER can identify tasking cut-offs that achieve the right balance between a program's parallelism and its scheduling overheads for a given input. Further, we design a differential analysis technique that enables OMP-ADVISER to identify program regions experiencing scalability bottlenecks caused by secondary effects of execution.

Second, writing correct parallel programs is challenging due to the possibility of bugs such as deadlocks, livelocks, and data races that do not manifest when writing a serial program. A data race occurs when two parallel fragments of the program access the same memory location while one of the accesses is a write. Data races are common in OpenMP applications. Due to the exponential number of possible instruction and thread interleaving in parallel applications, identifying and reproducing data races using manual or automated testing techniques is impractical. Hence, specialized data race detection tools have been proposed to identify data races. We propose OMP-RACER, a novel apparent data race detector for OpenMP that supports a large call of OpenMP applications. To detect data races, OMP-RACER constructs the program's OSPG to encode the logical series-parallel relation for different serial fragments of the program. By capturing the program's logical series-parallel relations, OMP-RACER can detect data races in possible thread interleavings of the program for a given input. Thus, alleviating the need for schedule exploration. Compared to other OpenMP data race detectors, OMP-RACER can correctly identify races in a larger subset of OpenMP that use task-dependencies and locks, while achieving similar overheads.

Our results with testing the OMP-ADVISER and OMP-RACER prototypes with 40 OpenMP applications and benchmarks indicate their respective effectiveness in performance analysis and data race detection. Furthermore, it demonstrates the usefulness of our proposed OSPG data structure in enabling the creation of different types of analysis tools for OpenMP applications.

Acknowledgements

I would like to extend my deepest gratitude to my Advisor, Santosh Nagarakatte, for his constant guidance and support. Santosh's enthusiasm for solving interesting research problems has been a source of inspiration to me throughout my years of working with him. Thanks to his mentorship and support both academically and personally, I was able to grow as a researcher. This dissertation would not have been possible without the support and nurturing of Santosh.

I would like to extend my sincere thanks to my dissertation committee members, Badri Nath, Srinivas Narayana Ganapathy, and Martha Kim. Their insight and feedback helped improve this dissertation.

In my early years of doing a Ph.D., I had the great pleasure of working with my advisor Liviu Iftode, who is no longer with us. I will treasure his insight, knowledge, and wit for the rest of my life. I would also like to extend my gratitude to Vinod Ganapathy for his valuable insight and feedback. I also wish to thank Thu Nguyen for his support during some hard times in my Ph.D. studies.

I would like to thank my lab-mates, Adarsh Yoga, Jay Lim, Mohammedreza Soltaniyeh, Sangeeta Chowdhary, and David Menendez, at the RAPL research group for their technical and motivational support. Special thanks to my collaborator Adarsh Yoga. Throughout my Ph.D. studies, I also had the pleasure of interacting with many brilliant Ph.D. Students. I extend my thanks to Ruilin, Daehan, Hai, Daeyoung, Lu, Mohan, Liu, Amruta, Rezwana, Shakeel, Babak, Soheil, and Guilherme.

Many thanks to my amazing friends, who bring joy and laughter to my life. Special thanks to Yayun, Alireza, Touraj, Hooman, Farhad, Omid, Navid, Soroosh, and Ali for their invaluable friendship and support. Lastly, I cannot begin to express my thanks to my Family. I would not be where I am without the constant support, encouragement, and love from my parents Jamshid and Vilma, and sister Leila.

Dedication

In memory of my grandfather, Mohammad Javad

Table of Contents

Abstract ii

Acknowledgements iv

Dedication v

List of Figures 1

List of Tables 10

1. Introduction 11

1.1. Performance Analysis Tools for Parallel Programs 13

1.2. Race Conditions in Parallel Programs 14

1.3. Modeling Parallel Program Execution Using Series-Parallel Graphs 16

1.4. Thesis Statement 17

1.5. Research Contributions 17

1.6. Modeling the Execution of OpenMP Programs Using the OSPG 17

1.7. Performance Analysis of OpenMP Programs With OMP-ADVISER 19

1.8. Apparent Data Race Detection Using OMP-RACER 23

1.9. Contributions to This Dissertation 24

1.10. Organization of This Dissertation 25

2. OpenMP Series-Parallel Graph 26

2.1. OpenMP 26

2.1.1. Introduction to OpenMP 27

2.2. OSPG Overview 29

2.3. OSPG Definition 31

2.4. OSPG Construction 33

2.4.1. Program Start 34

2.4.2. Parallel Construct 35

2.4.3. Barrier Construct 36

2.4.4. Critical Construct 38

2.4.5. Worksharing Constructs 38

Single Construct 38

Sections Construct 39

For Construct 41

2.4.6. Tasking Constructs 42

Task Construct 43

Taskgroup Construct 45

Taskwait Construct 46

Task Dependencies 50

2.5. Identifying Logical Series-Parallel Relations 52

2.6. Summary 55

3. Performance Profiling with What-if Analysis using OMP-ADVISER 56

3.1. Motivation 57

3.2. Overview of Profiling With OMP-ADVISER 62

3.3. Performance Model Construction 65

3.3.1. OSPG Construction During Program Execution 66

Profiler Start/End 67

Parallel Construct 70

Barrier Construct 72

Worksharing Constructs 74

Tasking Constructs 76

3.3.2. Example OSPG Construction 80

3.4. Fine-grained Measurements 83

3.5. Estimating Runtime Parallelization Costs 86

3.6. Parallelism Profile Computation 88
3.6.1. Offline Analysis to Compute Parallelism 90
Generating the Program's Parallelism Profile. 98
3.6.2. On-the-fly Profiling Mode to Compute Parallelism 100
Online Analysis in the Presence of ST-nodes 101
Online Analysis in the absence of ST-nodes 104
Algorithm Analysis 110
3.7. What-if Analysis 111
3.7.1. User-defined What-if Analysis 114
3.7.2. Automatic What-if Analysis 117
3.8. Differential Analysis with OMP-ADVISER 120
3.8.1. Differential Analysis Approach 122
3.9. Evaluation 126
3.9.1. Effectiveness of OMP-ADVISER in Identifying Scalability Bottlenecks 128
Benchmarks with Serialization Bottlenecks 128
Benchmarks with High Tasking Overheads 134
3.9.2. Effectiveness of OMP-ADVISER's Automatic What-if Analysis 137
3.9.3. Effectiveness of OMP-ADVISER's Differential Analysis 140
3.9.4. OMP-ADVISER Performance Overhead 141
3.9.5. Comparison With Other Performance Analysis Tools 143
3.10. Limitations 147
3.11. Summary 148

4. On-the-fly Apparent Data Race Detection with OMP-RACER 149
4.1. Background 149
4.1.1. Characterizing Data Races 151
4.1.2. Static Data Race Detectors 153
4.1.3. Dynamic Data Race Detectors 153
4.1.4. Prior Dynamic Data Race Detectors for OpenMP Applications. 154

4.2. Overview of OMP-RACER . . . 155
4.2.1. Illustrative Example . . . 156
4.3. Data Race Detection Algorithm . . . 159
4.3.1. Data Race Detection in Fast Mode . . . 159
OSPG Construction in Fast Mode . . . 160
Computing Series-Parallel Relations in Fast Mode . . . 162
Access History Management in Fast Mode . . . 163
Data Race Detection Algorithm in Fast Mode . . . 164
4.3.2. Data Race Detection in Precise Mode . . . 168
Computing Series-Parallel Relations in Precise Mode . . . 168
Access History Management in Precise Mode . . . 168
Data Race Detection Algorithm in Precise Mode . . . 172
4.3.3. Illustrative Example . . . 176
4.4. Evaluation . . . 178
4.4.1. Prototype Implementation . . . 179
4.4.2. Evaluation Setup . . . 184
4.4.3. OMP-RACER Evaluation Results . . . 186
4.5. Summary . . . 190
5. Related Work . . . 192
5.1. Series-Parallel Graphs . . . 192
5.2. Performance Analysis Tools . . . 195
5.3. Data Race Detection Tools . . . 202
6. Conclusion . . . 205
6.1. Summary of Contributions . . . 205
6.2. Future Work . . . 207
6.3. Concluding Thoughts . . . 208

List of Figures

1.1. (a) Code snippet illustrating 3 threads that write to shared variables x,y, and z. (b) Different interleavings an a multithreaded program may lead to different final values for shared variables. Number of interleavings grows exponentially with thread counts and the operations within each thread. The example illustrates two possible traces that produce different results 14

1.2. (a) A simple task-parallel program creating three tasks that logically execute in parallel. (b) In task-parallel frameworks, the runtime dynamically schedules tasks by mapping them to available threads. This figure depicts two possible schedules on a machine with two threads. 18

2.1. A simple C program to sort the elements of an array. (a) The sequential program. (b) The program parallelized with OpenMP. 28

2.2. (a) A simple OpenMP program using the `parallel` directive, which creates two worker threads that execute the code block at lines 5-7 in parallel. (b) The program's OSPG where the W-nodes represent the program's fragments. (c) The program's trace, illustrating the series-parallel relations between program fragments. 30

2.3. Step by step OSPG construction for the program in Figure 2.2. (a) OSPG at the beginning of the program's execution. (b) The OSPG when the program encounters the `parallel` construct at line 3. (c) The OSPG when the program exits the `parallel` block at line 7 Figure 2.2. 34

2.4. (a) A simple OpenMP program with an explicit OpenMP `barrier` construct. (b) OSPG Construction before and after the execution encounters the `barrier` construct at line 7. 36

2.5. (a) A simple OpenMP program using the `single` construct. (b) Two possible OSPG constructions before and after the program enters the `single` construct. Both OSPGs encode the same series-parallel information between program fragments. 39

2.6. (a) OpenMP code snippet with a `sections` construct. (b) OSPG construction before entering the `sections` construct at line 6 and after exiting it at line 14. 40

2.7. OSPG construction for the OpenMP `for` construct. (a) Example OpenMP code snippet with a `for` loop construct. (b) The program's OSPG construction before entering the loop at line 6 and after all iterations of the loop complete at line 9. 41

2.8. OSPG construction for a simple task-based OpenMP code snippet. (a) OpenMP code snippet with `task` construct and no task-specific synchronization. (b) OSPG construction when the encountering thread in the `single` construct reaches different lines in the `single` construct. 43

2.9. OSPG construction for a task-based OpenMP code snippet that uses `taskgroup` for synchronization. (a) OpenMP code snippet. (b) OSPG construction before the encountering thread reaches the `taskgroup` directive at line 5, upon reaching the `taskgroup` directive at line 7, and after exiting the `taskgroup` block at line 24. 44

2.10. (a) OpenMP program snippet with perfectly nested taskwaits. (b) Part (a)'s code snippet's OSPG. (c) OpenMP program snippet that does not have perfectly nested taskwaits. (d) An incorrect attempt to construct part (c)'s OSPG in fast mode. (e) Correct OSPG construction for the code snippet in part (c) using ST-nodes. 45

2.11. OpenMP program snippet with a task nesting level of 3, Illustrating different variations of taskwait usage and their corresponding OSPG, which correctly captures the series-parallel relations between program fragments. 49

2.12. (a) OpenMP program snippet with task-dependencies in a program with perfectly nested taskwaits. (b) The code snippet's OSPG. Each W-node is named after its corresponding serial function in the program. 51

2.13. (a) OpenMP program snippet with task-dependencies in a program with unrestricted usage of taskwaits. (b) The code snippet's OSPG. Each W-node is named after its corresponding serial function in the program. 52

3.1. An example parallel application and using its parallelism profile to identify which program fragment to optimize first and the impact of first round of optimization (a) Illustration of a parallel program's fragments and the ordering of the fragment's execution. (b) Parallelism profile of the program. (c) A possible trace of the program when executed on a 4 core machine. (d) The program's parallelism profile after parallelizing fragment `iv` by a factor of 4. (e) The optimized program's trace when executed on a 4 core machine. 58

3.2. (a) An example OpenMP program. (b) Performance model of the example program in (a). (c) OMP-ADVISER's parallelism profile of the program. (d) OMP-ADVISER's automatic what-if analysis, identifying which regions to parallelize and the resulting what-if parallelism profile. 63

3.3. An example OpenMP program (a) Task-based OpenMP program. (b) The program's OSPG. Each W-node in the OSPG is named after the corresponding serial function in the program. 80

3.4. Incremental OSPG construction for the example program in Figure 3.3. 84

3.5. Use of performance counters to measure the program's computation and attribution of work to corresponding W-node during OSPG construction. 86

3.6. Execution of the PROCESSNODE when computing the work, serial work, and critical path on the OSPG subtree from Figure 3.2. The node currently being processed is highlighted by bold text in the table. The child node being examined is shown with dashed circles in the Figure. 97

3.7. (a) An OpenMP program snippet that uses tasking and taskwaits. (b) The program's performance model. the table under each W-node contains the corresponding line in the program and its amount of work. the `st_val` of nodes is shown with a hexagon each to relevant nodes. 102

3.8. Execution of the on-the-fly profiling algorithm (shown in Algorithm 13) to compute the parallelism profile of an example OSPG. Nodes that haven't been created or have completed execution are greyed out. Nodes being popped from the OSPG are marked with a dashed circle. 109

3.9. A data-parallel for loop executed serially and in parallel. (a) Serial program with data-parallel loop (b) The serial program's trace. (c) Performance model of the serial program. (d) Parallelelized version of the program (e) The parallel program's trace. (f) Performance model of the parallelized program. 113

3.10. (a) A data-parallel for loop performing a simple map-reduce computation. (b) the serial for loop execution trace. (c) An OpenMP conversion of the serial application. The program has work inflation caused by false sharing on variable `p_sum`. The execution trace of the parallel for loop (lines 6-10 in Figure 3.10). Because of false sharing, the parallel version performs more work compared to the serial program. 120

3.11. (a) Performance model of the single thread execution of the program in Figure 3.10(c). (b) Performance model of the parallel execution of the program in Figure 3.10(c) with four worker threads. 123

3.12. Parallelism profile for AMGmk. (a) parallelism profile on the original program. (b) what-if profile after manually annotating selected regions. (c) The parallelism profile of the program after concretely parallelizing the regions selected using what-if analysis. 129

3.13. Parallelism profile for Quicksilver. (a) parallelism profile on the original program. (b) what-if profile after manually annotating selected regions. (c) The parallelism profile of the program after concretely parallelizing the regions selected using what-if analysis. 130

3.14. Parallelism profile for Delaunay Triangulation. (a) parallelism profile on the original program. (b) what-if profile after manually annotating selected regions. (c) The parallelism profile of the program after concretely parallelizing the regions selected using what-if analysis. 131

3.15. Parallelism profile for Minimum Spanning Forest. (a) parallelism profile on the original program. (b) what-if profile after manually annotating selected regions. (c) The parallelism profile of the program after concretely parallelizing the regions selected using what-if analysis. 132

3.16. Parallelism profile for NBody. (a) parallelism profile on the original program. (b) what-if profile after manually annotating selected regions. (c) The parallelism profile of the program after concretely parallelizing the regions selected using what-if analysis. 133

3.17. Parallelism profile for Convex Hull. (a) parallelism profile on the original program. (b) what-if profile after manually annotating selected regions. (c) The parallelism profile of the program after concretely parallelizing the regions selected using what-if analysis. 133

3.18. Parallelism profile with tasking overhead measurements for KDTree. (a) parallelism profile of the original program. (b) The parallelism profile of the program after reducing its tasking overheads. 135

3.19. Parallelism profile with tasking overhead measurements for BOTS knapsack. (a) parallelism profile on the original. (b) The parallelism profile of the program after reducing its tasking overheads. 136

3.20. Parallelism profile and differential analysis for Lulesh. (a) parallelism profile of the original program (b) Differential analysis result. 141

4.1. (a) Code snippet illustrating multithreaded program with a data race on shared variable `x`. (b) Two possible execution traces on a system with two threads. . . 150

4.2. (a) Code snippet illustrating multithreaded program with two apparent data races on shared variables `x` and `f1`. (b) Two possible execution traces of the code snippet in (a), illustrating that the data race on variable `f1` is feasible. 152

4.3. (a) OpenMP program snippet with a write-read data race on shared variables `x` (b) Possible execution trace of the code snippet in (a), when executing on a 2 threaded system. (c) The code snippet's OSPG. The code fragment each W-node represents is shown below it. 159

4.4. Illustration of an example OpenMP program snippet where the use of constant amount of access history metadata is not sufficient to detect the data race on variable x. (a) OpenMP program snippet. (b) The code snippet's OSPG. (c) Program trace with fast mode's access history metadata management. (d) Program trace with precise mode's access history metadata management. 169

4.5. (a) Example OpenMP program with a write-read apparent data race on variable psum[1] at lines 18 and 23. (b) The execution trace of the example program when executed with two threads. The first column specifies the ordering of the observed trace. The second column specifies the memory access as a 4-tuple comprised of thread id, memory access type, memory access location, and the W-node performing the access. The third column illustrates the access history maintained by OMP-RACER for the corresponding memory access as a 4-tuple with the node identifier, W-node corresponding to the latest write, and two W-nodes corresponding to two parallel read accesses (- denotes the empty set). (c) The program's OSPG. The code fragment each W-node represents is shown below it. The square boxes next to some OSPG nodes represent the value indicating the nesting depth in the presence of taskwaits. 177

4.6. Performance slowdown over parallel execution of OMP-RACER and Archer with our performance overhead benchmarks. 187

List of Algorithms

1. The IsParallel function returns true if a pair of W-nodes, W_l and W_r logically execute in parallel. The DirectedPath function returns true if there is a path comprised of dependency edges between the input P-nodes. The Count(W_l, L) function returns the sum of the nesting depth values (*i.e.*, `st_val`) on the path from W_l to L. 54

2. OSPG construction at the start of the OMP-Adviser profiler execution. The algorithm initializes the $nodes$ per-thread stack and creates the OSPG root node, S_1 and its W-node child, W_1. 68

3. OSPG construction at the end of the profiler execution. The algorithm simply pops the last remaining active OSPG nodes in the main thread of execution. . . . 68

4. OSPG construction when a parallel construct is encountered. The construction algorithm relies on two callbacks to update the per-thread view of the program's OSPG. 69

5. OSPG construction when exiting a parallel region. The construction algorithm relies on two callbacks to update the per-thread view of the program's OSPG. . . 71

6. OSPG construction when the program trace encounters a barrier construct. The algorithm uses two callbacks to distinguish when a worker threads enters a barrier and when it exits a barrier construct. In `BarrierExit`, $first_tid$ denotes the worker thread that first exits the barrier construct. 73

7. OSPG construction when the program encounters task creation constructs. The construction algorithm relies on different callbacks to both the parent and child tasks' view of the program's OSPG. 76

8. OSPG construction when the program encounters different task synchronization constructs. 78

9. Generate program's parallelism profile for OSPG G rooted at node R. D is the set of the program's directives used for aggregation. CHILDNODES returns all the child nodes of the input node. COMPUTESERIALWORK identifies the serial work contribution of each static directive to the program's critical path. AGGREGATEPERSTATICDIRECTIVE aggregates work, serial work, and tasking overheads to produce the parallelism profile similar in format to Figure 3.2 (c) and (d). 91

10. Compute input node N's work ($N.w$) and and the set of W-nodes on its critical path ($N.S$). 93

11. Compute the critical path contribution of each static location in the program . . . 100

12. Algorithm to compute an internal OSPG node's critical path form its child nodes 105

13. On-the-fly profiling mode algorithm for program's with no ST-nodes. The function PUSHNODE occurs when adding a new OSPG node N as the child node of P. The function POPNODE is called after the entire subtree under node N completes execution and node N updates its parent node's attributes before getting deallocated. 106

14. OSPG construction when a user-defined what-if annotation is encountered. The annotation includes its location in the program source. 115

15. Compute input node N's work ($N.w$) and and the set of W-nodes on its critical path ($N.S$) in what-if profiling mode. 116

16. Automatically generate a list of program regions that, if parallelized, the program's parallelism becomes greater or equal to the target parallelism. Conversely, the algorithm may terminate early without being able to produce such a list. . . . 119

17. Analysis algorithm checking if using fast mode is safe for an OpenMP program. This analysis requires OMP-RACER to run the program once before running its data race detection algorithm using fast or precise mode. 160

18. The ISPARALLEL relation when using OMP-RACER's fast mode for a pair of W-nodes where W_l is left of W_r. 162

19. OMP-RACER data race detection algorithm in fast mode. The function is invoked whenever the program accesses a memory location. 165

20. OMP-RACER data race detection algorithm in precise mode. 174

List of Tables

3.1. List of OpenMP benchmarks used for evaluating OMP-ADVISER 127

3.2. List of OpenMP benchmarks used for evaluating OMP-ADVISER 129

3.3. List of OpenMP applications with low initial parallelism used with OMP-ADVISER's automatic what-if analysis. 137

3.4. OMP-ADVISER' s performance overheads. The first two columns show the on-the-fly profiler's run-time overhead compared to the unmodified serial and OpenMP application. The third column compares the program's resident memory usage in profiling mode over parallel execution. 142

3.5. List of applications used with Intel Advisor to analyze. 144

3.6. Summary of Intel VTune's analysis report for different OpenMP applications. . 146

4.1. OMPT callbacks used by OMP-RACER prototype 180

4.2. List of applications used for evaluating OMP-RACER's performance overheads 184

4.3. Data race detection report on DataRaceBench 1.2.0 185

4.4. Performance overhead comparison between OMP-RACER's fast mode and precise mode 189

Chapter 1

Introduction

With core frequency scaling coming to an end in the mid-2000s, multi-core CPUs, once mainly used in the High-Performance Computing (HPC) domain, have become ubiquitous in all domains [156]. In recent years, to meet the ever-increasing demand for more computational power, heterogeneous computing using accelerators such as GPUs and FPGAs has become widely adopted. All these advances illustrate a trend of increasing hardware parallelism.

To effectively utilize the increasing amount of hardware parallelism, new software must be written to use the additional parallelism in hardware. Moreover, a program written to utilize the available parallelism in current hardware may not effectively scale to leverage the additional parallelism that becomes available in future devices—requiring a need for high-performing and scalable parallel programs. Generally, it is a requirement for all software to be correct and meet performance requirements. However, due to additional complexities that do not exist in serial programs, writing correct, high-performing, and scalable parallel software is known to be a challenging endeavor [131], requiring domain knowledge and years of developer expertise.

The most common form of parallel programming, thread-based programming, exacerbates the complexity of writing parallel software [113]. Thread-based programming is a low-level abstraction for parallel programming as it closely matches the way operating systems manage parallel execution. Two common issues with parallel programming using low-level threads exist. First, it is challenging to reason about program behavior due to many possible thread interleavings. This potentially leads to bugs that are hard to discover, reproduce, and fix. Second, thread-based abstraction does not decouple scheduling from the computation. Developers must choose the number of threads to create and how to orchestrate them to run their programs. For example, static scheduling in thread-based programming may lead to low program parallelism and work imbalance among threads, leading to sub-par performance.

To assist developers with writing parallel applications and address some shortcomings of thread-based programming, different high-level programming languages and frameworks such as OpenMP [153], Cilk [47], X10 [41], TBB [163], OpenACC [151], and CUDA [51] have become popular. By introducing higher-level abstractions and parallelism primitives compared to thread-based programming, these languages simplify the task of writing parallel applications. While these languages have successfully lowered the barrier of entry for parallel programming, many correctness and performance considerations remain even when these languages are used for parallel programming.

A substantial amount of research has been done to increase parallel programmers' productivity and assisting them to develop correct and scalable applications. One branch of this research has resulted in the design of performance analysis and verification tools for parallel applications. A performance analysis tool aims to identify bottlenecks sources in a program and assist programmers in locating root-causes for lack of performance and scalability. Earlier performance analyzers for sequential programs [76] have inspired the design of parallel program performance analysis tools. Hence, many performance analysis tools report regions where the program spends most of its time as sources of bottlenecks [14, 50, 71, 189]. While this approach is satisfactory for analyzing sequential programs, it is not sufficient for analyzing parallel programs. A parallel program's performance is determined by the longest sequence of serial regions in the program (i.e., Amdahl's law [18]), known as its critical path. For example, a region where the program spends most of its time may not lie on the program's critical path. Optimizing it first will not impact the program's performance. Our goal is to develop algorithms and design tools that help parallel programmers with writing scalable programs.

In addition to bugs common with sequential programs, some bugs such as deadlocks, livelocks, and data races are specific to parallel programs. Identifying and reproducing these bugs is difficult, mainly due to a large number of possible thread interleavings in a parallel program, which can grow exponentially with the number of hardware threads and instructions per thread. Researchers have proposed different solutions to automate the identification of these types of bugs [9, 66, 69, 88, 114, 117, 141, 168, 175].

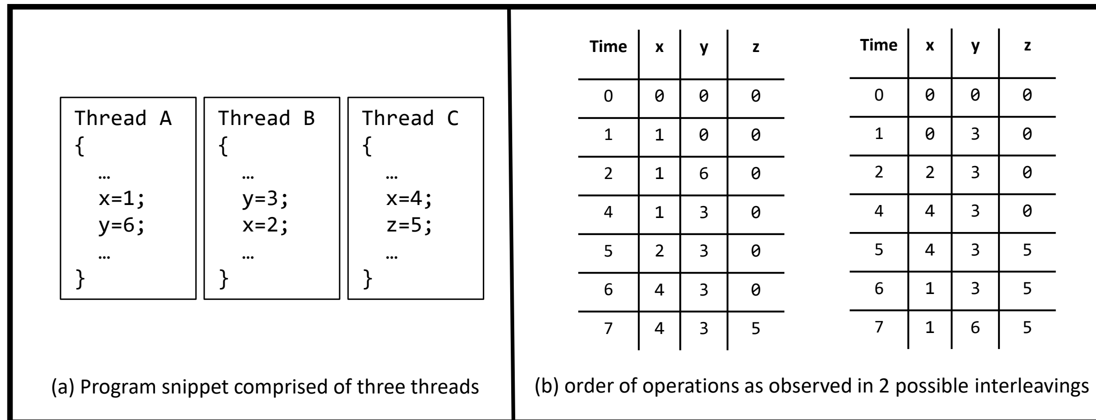
1.1 Performance Analysis Tools for Parallel Programs

By measuring the computation and memory usage of programs, performance analysis tools are used to identify its performance bottlenecks. Understanding the behavior and performance issues in parallel applications is challenging. It is desired that a performance analysis tool assists developers by accurately identifying the sources of bottlenecks. Moreover, it should guide its user to address performance bottlenecks by pinpointing the cause and providing actionable recommendations to guide the developer in fixing performance issues.

For a sequential application, the program's execution can be thought of as a chain of function calls. For a given input, this chain forms a total order. Thus, to diagnose performance bottlenecks, it is sufficient to identify function calls where the program performs the most computation since optimizing them will provide the highest performance gains. Hence, profiling tools for sequential programs tend to follow the same principle, in which they measure resource utilization and aggregate the measurements at different granularities. For example, a call path profiler [83] aggregates its measurements to each function call in the program.

The design of some parallel application performance profilers is inspired by their serial counterparts. Many of these tools employ a similar strategy used by sequential program profilers, where each thread's resource utilization is measured. The profiler provides a thread-based view of thread utilization, and it reports which function in the program utilizes the most resources (*i.e.*, a hotspot) [14, 50, 189]. While these profilers are useful in detecting certain performance issues in parallel applications, they fail to identify other bottlenecks. By adapting the serial program profiling techniques to a parallel setting, these profilers do not accurately model a parallel program's execution.

Compared to a serial program, the execution of a parallel program is more complicated. At any point in execution, multiple threads could run different sequential operations. We can model the parallel program's execution with a graph. Each node represents sequential operations, and each edge captures the ordering between operations. The longest path on this graph represents the longest sequence of execution in the program, known as the critical path. The critical path provides a theoretical upper bound on the program's execution time (e.g., Amdahl's law). Hence, reducing a program's critical path is a necessary condition to improve



Time	x	y	z
0	0	0	0
1	1	0	0
2	1	6	0
4	1	3	0
5	2	3	0
6	4	3	0
7	4	3	5

Time	x	y	z
0	0	0	0
1	0	3	0
2	2	3	0
4	4	3	0
5	4	3	5
6	1	3	5
7	1	6	5

Figure 1.1: (a) Code snippet illustrating 3 threads that write to shared variables x,y, and z. (b) Different interleavings an a multithreaded program may lead to different final values for shared variables. Number of interleavings grows exponentially with thread counts and the operations within each thread. The example illustrates two possible traces that produce different results

its performance. Identifying the critical path and aiming to reduce it has been used by some profiling tools [16, 36, 155] to discover bottlenecks in parallel programs.

However, prior profiling tools tend to have two limitations. First, some techniques identify a trace specific critical path [172], which may not be the same as the program's critical path as specified by Amdahl's law. Thus, optimizing a trace specific critical path may only improve the application's performance on the current system. In contrast, the program's performance may not improve when executed on a system with higher hardware parallelism. Second, while some systems aim to identify the critical path[1] as specified by Amdahl's law, they are limited to analyzing applications with structured parallelism [161, 193]. Our goal is to design performance analysis tools that improve upon prior solutions by targeting a larger class of parallel applications while measuring the program's critical path. Achieving this goal requires new ways to model the execution of a parallel application.

1.2 Race Conditions in Parallel Programs

A parallel program's execution may have many possible interleavings[2] for a given input. The number of program interleavings grows exponentially with the number of hardware threads and

[1] Also referred to as the program's span.

[2] Also referred to as a schedule.

instructions per thread. If a program produces different outputs for a given input depending on the program's schedule, the program is nondeterministic. Nondeterminism is the root cause of many bugs in parallel programs, including deadlocks, livelocks, and race conditions. Figure 1.1 illustrates an example of nondeterminism caused by different thread interleavings in a multithreaded application.

Because of the large number of possible interleavings in a parallel program, attempts to manually identify and reproduce these types of bugs are difficult and error-prone. Further, testing techniques have limited efficacy due to limited coverage of all possible schedules. Hence, to effectively identify these bugs, specialized tools are used. Data races commonly occur in parallel programs and often lead to undefined behavior. The problem of automated data race detection is well-studied. Some proposed solutions detect races using static analysis [9, 66, 140, 158], and some use dynamic analysis techniques [69, 132, 168, 175].

According to Netzer and Miller's [146] classification, a data race may be apparent or feasible. A feasible data race is one in which it manifests in an actual program trace. To detect a feasible data race, we should consider the program's computation, synchronization, and parallelism constructs. In contrast, an apparent data race approximates the program's feasible data races by primarily considering its synchronization and parallel constructs without taking the actual computation into account. While not every apparent race is not feasible, in some classes of programs know as Abelian [44], every apparent race is also a feasible data race. A data race detector may aim to detect either feasible or apparent data races. Detecting apparent data races is considered to be easier since it alleviates the need for schedule exploration.

Data race detection tools can be broadly classified into two categories, static and dynamic. Static data race detectors use static analysis techniques to identify races in the program. While this approach has the benefit of low overheads and can be used to detect races for all program inputs, its main drawback is higher rates of false positives. Dynamic data race detectors aim to detect data races by executing the program for a given input. Compared to a static data race detector, a dynamic data race detector reports a lower false positives rate.

The main drawback of dynamic data race detection is that they may report data races for only a given input and a specific thread interleaving in the program. As discussed previously, the number of interleaving in a program may increase exponentially with the number of threads.

Thus, limiting the tool's ability to identify data races for all schedules. Some tools address this issue by detecting data races that occur across all schedules using data structures that capture logical series-parallel relations. One limitation of these tools is that they target programs with a specific form of structured parallelism [68, 132, 160, 194]. To perform data race detection in a larger class of parallel programs across all schedules, we need to design new data structures that can accurately model the logical series-parallel relations specified by their parallelism constructs.

1.3 Modeling Parallel Program Execution Using Series-Parallel Graphs

To design algorithms and tools that can effectively analyze parallel applications, the algorithm must accurately capture the parallel application's execution. One approach to model a parallel program's execution utilizes data structures to capture the logical series-parallel relations between sequential fragments in its dynamic execution trace. Two fragments logically execute in parallel if based on the parallel program's semantics, they may execute in parallel in some possible program schedule for a given input. The logical series-parallel relation between two fragments is a property of the program for a given input. Intuitively, two logically parallel fragments will execute in parallel when the program is executed on a hypothetical system with infinite processors. Capturing the logical series-parallel relations in a parallel program enables the design of analysis tools that can reason the application across different thread interleavings for a given input. Capturing the logical series-parallel relation is complicated for general thread-based programs. Hence, prior research has focused on modeling structured parallelism, including applications that either use fork-join [132] or task-based parallelism [68, 161].

More recently, a line of research aims to capture logical series-parallel relations in a larger class of parallel applications. For example, R-Sketch [15] and MultiBags+ [187] propose data structures and algorithms to capture logical-series parallel relations in programs with futures. The use of unstructured parallelism constructs such as futures is becoming more popular in parallel programming languages since simplify expressing certain parallel algorithm. To design analysis tools for this class of programs, we need to develop new data structures and algorithms that can accurately encode logical series-parallel relations in the presence of unstructured parallelism.

1.4 Thesis Statement

Using logical series-parallel graphs to model the execution of parallel programs in OpenMP, we can build practical performance analysis and debugging tools that assist developers in writing correct and scalable programs.

1.5 Research Contributions

This dissertation explores the design of new series-parallel graphs to capture the execution of parallel applications written in OpenMP. Using this series-parallel graph, we design a new profiling tool to identify performance bottlenecks and develop a tool to identify a common correctness issue in OpenMP programs, data races. Our contributions are summarized as follows:

1. The OpenMP Series-Parallel Graph (OSPG), a data structure that captures the logical series-parallel relations between serial fragments of dynamic execution in an OpenMP program. The OSPG is constructed incrementally and in parallel during program execution. It correctly models OpenMP programs with both worksharing and tasking constructs.

2. The performance analysis tool and adviser, OMP-ADVISER, uses the OSPG to construct a parallelism profiler for OpenMP applications. Using what-if and differential analyses, OMP-ADVISER identifies performance bottlenecks and recommends which regions to optimize to improve the program's performance.

3. The apparent data race detector, OMP-RACER, utilizes the OSPG to design a data race detection algorithm that alleviates the need for repeated executions and interleaving exploration for a given input.

The rest of this section provides a brief description of the research contributions listed above and highlights the high-level ideas used to design them.

1.6 Modeling the Execution of OpenMP Programs Using the OSPG

In this dissertation, we capture the execution of parallel applications written in OpenMP by proposing a new data structure called the OpenMP Series-Parallel Graph (OSPG). The OSPG

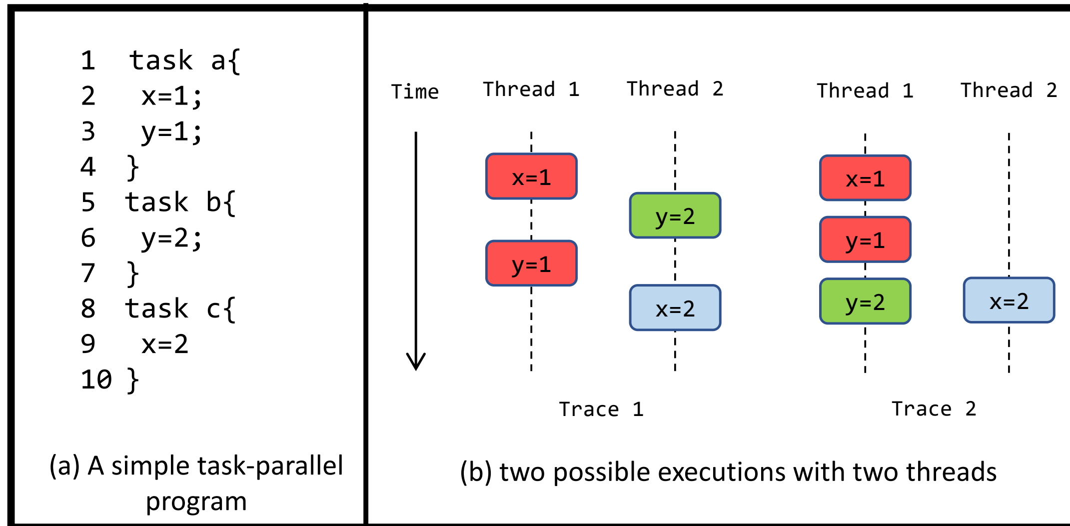


Figure 1.2: (a) A simple task-parallel program creating three tasks that logically execute in parallel. (b) In task-parallel frameworks, the runtime dynamically schedules tasks by mapping them to available threads. This figure depicts two possible schedules on a machine with two threads.

encodes the logical series-parallel relations between different fragments in the program execution trace. OpenMP [153] is a popular programming API that supports a wide range of parallelism constructs. The high-level parallelism constructs in OpenMP support fork-join, task-based, SIMD, and heterogeneous computing in the form of offloading. Further, OpenMP supports incremental parallelism, where the developer adds parallelism to their application by using OpenMP's parallelism constructs as a set of compiler directives. OpenMP has been used for decades in the HPC domain to write shared-memory parallel applications. More recently, it has been used to speedup parallel applications in other domains [154].

While prior research has proposed data structures to capture the logical series-parallel relations in parallel programming frameworks, these works mainly either support fork-join or task-based parallelism constructs [27, 86, 132, 162, 169, 193]. Whereas the OSPG supports the OpenMP API, thus modeling both forms of parallelism constructs. The OSPG is an ordered directed acyclic graph (DAG) that captures the dynamic execution of an OpenMP program as a set of fragments. A fragment is the longest sequence of instructions between two OpenMP parallelism constructs; hence a fragment is executed serially by a thread. The OSPG captures the logical series-parallel relations between any pair of fragments in an OpenMP program by using different types of nodes and edges.

Instead of capturing a specific schedule of the program's execution, the OSPG captures the logical series-parallel relations between program fragments. For example, a pair of logically parallel fragments might execute serially by the same thread on a given schedule. Meanwhile, the program's semantics may indicate that the pair of fragments may execute in parallel. The example in Figure 1.2 illustrates this point where two logically parallel tasks b and c, are executed in series in a possible execution trace on a system with two threads. Intuitively, the OSPG captures the series-parallel relations of the program when running with an infinite number of processors using an idealized runtime. Hence, the program's logical-series parallel relations are a property of the program for a given input.

To use the OSPG as the basis for designing practical and scalable performance analysis and verification tools, we must ensure that its construction is lightweight and can be constructed in parallel. To realize this goal, we propose an incremental construction algorithm that constructs the program's OSPG in parallel.

1.7 Performance Analysis of OpenMP Programs With OMP-ADVISER

The parallelism constructs in OpenMP are easy to use and enable application developers to add parallelism to their programs incrementally. However, the program may still not meet the developer's performance goals. Further, even when a program exhibits good speedups on a low number of hardware threads, it may not have scalable speedup when executed on a system with a higher core count. Our goal with OMP-ADVISER is to design performance analysis techniques for OpenMP programs to assist developers with building performance portable applications.

As described earlier, Amdahl's law provides a theoretical guarantee on improving the program's performance by quantifying and reducing its critical path, the longest sequence of sequential computation. Thus, to quantify a program's scalability, one can use the program's logical parallelism. Logical parallelism is the ratio of the program's serial work over its critical path[3]. It represents the program's speedup when executed on an ideal system with an infinite number of processors with no scheduling or runtime overheads. While logical parallelism is a metric that characterizes performance for an idealized system, our crucial observation was

[3] Another terminology used for defining logical parallelism is the ratio of program's work over its span.

that it is an effective metric in assessing the performance portability of an OpenMP application since a program with sufficient logical parallelism will have scalable speedup when running on a system with a higher amount of hardware parallelism. Hence, OMP-ADVISER measures the logical parallelism of the program and each static OpenMP directive in the program.

To measure the logical parallelism, we construct a performance model of the application under test. The performance model is comprised of two components. First, we need a mechanism to capture the logical series-parallel relations for each program fragment. Second, we need a mechanism to measure the work performed in each program fragment. By using these two components, OMP-ADVISER can compute the logical parallelism in the program using its parallelism computation algorithm. Chapter 3 provides detail on how OMP-ADVISER calculates parallelism using these components.

The OSPG encodes the logical series-parallel relations in an OpenMP program. Thus, we use it to realize the first component in OMP-ADVISER's performance model. As described in Section 1.6, the OSPG data structure is a graph that captures each program fragment and encodes their logical series-parallel relations with different types of nodes and edges. When analyzing a program, OMP-ADVISER constructs its OSPG on-the-fly for a given input.

To measure the work performed by each program fragment, the longest sequence of instructions in the dynamic execution before encountering an OpenMP construct, OMP-ADVISER leverages the availability of hardware performance counters in modern processors whenever possible. These hardware counters provide the means to perform fine-grained measurements of different metrics (*e.g.*, instruction count, cycle count, *etc.*) with low runtime overheads. OMP-ADVISER measures the amount of computation and attributes them to the program's OSPG nodes.

When using OMP-ADVISER as a parallelism profiler, the user runs the application under test with a given input. The profiler executes the program, constructs its OSPG in parallel, and uses hardware performance counters to perform fine-grained work measurements. Using its parallelism profiling algorithm OMP-ADVISER generates a report that includes the logical parallelism and contribution to the program's critical path for the entire program and each static OpenMP directive in the program. This profile can be used to identify serialization bottlenecks and OpenMP directives in the program that exhibits low parallelism.

While OpenMP directives with low parallelism are potentially good candidates to optimize to improve the program's performance, optimizing them may not always increase the program's parallelism. This can occur when the directive is not on the critical path, and increasing the directive's parallelism will not shorten the critical path. Alternatively, when the program has multiple paths in the program's execution graph that perform similar amounts of work compared to the critical path. In this case, optimizing the directive will shift the program's critical path but not necessarily reduce it. Leading us to the question, can we use our performance model to identify regions that improve program parallelism?

We demonstrate that OMP-ADVISER's performance model can be used to estimate the improvements in program parallelism if a selected program region is hypothetically parallelized. We call this type of analysis, what-if analysis. What-if analysis helps developers prioritize which regions to optimize first to gain the most improvement in program parallelism. OMP-ADVISER supports user-defined what-if analysis queries where the developer annotated regions of code to perform what-if analysis. While users found what-if analysis useful in identifying program bottlenecks, it sometimes led to many trial and error runs to identify the regions to optimize. We noticed that this manual process could be automated. Hence, for a given target parallelism value, OMP-ADVISER can automatically perform what-if analysis queries to identify a list of regions that must be optimized to reach the desired parallelism. Alternatively, it reports that it is infeasible to achieve the target parallelism. We refer to this technique as automatic region identification and describe it in detail in Chapter 3.

While increasing a program's parallelism is necessary to improve its speedup, it is not sufficient to guarantee a program is performance portable. There are many reasons that hinder a program from being performance portable, leading to the difficulty associated with writing scalable parallel applications. Hence, to improve the efficacy of OMP-ADVISER as a performance analysis tool, we need to augment its performance model to capture additional sources of performance bottlenecks. More specifically, in its current design, OMP-ADVISER identifies two additional causes of bottlenecks.

First, in the real world, the OpenMP runtime has to use some resources to orchestrate the parallelism expressed in the program. In practice, for a program to be performance portable, it needs to perform sufficient parallel computation to amortize the runtime costs associated with

creating and managing parallelism constructs. To account for this phenomenon, OMP-ADVISER measures the runtime costs associated with creating a parallelism construct in the OpenMP program and identifies if the program performs sufficient computation for each construct to amortize associated runtime costs. Further, OMP-ADVISER uses runtime costs measurements when performing automatic region identification, which identifies the limit a program can be parallelized before runtime costs dominate.

Another common source of bottlenecks in parallel applications is resource contention, which inhibits the application's scalability. While many parallel programs use shared resources such as locks and other synchronization primitives to perform correctly, not all contention is necessary or intended by the application developer. Hence, it is desirable for a performance analysis tool to identify and report such sources of contention. An application with resource contention may exhibit secondary effects of execution. Directly identifying resource contention in an application is known to be difficult.

The key insight used by OMP-ADVISER is that when a parallel program suffers resource contention, one will typically observe work inflation in some metric of interest [13]. A program has work inflation if its operations cost more compared to an oracle execution that does not suffer from the resource contention in the parallel execution. Hence OMP-ADVISER uses differential analysis [130] to identify OpenMP program regions with work inflation and reports them to the user. Using differential analysis, OMP-ADVISER runs the program twice, once with an oracle execution and once in parallel, and then compares the OSPG of the two runs to identify regions with work inflation. A key insight that enables OMP-ADVISER's differential analysis is that the two constructed OSPGs can be compared directly, which results from the OSPG capturing the logical series-parallel relations which are independent of the program's thread interleaving or the number of hardware threads used to execute the program. Chapter 3 describes the details of OMP-ADVISER's approach to performing differential analysis.

Using OMP-ADVISER, we have analyzed over 40 OpenMP applications from differential benchmark suites and HPC applications. OMP-ADVISER has proven useful in identifying performance bottlenecks and providing advice on which regions to optimize to gain speedup.

1.8 Apparent Data Race Detection Using OMP-RACER

Similar to other multithreaded applications, an OpenMP program may have data races. Data races may cause nondeterminism, undefined program behavior, and often are the cause of bugs in parallel applications [29]. A data race occurs when a shared memory location is accessed by two or more threads in parallel when one of the accesses is a write operation. For example, the program in Figure 1.2 has two data races over variables `x` and `y`, leading to nondeterministic final values.

Our work with OMP-RACER answers the question, "can the OSPG be used to design debugging tools for OpenMP applications?" by using it in the context of data race detection. With OMP-RACER, our goal is to develop an apparent dynamic data race detector for OpenMP applications that can detect data races that occur for different program schedules for a given input—hence alleviating the need for schedule exploration. For example, a per schedule data race detector may miss the data race on variable `y` in Figure 1.2(a) if the captured execution matches trace 2 Figure 1.2(b).

An interesting problem that we faced when designing OMP-RACER was that since the OpenMP API is comprised of fork-join, tasking, and some forms of unstructured parallelism constructs, our original OSPG design needed enhancements for use in data race detection. We updated our OSPG design to precisely encode the semantics of unstructured OpenMP constructs (*e.g.*, taskwaits and task dependencies).

To detect data races on-the-fly, in addition to constructing the program's OSPG, a data race detection algorithm must include a mechanism to maintain previous memory accesses to a given memory location [44]. A naive solution is to maintain all prior accesses to each memory location in the program as part of the access history metadata. While correct, this approach is impractical since the access history grows in proportion to the program memory trace and incurs large memory overheads.

We devise two modes to manage access histories with OMP-RACER: a fast and a precise mode. The fast mode maintains a constant amount of metadata per shared memory location. This mode has low runtime overheads, and it is comparable to other state-of-the-art OpenMP data race detectors in terms of performance. However, the fast mode is limited to OpenMP

programs that use fully nested taskwaits (*i.e.*, one of OpenMP's synchronization constructs) and do not use locks. When the fast mode is not suitable for the OpenMP program under test, OMP-RACER reverts to using its precise mode. Instead of a constant amount of metadata per shared memory location, OMP-RACER's precise mode metadata grows in proportion to the task nesting level and the program's lockset size. Compared to fast mode, OMP-RACER's precise mode has higher runtime and memory overheads.

With these two mechanisms, OMP-RACER detects races as follows. For a given program input, OMP-RACER constructs the program's OSPG. Whenever a new memory access occurs, OMP-RACER retrieves the access history metadata associated with that memory location. Next, by querying the OSPG, it checks whether the current access is involved with any access from the retrieved metadata, reporting any data race detection. Further, it updates the access history to include the current memory access.

Our experience with using OMP-RACER with popular OpenMP data race detection benchmarks and applications indicates that OMP-RACER is useful in detecting apparent data races in different OpenMP applications. We describe the details of OMP-RACER data race detection algorithm in Chapter 4.

1.9 Contributions to This Dissertation

The ideas and techniques described in this dissertation are drawn from the following research papers written in collaboration with my advisor Santosh Nagarakatte and my collaborator Adarsh Yoga.

1. "A Parallelism Profiler with What-If Analyses for OpenMP Programs" [31] introduces OMP-WHIP, which uses the proposed OSPG data structure to construct a performance model that enables parallelism profiling and what-if analyses.

2. "On-the-fly Data Race Detection with the Enhanced OpenMP Series-Parallel Graph" [33], which introduces the EOSPG, an extension of the OSPG, and describes a data race detection algorithm to detect apparent races in OpenMP applications.

1.10 Organization of This Dissertation

The rest of this dissertation is organized as follows. In Chapter 2, we provide a brief primer on OpenMP and introduce the OpenMP Series-Parallel Graph (OSPG). The OSPG data structure captures the logical series-parallel relations between the serial fragments of an OpenMP program. The OSPG enables the creation of the performance analysis and debugging tools described in later chapters in the dissertation. Chapter 3 describes our performance profiling and advising tool, OMP-ADVISER. Based on the OSPG, OMP-ADVISER can identify code regions that matter for performance by providing a parallelism centric profiling technique with what-if analyses. We describe our performance model's details and elaborate on automatic region identification and differential analysis techniques to find performance bottlenecks. Chapter 4 provides the details of our apparent data race detector and evaluation using our prototype, OMP-RACER. Chapter 5 reviews related work. We conclude in Chapter 6, providing a summary of this dissertation and reflect on potential ideas for expanding upon the work described in this dissertation.

Chapter 2

OpenMP Series-Parallel Graph

To design tools for analyzing parallel programs written in OpenMP, we need data structures and algorithms to precisely encode the semantics of the OpenMP API. In this chapter, we introduce the OpenMP Series-Parallel Graph (OSPG), a data structure that precisely captures the logical series-parallel relations in OpenMP programs that use a variety of the API's fork-join and task-based constructs. The OSPG is a key component in the design of our performance analysis and data race detection tool described in Chapters 3 and 4. A program's OSPG captures the logical series-parallel relations between all fragments in the program for a given input. A fragment is the longest sequence of instructions in the dynamic execution without encountering an OpenMP parallelism construct. For a given program input, the OSPG captures the series-parallel relations induced by the semantics of the OpenMP API instead of a given execution trace. Two fragments logically execute in parallel if based on the parallel program's semantics, they may execute in parallel in some possible program schedule for a given input. The logical series-parallel relation between two fragments is a property of the program for a given input. Intuitively, the OSPG captures the series-parallel relation between program fragments that occur on an idealized system with an infinite number of processors, which we refer to as the logical series-parallel relation.

2.1 OpenMP

The OpenMP[1] API consists of a set of compiler directives and library functions to specify different types of parallelism constructs. OpenMP was introduced by the OpenMP ARB[2], a

[1] Open Multi-Processing.

[2] Architecture Review Board.

group of members from academia and industry, in 1997. Since then, its set of features have grown with each version[3].

OpenMP is widely supported and is one of the most used parallelism APIs for shared-memory multiprocessor systems. Currently, OpenMP supports the following languages: C, C++, and Fortran. Most compilers, including GCC, Clang, and Icc, support OpenMP. For example, the latest version of LLVM's OpenMP implementation fully supports OpenMP's 3.1 standard and partially supports features from version 4.0, 4.5, and 5.0.

One of the reasons for the wide adoption of OpenMP is its ease of use. Using a variety of parallelism constructs provided as compiler directives, the programmer can incrementally add parallelism to their application until their performance goals are met. Initially, OpenMP supported only SPMD (single program, multiple data) parallelism. Later versions of OpenMP have added worksharing, task-based, and SIMD parallelism. Thus, many serial applications can be parallelized with the expressive set of constructs defined in the OpenMP API.

2.1.1 Introduction to OpenMP

The OpenMP API has grown in complexity with each new version. We now provide an overview of the OpenMP programming model and a cursory discussion of the common OpenMP parallelism constructs. We elaborate on other supported OpenMP constructs in later sections of this chapter. A complete description of the OpenMP API can be accessed from the latest version of its specification [153]. At its core, the OpenMP API uses the fork-join model of parallel execution [153]. An OpenMP program begins with a single thread of execution, named the initial thread. By using the OpenMP parallel construct, the programmer can fork a team of worker threads, including the initial thread. The team of worker threads executes the parallel region specified by the `parallel` construct. The thread that creates the team is called the master thread. At the end of the parallel construct, the execution is joined by an implicit barrier. After the parallel construct, only the master thread continues execution. The programmer can incrementally add more parallel constructs to their program as they parallelize different parts of

[3]Currently at version 5.0 [153].

```
1  #define SIZE 1000
2  int main(){
3    int arr[SIZE];
4    for(int i=0; i < SIZE; ++i){
5      arr[i] = read_elem(i);
6    }
7    size_t pivot = partition(arr);
8    sort(arr, 0, pivot);
9    sort(arr, pivot, SIZE);
10   return 0;
11 }
```

(a) Sequential program

```
1  #define SIZE 1000
2  int main(){
3    int arr[SIZE];
4    #pragma omp parallel
5    {
6      #pragma omp for
7      for(int i=0; i < SIZE; ++i)
8      {
9        arr[i] = read_elem(i);
10     }
11     #pragma omp single
12     {
13       size_t pivot = partition(arr);
14       #pragma omp task
15         sort(arr, 0, pivot);
16       #pragma omp task
17         sort(arr, pivot, SIZE);
18     }
19   }
20   return 0;
21 }
```

(b) Parallelized program with OpenMP

Figure 2.1: A simple C program to sort the elements of an array. (a) The sequential program. (b) The program parallelized with OpenMP.

their applications. Hence, OpenMP supports adding incremental parallelism to the application until the developer's performance goals are met.

By default, each worker thread executes the same code within a parallel region. However, this results in redundant computation in the application and no performance gains. Thus, OpenMP supports a variety of parallelism constructs used to specify different forms of parallel computation within a parallel region. For example, worksharing constructs distribute program computation between the team's worker threads. The worksharing `for` construct distributes the iterations of a bounded loop between the team's threads. The `for` construct is used to implement data-parallel computations. The worksharing `single` construct is used to specify a code block that only one of the team's threads will execute within a parallel region.

Additionally, OpenMP supports task-based parallelism with the `task` construct. A task instance is comprised of a block of code and its associated data environment. Once a task construct is encountered by one of the worker threads during execution, a task instance is created. The OpenMP runtime schedules a task instance for execution by one of the worker threads. OpenMP supports nested tasks. In OpenMP, different task synchronization and dependency constructs enable support for some forms of irregular parallelism.

In C and C++, OpenMP parallelism constructs are specified as compiler directives. Upon compilation, OpenMP directives get replaced by calls to the OpenMP runtime that orchestrates the parallelism specified by the application developer. Consider a simple C program that sorts an array using a divide-and-conquer algorithm such as Quicksort. Figure 2.1(a) shows a sequential implementation. The for loop at line 4 initializes the array. Lines 7-9 implement the Quicksort algorithm by partitioning the array and recursively sorting the left and right subarrays. Figure 2.1(b) shows the same program being parallelized using OpenMP's parallelism constructs. In this sample implementation, the data-parallel for loop is parallelized using the OpenMP `for` construct. Further, OpenMP tasking constructs are used to parallelize the divide-and-conquer sorting algorithm by specifying two tasks to sort the left and right subarray, respectively.

2.2 OSPG Overview

The OpenMP Series-Parallel Graph (OSPG) is an ordered directed acyclic graph (DAG) that captures the execution of an OpenMP program as a set of fragments and their logical series-parallel relations. Our goal with designing the OSPG is a data structure that can precisely encode the logical series-parallel relations in a large class of OpenMP programs. We use the OSPG as part of the following tools. OMP-ADVISER, our proposed performance analysis tool where the OSPG is used as part of its performance model to measure logical parallelism. OMP-RACER, our proposed apparent data race detector for OpenMP applications where the OSPG is used to encode the logical series-parallel relations between sequential fragments in the program to check for possible data races.

A program fragment is the longest sequence of instructions without encountering an OpenMP parallelism construct. By design, each program fragment is modeled by a leaf node in the OSPG. The OSPG's intermediate nodes and edges encode the logical series-parallel relations between the leaf nodes. A program's OSPG captures the logical series-parallel relations between any pair of fragments for a given input. Given any pair of fragments, their series-parallel relations can be retrieved by performing a least common ancestor (LCA) query between the leaf nodes corresponding to the pair of fragments in the program's OSPG. A program's OSPG has different types of nodes. The leaf nodes, which represent program fragments, are called W-nodes. Further,

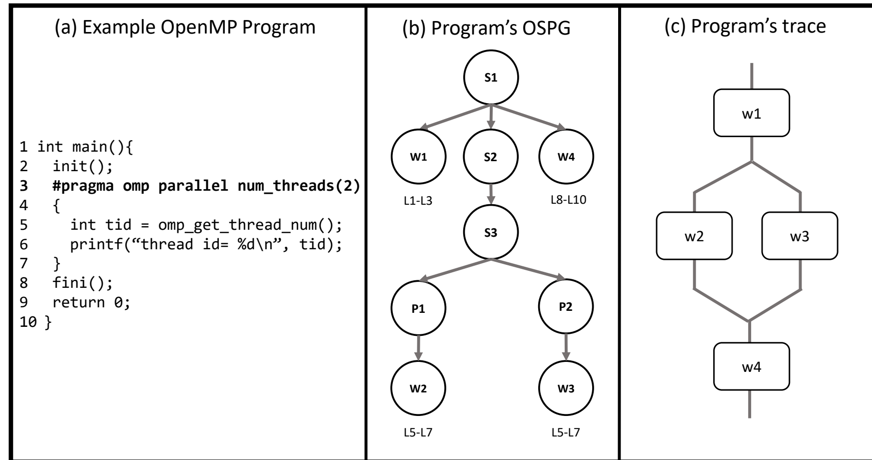


Figure 2.2: (a) A simple OpenMP program using the `parallel` directive, which creates two worker threads that execute the code block at lines 5-7 in parallel. (b) The program's OSPG where the W-nodes represent the program's fragments. (c) The program's trace, illustrating the series-parallel relations between program fragments.

the OSPG has three types of internal nodes, P-nodes, S-nodes, and ST-nodes. Internal nodes and OSPG edges encode the logical series-parallel relations between any W-node pair in the program's OSPG.

To demonstrate these ideas, we start with the OpenMP program illustrated in Figure 2.2(a). This simple C program uses an OpenMP `parallel` construct that creates a team of worker threads that execute the code block at lines 4-7 in parallel. The program starts sequentially with the main thread of execution. The main thread executes the serial function `init()` at line 2 before encountering the `parallel` construct at line 3. The `num_threads()` clause specifies the number of worker threads that the OpenMP runtime creates. Thus, in this example, two worker threads are created. Each worker thread executes lines 5-6 in parallel. Each worker thread uses the OpenMP runtime library function `omp_get_thread_num()` to get its thread number within the team[4] and prints it afterward. As specified by the OpenMP standard, a `parallel` construct ends with an implicit `barrier` where the main thread of execution resumes execution once both worker threads finish executing the `parallel` block. After the `barrier`, the main thread executes the serial function `fini()` before the program returns and completes execution.

[4]With a team of two threads, the function returns 0 and 1 respectively for each worker thread.

This program has two OpenMP parallelism constructs, the `parallel` construct at line 3 and the implicit `barrier` at line 7. Based on the definition of a fragment, the longest sequence of instructions without encountering an OpenMP parallelism construct, the program has four fragments. Figure 2.2(b) depicts the OSPG of the example program where each fragment is captured by a leaf node labeled $W1 - W4$. The remaining nodes in the OSPG capture the series-parallel relations between any pair of fragments in the program. Figure 2.2(c) shows the series-parallel relations between the fragments by depicting a program trace. From the trace, we observe that fragments $W2$ and $W3$, which represent the execution within the `parallel` block, execute in parallel.

Further, both fragments $W2$ and $W3$ execute in series relative to $W4$, which represents the main thread's execution trace after exiting the implicit OpenMP `barrier` at the end of the parallel block. All of these relations are encoded in the program's OSPG. For this simple program, internal S-nodes and P-nodes encode the relations between all fragments. An S-node and a P-node encode a series and a parallel relation, respectively. In this example, a pair of W-nodes execute in parallel if their LCA node's left child is a P-node. Otherwise, the pair of fragments execute in series.

In the presence of OpenMP tasking, this check is slightly more involved, which we describe in detail in Section 2.5. For example, consider the parallel pair of fragments $W2$ and $W3$ in Figure 2.2(b). The LCA node for the pair is the S-node $S3$. The left child of $S3$ is the P-node $P1$. Hence, the OSPG captures the parallel relation between the fragments $W2$ and $W3$. In contrast, the pair of fragments $W2$ and $W4$ execute in series. The program's OSPG encodes this information as follows. The LCA node for the pair of W-nodes, $W2$ and $W4$, is the node $S1$. The left child of $S1$ is the S-node $S2$, capturing the serial relation between the fragments that correspond to W-nodes $W2$ and $W4$.

2.3 OSPG Definition

The OSPG is an ordered directed acyclic graph (DAG), $G = (V, E)$, where the set V consists of four types of nodes[5], W-nodes, S-nodes, P-nodes, and ST-nodes. Thus, $V = V_w \cup V_p \cup V_s \cup V_{st}$.

[5] Also referred to as vertices.

The set of edges, $E = E_{pc} \cup E_{dep}$, where E_{pc} denotes the parent-child edges between nodes and E_{dep} denotes the dependency edges. Since the OSPG is a DAG, it has a topological ordering [6], and the node that appears first in the topological ordering has an in-degree of zero. We refer to this node as the OSPG's root node. By design, the root S-node has a unique directed path consisting of only E_{pc} edges to all other nodes in the OSPG. A node's depth is defined as the number of edges on the path consisting of E_{pc} edges from the root node to it. Two nodes with the same depth are referred to as sibling nodes. For a pair of sibling nodes V_i and V_j at depth d, node V_p is their parent node if it has depth $d - 1$ and the edges (V_p, V_i) and (V_p, V_j) connecting the parent node V_p to its children nodes V_i and V_j respectively. We label edges that establish a parent-child relationship as E_{pc}, and label edges between sibling nodes as E_{dp}. An edge labeled as E_{dp} indicates that the edge represents a dependency between sibling nodes. By design, other than the root node, all OSPG nodes have a unique parent node. OSPG node. In an OSPG, sibling nodes are ordered from left to right. The left to right ordering between nodes in the OSPG represents the logical sequencing of operations in the program. For a program's OSPG, a sibling node U is left of node V if U is visited before V in its depth-first execution order. Each node maintains additional information to encode the nesting depth of the program in the presence of certain synchronization constructs (*i.e.*, `taskwait`). We refer to this per node state as the node's `st_val`, which is an integer in $\{-1, 0, 1\}$. The `st_val` is used to precisely encode the logical series-parallel relation in the presence of OpenMP `taskwait` constructs.

W-node. A W-node represents a serial fragment of dynamic execution in the program. A program fragment[7] is the longest sequence of instructions without encountering an OpenMP parallelism construct. By construction, a W-node is always a leaf node in the OSPG. A fragment either starts from the beginning of the program or when the execution encounters an OpenMP construct. The fragment continues until the program ends, or it reaches another OpenMP construct. In the absence of any OpenMP directives in the program, the entire program is serially executed and is comprised of one fragment. Hence, the OSPG of a sequential program consists of a single W-node that is a direct child of the root S-node. A W-node has an `st_val` of zero.

[6] A linear ordering for the nodes of the graph such that if (u, v) is an edge of the graph then u appears before v in the ordering.

[7] Another common terminology for a fragment is a strand.

The set of W-nodes in a program's OSPG represent the entire computation performed in the program's execution.

P-node. A P-node encodes a logically parallel relation between W-nodes. The W-nodes in the subtree under a P-node execute in parallel with all W-node sibling and descendants to its right. For example, as illustrated in Figure 2.2(b), when a program encounters an OpenMP parallel construct, it creates a P-node for each worker thread in the team to represent the logical parallel execution between each worker thread fragment. Resulting in creating P-nodes $P1$ and $P2$ in Figure 2.2(b). Since a P-node adds to the program's nesting level, the st_val of a P-node is one.

S-nodes. Complementary to P-nodes, an S-node encodes serial execution between all W-nodes under its subtree and all W-node descendants of its right sibling nodes. For example, when a program's execution encounters OpenMP parallel construct, an S-node is created to represent the serial execution between the code block within the parallel construct and the code fragment that executes after it as illustrated in Figure 2.2(b) with the creation of S-node $S2$. The st_val of an S-node is zero.

ST-node. This node also encodes the logical series-parallel relations between W-nodes. Unlike S-nodes or P-nodes, the logical series-parallel relation between W-nodes under the subtree of an ST-node and its right siblings depend on the number of OpenMP taskwait constructs the program encounters and the program's nesting level. ST-nodes are used whenever S-nodes or P-nodes cannot correctly encode the logical series-parallel relations for W-nodes under their subtree (See Section 2.4.6). For an ST-node, the W-nodes under its subtree are partitioned into two subsets. First, W-nodes that execute serially with all right siblings, second, W-nodes that execute in parallel relative to their right siblings. Upon creation, an ST-node has an st_val of zero. Its st_val changes to -1 if the program encounters a taskwait construct.

2.4 OSPG Construction

This section describes the construction of the OSPG when encountering different OpenMP constructs in the dynamic execution. The OpenMP API, currently at version 5.0 [153], is comprised of different parallelism constructs that enable the developer to implement a wide

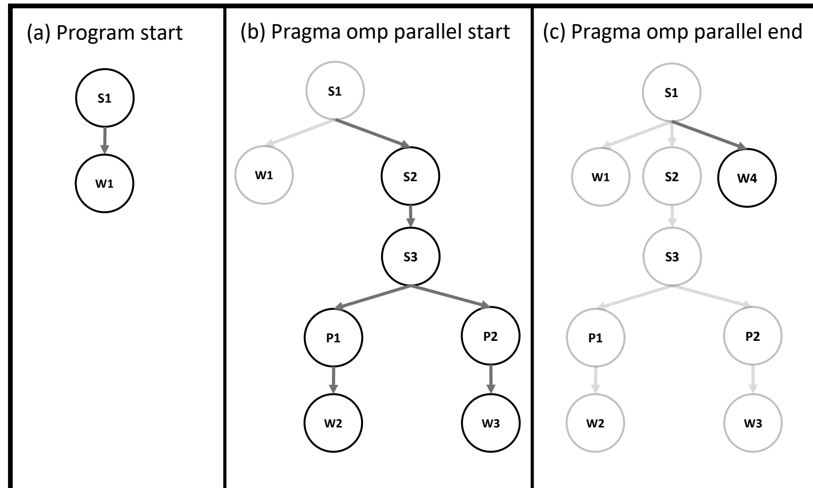


Figure 2.3: Step by step OSPG construction for the program in Figure 2.2. (a) OSPG at the beginning of the program's execution. (b) The OSPG when the program encounters the `parallel` construct at line 3. (c) The OSPG when the program exits the `parallel` block at line 7 Figure 2.2

range of parallel algorithms. Currently, our OSPG construction supports a subset of the OpenMP API called the common core plus some additional constructs that include features recently added to the API, including task dependencies. By supporting the common core of OpenMP, the OSPG can capture the execution of a large number of different OpenMP applications that use both worksharing and tasking constructs.

2.4.1 Program Start

At the beginning of the program trace, the program's OSPG starts with two nodes. It has a root S-node with a child W-node, as illustrated in Figure 2.3(a). The W-node represents the first fragment of execution in the program. For a sequential program, a singular W-node represents the entire program's execution. Thus, Figure 2.3(a) also illustrates the minimal OSPG that occurs for a sequential program with no OpenMP constructs.

During program execution, whenever the execution trace encounters an OpenMP construct, it incrementally adds new nodes and edges to capture the most recent program fragments and their series-parallel relations. For example, for the program in Figure 2.2(a), W-node $W1$ represents the program fragment starting from the beginning of the program's execution until the program encounters the OpenMP `parallel` construct at line 3. At the end of the program's execution,

the incrementally constructed OSPG matches the program's OSPG. For example, Figure 2.3 illustrates the incremental OSPG construction for the example program in Figure 2.2(a). Hence, the incremental OSPG construction at Figure 2.3(c) matches the program's OSPG illustrated in Figure 2.2(b).

2.4.2 Parallel Construct

The OpenMP `parallel` construct follows the fork-join model of parallel execution. The thread encountering a `parallel` construct creates a team of threads, including itself, referred to as the master thread. The team of worker threads execute the structured block specified by the `parallel` construct. The developer can specify the team size explicitly using the `num_threads` clause or let the OpenMP runtime choose the size of the team. Once all threads in the team complete their execution, they are synchronized with an implicit OpenMP `barrier` where only the master thread continues execution.

To correctly support the semantics of the `parallel` construct, the OSPG construction must capture four pieces of information. First, it should create W-nodes to capture the fragments of work originating at the beginning of the `parallel` construct and performed by each thread in the team. Second, it should capture the parallel relation between these W-nodes. Third, the OSPG should capture the serial relation between the `W-nodes` representing the execution within the `parallel` region and the code fragments that represent the program's continuation after exiting a `parallel` region. Finally, the OSPG must capture the serial relation between the first fragments in the team and followup fragments within the `parallel` region that may get created when the threads in the team encounter synchronization constructs such as barriers within the `parallel` construct.

To capture this information and incrementally construct the OSPG, the thread encountering the beginning of the `parallel` construct adds a new S-node to the OSPG. This S-node captures the serial relation between the work performed by the team of threads within the `parallel` region and the work performed by the master thread after exiting the `parallel` region. As shown in Figure 2.3(b), S-node $S2$ is added as the child of the root node when the program's execution trace reaches the beginning of the `parallel` region. From this point on, every computation within the `parallel` region will be captured in the subtree rooted at the S-node

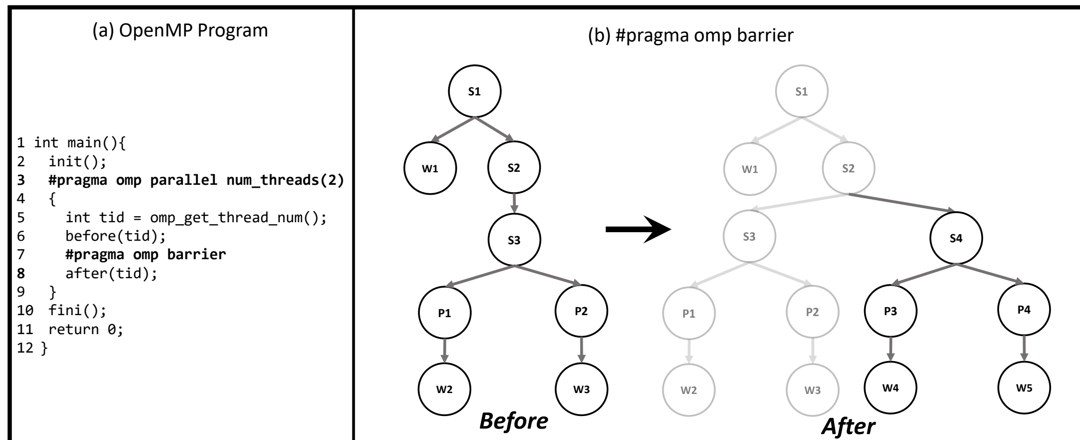


Figure 2.4: (a) A simple OpenMP program with an explicit OpenMP `barrier` construct. (b) OSPG Construction before and after the execution encounters the `barrier` construct at line 7.

$S2$. At this point in the construction, we do not know a priori if the program may encounter any additional barriers within the `parallel` region. We add the S-node $S3$ as the first child of $S2$ in anticipation of the first barrier in the `parallel` region. To represent the parallel computation performed by the team of threads starting from the beginning of the `parallel` construct, we add a P-node for every thread in the team as child nodes of $S3$. Since the example program in Figure 2.2(a) creates a team with two threads, the construction adds P-nodes $P1$ and $P2$ per each worker thread in the team. For each newly created P-node, we add a W-node child to represent the fragment of execution by each thread in the team. This construction precisely captures the series-parallel information induced by the `parallel` construct, resulting in the OSPG illustrated in Figure 2.3(b).

2.4.3 Barrier Construct

An OpenMP `barrier` synchronizes the code fragments before the `barrier` with the code that executes after it. Hence, all threads must execute the `barrier` before the program can continue executing the code after the `barrier`. An OpenMP `barrier` can be implicit, like the `barrier` at the end of a `parallel` region, or explicit, specified by the programmer using the OpenMP `barrier` directive. In either case, the OSPG must capture the following information. First, it has to capture the code fragments before and after the `barrier` as different W-nodes. Suppose W_{pre} is the set of W-nodes that represent the work performed by

a team of threads up to the point where the team encounters a `barrier`, and W_{cont} is the set of `W-nodes` that represent the work performed after the `barrier`. Thus when encountering a `barrier`, the OSPG construction algorithm must add new W-nodes to represent W_{cont}. Second, the OSPG has to capture the serial execution between any members of W_{pre} and W_{cont}. During the construction, members of W_{pre} must appear to the left of members W_{cont}. Capturing the serial relation between the fragments W_{pre} and W_{cont} is accomplished by an S-node. The S-node S_{bar} should be an ancestor of all members of W_{pre}, and all members of W_{cont} must occur to its right. Since the OSPG is incrementally constructed, this requirement implies that S_{bar} must be added to the OSPG before any members of W_{pre} are created. We address this issue by anticipating an upcoming `barrier` in the OSPG construction. In OpenMP, all OpenMP `barrier` constructs must have an enclosing `parallel` region since the execution outside a `parallel` region is sequential, and a `barrier` with no enclosing `parallel` region is treated as a no-op. Thus, the OSPG construction algorithm does not need to consider barriers outside a `parallel` construct. As discussed in subsection 2.4.2, upon entering a `parallel` region, the master thread adds an S-node in anticipation of the first upcoming barrier.

When constructing the OSPG of the OpenMP program in Figure 2.4(a), as shown in Figure 2.4(b), the S-node $S3$ is added to the program's OSPG when the program enters the parallel region in anticipation of the upcoming `barrier`. When the program encounters the first `barrier` at line 7, we add W-nodes $W4$ and $W5$ to represent the fragments of execution after the `barrier` construct. The S-node $S3$ correctly captures the serial execution between the fragments of execution before and after the `barrier`. Further, upon encountering the `barrier`, the S-node $S4$ is added in anticipation of any additional barriers later in the parallel region. To capture the parallel execution between the computation performed by the team of threads after the `barrier`, we add P-nodes for each worker thread, which in Figure 2.4(b) are labeled as $P3$ and $P4$. During the OSPG construction, we maintain the invariant of adding one S-node to the program's OSPG for each newly encountered barrier. Hence, only the first thread in the team entering the barrier creates a new S-node.

2.4.4 Critical Construct

The `critical` construct in OpenMP restricts the execution of an enclosing block of code to one thread at a time. This construct acts similar to a mutex, where the team of threads executing the structured block contends for a lock. The developer can specify a name for a `critical` construct similar to using different locks in a multithreaded program.

The order of threads entering a `critical` construct may vary across different thread interleavings for a given program and its input. As noted in section 2.3, the OSPG is designed to capture the logical series-parallel relations between program fragments. Thus, for an OpenMP `critical` construct, the OSPG does not capture schedule specific information. However, entering and exiting a critical construct creates new fragments. Hence, the OSPG creating three `W-nodes` to represent the fragments of execution before, within, and after the critical construct.

2.4.5 Worksharing Constructs

OpenMP has several worksharing constructs. The code specified within a worksharing construct is distributed among the worker threads within a `parallel` region. This section describes the details of OSPG construction when the program encounters an OpenMP worksharing construct. In this section, we first discuss OpenMP `single` and `sections` constructs. Then, we discuss how OSPG construction in the presence of worksharing loops.

An OpenMP worksharing construct must occur within a `parallel` construct. Otherwise, the worksharing construct is treated as a no-op since all the code specified by it will execute sequentially. Thus, when constructing the program's OSPG, there is no need to update the OSPG when a worksharing construct is encountered outside a `parallel` region. By default, an OpenMP worksharing construct has an implicit `barrier` at its end. This `barrier` can be omitted by using a `nowait` clause. The implicit barriers for a worksharing construct are captured by the OSPG construction procedure as described in Section 2.4.3.

Single Construct

The `single` construct is used to ensure that the block of code specified by it will execute exactly by one of the team's threads. Since the thread that gets to execute the `single` construct

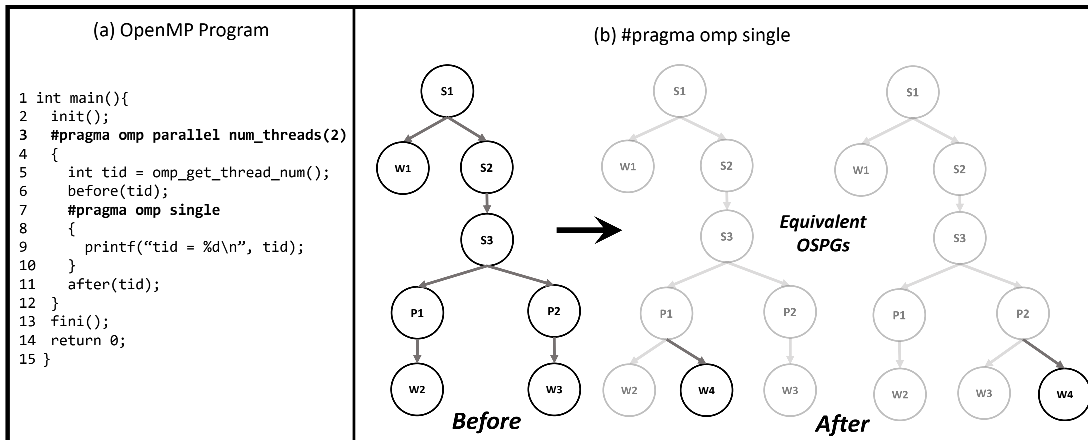


Figure 2.5: (a) A simple OpenMP program using the `single` construct. (b) Two possible OSPG constructions before and after the program enters the `single` construct. Both OSPGs encode the same series-parallel information between program fragments.

is among the threads in the team executing its enclosing `parallel` region, the constructed OSPG already captures the parallel relation between the selected thread and all other threads in the team, as described in Section 2.4.2. The OSPG captures the fragment of work starting at the beginning of the `single` construct with a W-node. Depending on which thread is assigned to execute the `single` construct, the newly created W-node corresponds to the P-node associated with the selected thread as illustrated in Figure 2.5(b) for the sample program in Figure 2.5(a). In this example, two possible OSPG construction exists for a `parallel` region with two worker threads. We consider these OSPGs equivalent since the series-parallel relation between any pair of W-nodes remains the same across all possible OSPG construction in the presence of a `single` construct.

Additionally, OpenMP supports the `master` construct. The `master` construct specifies that only the team's master thread must execute the specified code block. While the `master` construct is not classified as a worksharing construct, it can be considered as a specific case of the `single` construct. Thus, it follows a similar OSPG construction.

Sections Construct

When using the `sections` construct, the developer specifies a fixed number of structured code blocks using the `section` directive. These code blocks are distributed among the team's worker threads for execution, where each block is executed once. The semantics of the `sections`

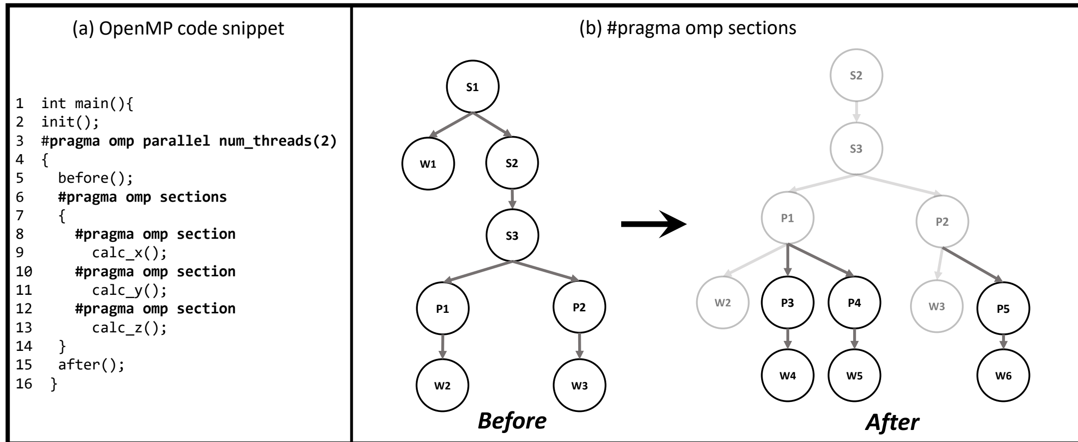


Figure 2.6: (a) OpenMP code snippet with a `sections` construct. (b) OSPG construction before entering the `sections` construct at line 6 and after exiting it at line 14.

construct imply that the specified structured blocks within the `sections` construct all execute in parallel. The OSPG already captures the parallel relation between the worker threads within a team, as described in Section 2.4.2. However, depending on worker availability, a pair of sections may execute serially on the same worker thread. To maintain the invariant of capturing the logical parallel relation between all `section` blocks even when they get serialized for an execution trace, we add a P-node during construction for each structured block. Furthermore, we add a W-node as the child of the newly added P-node to represent code fragment starting at each structured block within the `sections` construct.

Figure 2.6(a) illustrates a code snippet that has an OpenMP `sections` construct. The enclosing `parallel` region has a team of size two. The `sections` construct specifies three structured blocks that logically execute in parallel. Figure 2.6(b) depicts the OSPG construction before the program enters, and after it exits the sections construct (lines 6-14). The W-nodes corresponding to the parallel fragments at lines 9, 11, and 13 are represented with $W4$, $W5$, and $W6$, respectively. In a program trace with two worker threads, the fragments represented by $W4$ and $W5$ may execute serially. To capture the logical parallel relation between the fragment pair, it is necessary to add additional P-nodes during OSPG construction, which results in the addition of P-nodes $P3$, $P4$, and $P5$ for the example code snippet in Figure 2.6(a).

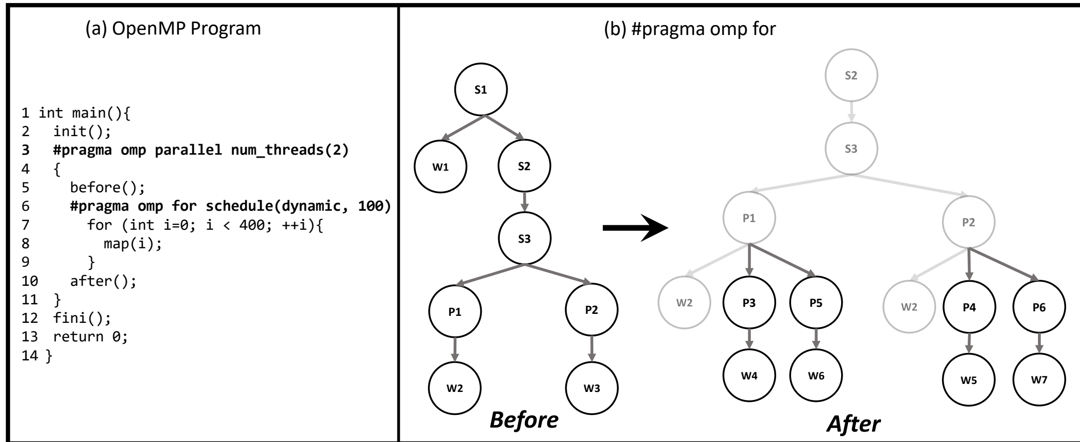


Figure 2.7: OSPG construction for the OpenMP `for` construct. (a) Example OpenMP code snippet with a `for` loop construct. (b) The program's OSPG construction before entering the loop at line 6 and after all iterations of the loop complete at line 9.

For Construct

The OpenMP `for` construct is used to create a bounded loop. The iterations associated with the loop execute in parallel by getting distributed between the enclosing `parallel` region's worker threads. When using the `for` construct, the developer specifies the loop's body, an iteration space, and an optional scheduling policy using the `schedule` clause. Based on the scheduling policy, the OpenMP runtime divides the iteration space into chunks of work assigned to worker threads. The static scheduling policy divides the loop's iteration into chunk sizes specified by the developer and assigns each iteration to worker threads in a round-robin fashion. In contrast, the dynamic scheduling policy distributes the loop's iteration into chunks. Each worker thread executes a chunk and, upon completion, requests another chunk until all chunks are executed. Based on the OpenMP API semantics, all pairs of loop chunks logically execute in parallel. For example, the program in Figure 2.7(a) has an OpenMP `for` construct that used the dynamic scheduling policy at line 6 with a chunk size of one hundred. This policy divides the iteration space into four chunks that logically execute in parallel.

The OSPG construction procedure for the `loop` construct can be thought of as a general case of the `sections` construct described in Section 2.4.5. Instead of capturing the parallel relation between the individual structured blocks of a sections construct, the OSPG construction captures the parallel relation between each chunk in the loop. Similar to the `sections` construct, an

OpenMP `for` loop's chunks may be serialized during the program's execution. However, the OSPG captures the logical parallel relation between all loop chunks. Thus, we create a P-node whenever a thread starts executing a new chunk. The fragment corresponding to each chunk is captured as a W-node child of the newly created P-node.

Figure 2.7(b) illustrates the OSPG construction of the example program before entering the `for` loop at line 6 and after exiting the `for` loop at line 9 when all four chunks complete execution. The parallel relation between chunks is encoded by P-nodes $P3 - P6$, and the computation performed by each chunk is W-nodes $W4 - W7$ respectively. Since the program executes in a `parallel` region with two worker threads, a possible execution trace may assign each thread two chunks. Other possible execution traces may assign chunks differently to each worker thread. While this may change the shape of the program's OSPG, it does not affect the series-parallel relation captured in the resulting OSPG.

2.4.6 Tasking Constructs

The OpenMP API supports task-based parallelism by providing different tasking constructs and task-specific synchronization. When using OpenMP's `task` construct, the developer specifies units of work as different tasks. As the program executes, the OpenMP runtime automatically assigns worker threads to execute the program's tasks. Tasks can recursively create child tasks, becoming their parent task. A child task may create additional tasks. These tasks are referred to as the descendant tasks of the parent task. Two child tasks are siblings if they share the same parent task. In the absence of synchronization, the continuation of a parent task may execute in parallel with its children and descendant tasks.

OpenMP specifies three task synchronization constructs: `taskgroup`, `taskwait`, and task dependencies using the `depend` clause. The first two constructs serialize the execution between parent tasks and their children or descendants. In contrast, task dependencies enable user-defined ordering between sibling tasks. These synchronization constructs may be combined to define complex ordering between task execution. In this section, we describe OSPG construction when using different tasking constructs.

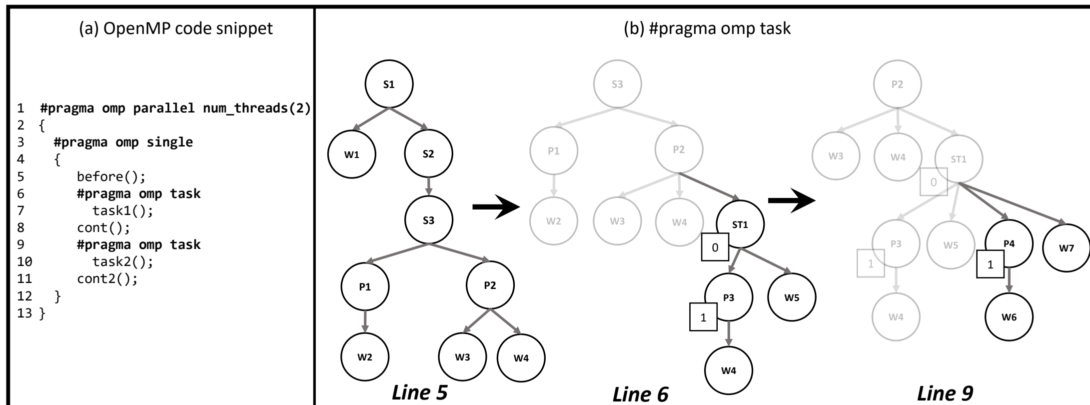


Figure 2.8: OSPG construction for a simple task-based OpenMP code snippet. (a) OpenMP code snippet with `task` construct and no task-specific synchronization. (b) OSPG construction when the encountering thread in the `single` construct reaches different lines in the `single` construct.

Task Construct

The OpenMP `task` construct is used to create a new task. When the execution trace reaches a `task` directive, the thread encountering it creates a new task. The OpenMP runtime maps the child task to execute by one of the worker threads in the enclosing parallel construct. The continuation and the child task logically execute in parallel. However, depending on the availability of worker threads during program execution, the two fragments may be serialized by running on the same thread.

Figure 2.8(a) illustrates an OpenMP code snippet that uses the `task` construct. It is a common pattern to create tasks within a `single` or a `master` directive to prevent the creation of duplicate tasks. In this example, the worker thread that enters the `single` construct creates two tasks at lines 6 and 9. The OpenMP runtime schedules the newly created tasks for execution by the worker threads of the enclosing parallel region. In the absence of task-specific synchronization and barriers, the newly created child task and the parent task's continuation logically execute in parallel. For example, in Figure 2.8(a), the task fragment at line 7 and the continuation at line 8 execute in parallel.

Whenever the parent task encounters a new `task` directive, and it creates its first child task in the program or creates a new task after a synchronization construct (*e.g.*, `taskwait`, `taskgroup`, and `barrier`), we add an ST-node to the OSPG. This ST-node is used to capture

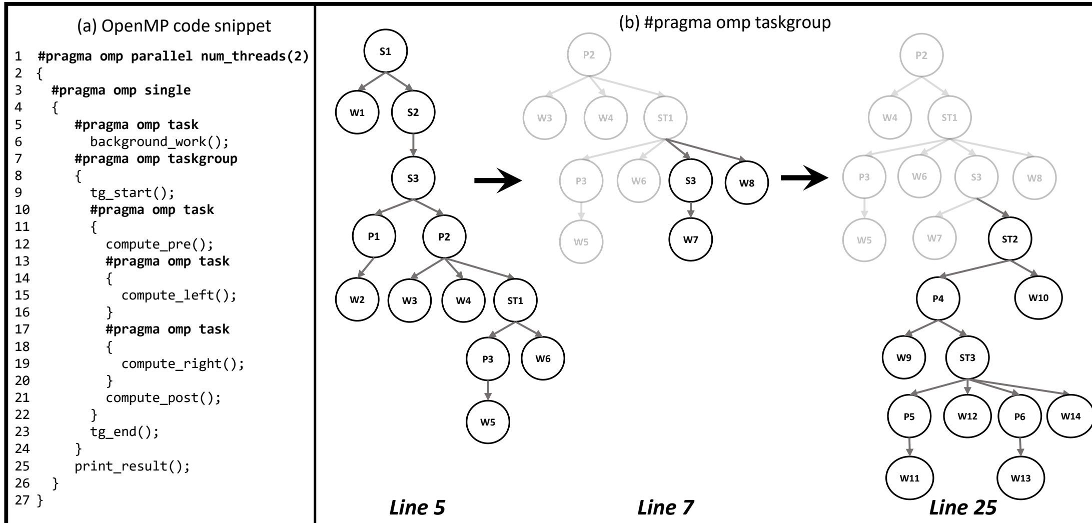


Figure 2.9: OSPG construction for a task-based OpenMP code snippet that uses `taskgroup` for synchronization. (a) OpenMP code snippet. (b) OSPG construction before the encountering thread reaches the `taskgroup` directive at line 5, upon reaching the `taskgroup` directive at line 7, and after exiting the `taskgroup` block at line 24.

the series-parallel relations between the current task's descendant tasks and its continuation after a `taskwait`. We describe this aspect of the OSPG construction in detail in Section 2.4.6. Further, whenever a parent task creates a new child task, the OSPG construction captures two newly created fragments of execution by creating two W-nodes. As illustrated in Figure 2.8(b), when the parent task creates a new task at line 6, W-node $W4$ represents the code fragment associated newly the created child task (lines 6-7), and $W5$ represents the fragment corresponding to the parent task's continuation (line 8). Further, the OSPG captures the parallel relation between W-nodes $W4$ and $W5$ by adding the P-node $P3$ as the parent node of $W4$, and setting $W5$ as the immediate right sibling of $P3$. Whenever the parent task creates its next child task (line 9 in Figure 2.8(a)), the same pattern is followed to expand the program's OSPG (illustrated by nodes $P4$, $W6$, and $W7$ in Figure 2.8(b)).

We note that since the creation of every new child task by a given parent task results in the creation of a new P-node under the parent's ST-node, the corresponding P-nodes for each child task fragments are sibling nodes in the program's OSPG. Thus, capturing the logical parallel relation between sibling task fragments. We use this invariant in capturing series-parallel relations when an OpenMP program uses task dependencies.

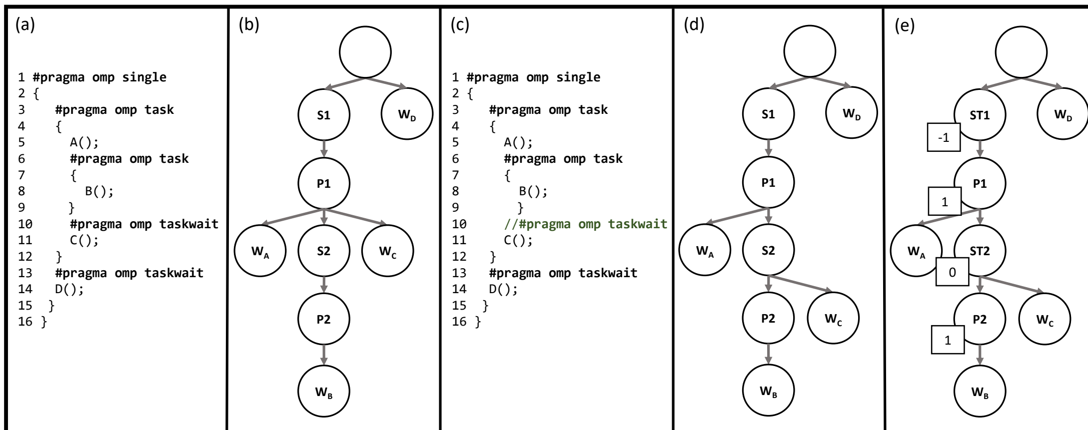


Figure 2.10: (a) OpenMP program snippet with perfectly nested taskwaits. (b) Part (a)'s code snippet's OSPG. (c) OpenMP program snippet that does not have perfectly nested taskwaits. (d) An incorrect attempt to construct part (c)'s OSPG in fast mode. (e) Correct OSPG construction for the code snippet in part (c) using ST-nodes.

Taskgroup Construct

A `taskgroup` construct enforces a serial ordering between the structured block associated with the `taskgroup` and the code fragments that execute after the `taskgroup`. Namely, the code following the `taskgroup` waits for the completion of all created tasks and their descendants within the taskgroup's structured block. Figure 2.9(a) contains an OpenMP code snippet that uses the `taskgroup` construct to synchronize the code fragments at lines 9-23 with the code fragment at line 25. As illustrated in Figure 2.9(b) at line 7, when the program trace encounters the beginning of the `taskgroup` directive, the OSPG encodes this serial relation by adding S-node, $S3$. All fragments in the `taskgroup` are contained in the subtree rooted at node $S3$. Moreover, the fragment executing after the `taskgroup` construct will be located as the right sibling of the newly created S-node, depicted as W-node $W8$ in Figure 2.9(b).

To illustrate that the OSPG construction supports the use of taskgroups in combination with nested `task` directives, Figure 2.9(b) illustrates the OSPG construction after all child tasks complete execution in the `taskgroup`. As illustrated, the complete OSPG correctly captures the series-parallel relations of all pairs of fragments in the code snippet from Figure 2.9(a), which includes nested tasks and a taskgroup.

Taskwait Construct

A `taskwait` construct specifies a serial ordering between the task encountering the `taskwait` construct and its child tasks. Unlike the `taskgroup` construct, a `taskwait` does not enforce a serial ordering with the parent task and its grandchildren and descendant tasks. While perfectly nested taskwaits produce a similar behavior to a taskgroup, it becomes more challenging to capture series-parallel relations when taskwaits are not fully nested correctly. Further, unlike the `taskgroup` construct, a `taskwait` construct must appear after the `task` construct it serializes. Hence, the OSPG construction algorithm should anticipate a potential `taskwait` directive in the future whenever the program execution trace encounters a `task` construct. In this part, we explain how the OSPG encodes the semantics of the `taskwait` construct using ST-nodes.

Consider the example OpenMP program snippet and its OSPG in Figure 2.10(a) and (b). The program creates two explicit tasks at lines 3 and 6. Assume that all functions A-D in Figure 2.10(a) and (d) execute serially and do not contain additional OpenMP constructs; thus, the programs have four fragments encoded with corresponding W-nodes with a subscript matching the function. In the program snippet in Figure 2.10(a), the `taskwait` constructs are perfectly nested. Hence, it is possible to encode the program fragments' series-parallel relations without using ST-nodes (*i.e.*, ST-nodes can be treated as S-nodes). Based on OpenMP semantics, a `taskwait` construct synchronizes the execution of the current task with its child tasks and not its descendant tasks. However, suppose the program has fully nested taskwaits. In such cases, after each `taskwait`, the current task is synchronized with all its descendant tasks. This behavior is similar to the semantics of OpenMP's `taskgroup` construct.

For example, the difference in logical series-parallel relations in the presence or absence of fully nested taskwaits is shown in Figure 2.10(a) and (d). Since the program snippet in Figure 2.10(a) uses fully nested taskwaits, its OSPG in Figure 2.10(b) correctly encodes the series-parallel relations between all fragments in the program. Because of the use of nested taskwaits in Figure 2.10(a), the function D executes serially after functions A-C. The `taskwait` at line 13 serializes the execution of functions A-C with the function D. Additionally, the `taskwait` at line 10 serializes the execution between the parent task's continuation, implying that function

C must execute serially after function B. Thus, the use of fully nested taskwaits ensures that the task encountering a `taskwait` is synchronized with its child tasks and all its descendant tasks. In a program's OSPG, the computation performed within a task is located at the subtree of the internal node created when the program encounters the task construct.

In contrast, if a program with tasks does not have fully nested taskwaits, we can no longer model the series-parallel relation between program fragments using S-nodes. Consider the program snippet in Figure 2.10(c). This program is similar to the program Figure 2.10(a), except that the `taskwait` at line 10 has been commented. Without it, the execution between functions C and B is no longer serialized. Hence, functions D and C may execution in parallel. Because of the `taskwait` at line 13, function D still executes after functions A and C. The OSPG in Figure 2.10(d) is an incorrect attempt to encode the program fragments' series-parallel relations in Figure 2.10(c) by only using internal S-nodes and P-nodes. The OSPG in Figure 2.10(d) is incorrect since the S-node $S1$ encodes a serial execution between functions D and B. Looking at the OSPG, we can see the problem is that some W-nodes (W_A and W_B) under the subtree of $S1$ execute in series with their right sibling W-node W_D, while W-node W_B executes in parallel with its right sibling W-node. Hence, in a program with no fully nested taskwaits, the use of just P-nodes and S-nodes is no longer sufficient to encode the program's series-parallel relations correctly. Instead, we need an internal node that partitions the W-nodes in its subtree into two sets that execute in series or parallel relative to its right siblings' W-node descendants. Thus, we model taskwaits with a different type of node (*i.e.*, ST-node) in the program's OSPG.

Naturally, the question arises: Which criteria should be used to partition the W-nodes under the subtree of an ST-node? The `taskwait` construct's semantics imply synchronization between the current task and its child tasks, but not its descendant tasks. This semantic indicates that we need to take the nesting level of a program fragment within a task into account when checking for its series-parallel relation with another program fragment. Further, we need to store taskwaits encounters during program execution. A node's `st_val` stores information about a task's nesting level and the potential encounter of a `taskwait` construct later in the execution trace. Thus the criteria to determine if a W-node under the subtree of an ST-node is a member of the first or the second subset, we use the nodes' st_val values on the path to the ST-node.

Intuitively, the st_val values on the path to the ST-node capture the nesting level and the number of encountered taskwaits. Whenever an ST-node is created, its st_val is set to zero. Suppose later in the program's execution trace, the task that created the ST-node encounters a taskwait. In that case, the OSPG construction algorithm captures this information by setting the st_val of the corresponding ST-node to -1. To capture the nesting level under an ST-node's subtree, the newly created P-node, which is the ST-node's immediate child, will have an st_val of one.

Invariant in the presence of ST-nodes. For a pair of W-nodes, ($W1$,$W2$), where $W1$ is under the subtree of ST-node $ST1$ and $W2$ is a right sibling of $ST1$, the pair of W-nodes execute in parallel if the sum of the `st_val` values of the OSPG nodes on the path from $ST1$ to $W1$ is a positive integer. Otherwise, the two W-nodes execute in series.

Going back to our example program in Figure 2.10(c), we can now correctly model the series-parallel relations between its program fragments by constructing an OSPG with ST-nodes. Consider the dynamic execution trace of this program snippet. When the implicit task executing the single creates the first task at line 3, the OSPG construction algorithm adds ST-node $ST1$ and P-node $P1$ with st_val of zero and one, respectively. Similarly, when the first explicit task creates the second task at line 6, we add ST-node $ST2$ and P-node $P2$ to the OSPG. Later in the dynamic execution trace, the implicit task encounters the taskwait construct at line 13. At this point, the OSPG construction algorithm has observed a taskwait and records this encounter by setting the st_val of ST-node $ST1$ to -1. When the dynamic execution starts executing function D, the OSPG construction algorithm adds the W-node W_D to the OSPG. At this point, the OSPG correctly captures the series-parallel relation between W_D and $W_A - W_C$. Because the sum of the st_val values from W_A to ST-node $ST1$ is zero, the OSPG encodes a serial relation between W_A and W_D. In contrast, since the sum of the st_val values from W_A to ST-node $ST1$ is one, the OSPG encodes a parallel relation between W_B and W_D. Thus, the OSPG in Figure 2.10(e) correctly encodes the series-parallel semantics of the program in Figure 2.10(c).

We want the OSPG construction algorithm to correctly model the program fragments' series-parallel relations for programs with all possible combinations of taskwaits in the program. Each time a parent task creates a child task, the nesting level of the program increases by one, and two W-nodes are created, one representing the beginning of the computation in the child task

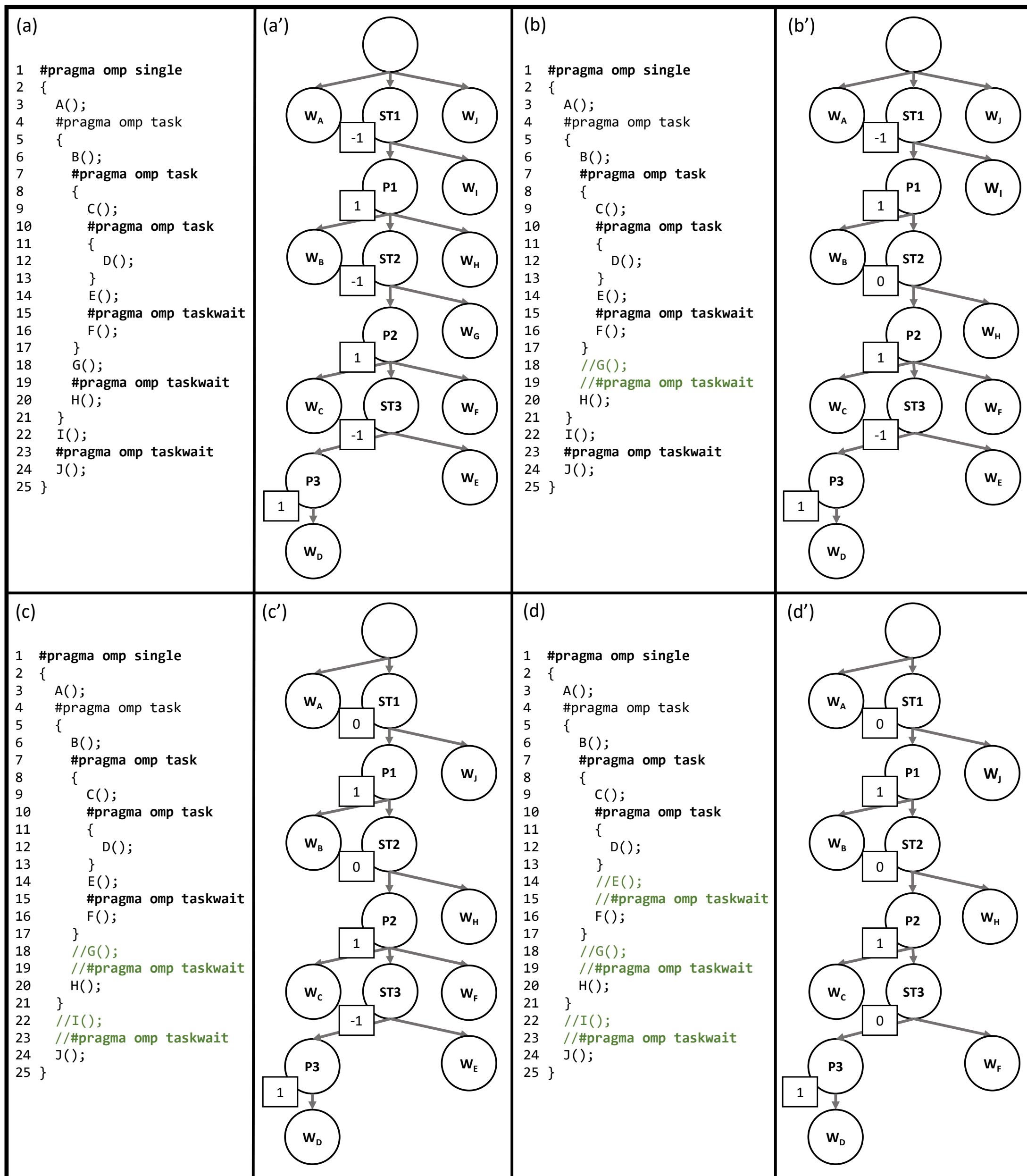


Figure 2.11: OpenMP program snippet with a task nesting level of 3, Illustrating different variations of taskwait usage and their corresponding OSPG, which correctly captures the series-parallel relations between program fragments.

and the other representing the continuation of the parent task. Consider a program with a root task that creates descendant tasks with a nesting level of n. In this program, there exists at least 2^n W-nodes under the subtree rooted at the root task's ST-node. Further, at the end of each child task, the program may have a taskwait or not. Thus, indicating that 2^n (size of the power set of n) program variations exist with different ways of adding taskwaits. Each variation may result in different series-parallel relations between the W-nodes under the root task's ST-node and its right sibling W-nodes or the descendant W-nodes of its right sibling nodes. Figure 2.11 shows four out of the eight possible program variations for a program snippet with a nesting level of three and their corresponding OSPG. In all cases, the constructed OSPG correctly captures the series-parallel relations between all pairs of program fragments.

Task Dependencies

In OpenMP, sibling tasks logically execute in parallel. However, using task dependencies with the `depend` clause, OpenMP supports user-defined ordering between sibling tasks. By specifying task dependencies among sibling tasks, a task will not start execution until all the sibling tasks it depends upon complete execution first. When using the `depend` clause, the developer can specify task dependencies by specifying `in`, `out`, or `inout` clauses over a shared memory location. We capture the serial ordering produced by task dependencies in the OSPG by adding E_{dp} edges between P-nodes for each pair of task dependency implied from the `depend` clause.

First, let's consider the example program Figure 2.12(a) that uses perfectly nested taskwaits. Hence, we can treat all its ST-nodes as S-nodes. Assume that all the functions A-H are serial functions. Thus they are encoded by a single W-node in the program's OSPG in Figure 2.12(b) with a corresponding subscript. By default, a task dependency clause only synchronizes the child fragments of the two dependent tasks and not necessarily the descendant fragment. For this example program, the task dependency between the task at lines 4 and 14 forces a serial ordering between functions E and D. However, the fully nested use of taskwaits in this program further imposes a serial ordering between function E and all the computation performed by child and descendant tasks of the task at line 4, effectively serializing the execution of functions E and C as well.

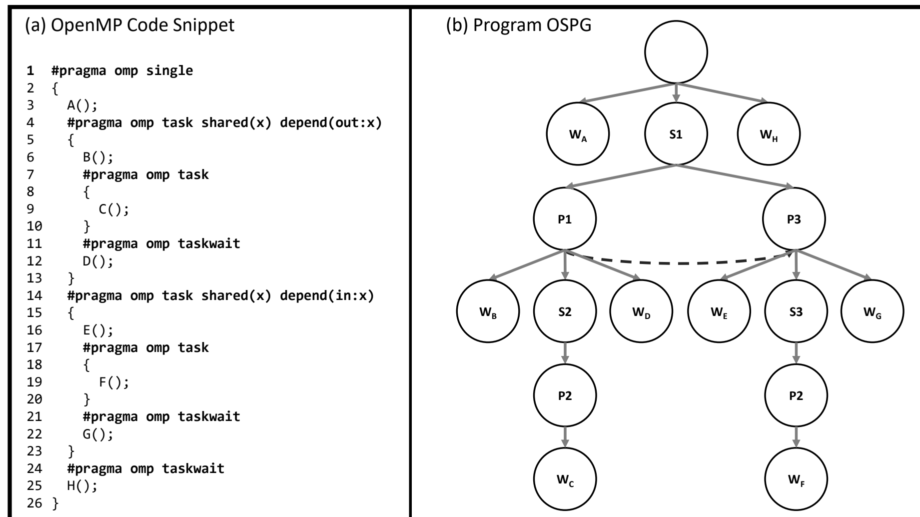


Figure 2.12: (a) OpenMP program snippet with task-dependencies in a program with perfectly nested taskwaits. (b) The code snippet's OSPG. Each W-node is named after its corresponding serial function in the program.

When two sibling tasks are dependent, in the program's OSPG, the tasks' corresponding P-nodes, (P_l, P_r), will be connected by a dependency edge. In the example program Figure 2.12(a) because of the use of nested taskwaits, the dependency edge encodes the serial relation between all the W-nodes of the subtree rooted at P_l and all the W-nodes of the subtree rooted at P_l. This serial relation can be seen in the program snippet and its OSPG shown in Figure 2.12. In this Figure, any W-node in the subtree rooted at $P1$, (W_B, W_C, W_D), executes in series to any the W-node in the subtree rooted at $P2$, (W_E, W_F, W_G).

For programs with nested taskwaits, it is sufficient to check for a directed path between a pair of P-nodes to infer serialization between the two P-nodes descendant W-nodes. For programs with unrestricted use of taskwaits, the check for serialization from a task dependency is slightly more involved. To see why let us look at the OpenMP program snippet in Figure 2.13, which is identical to the program snippet in Figure 2.12 except the `taskwait` at line 11 is commented out. In the absence of the `taskwait` construct at line 11, the task dependency only serializes the execution of functions D and E. However, functions C and E are not serialized, and they may execute in parallel (particularly when C is a long-running function relative to the other functions in the program).

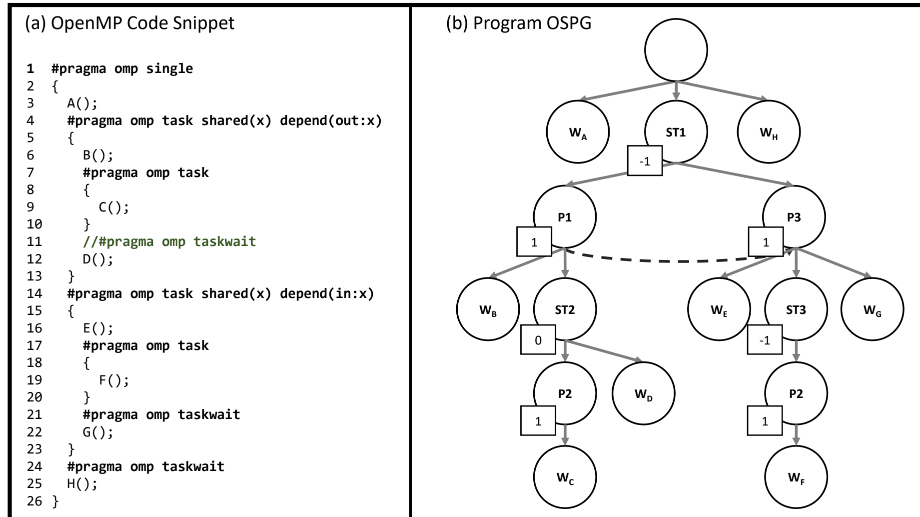


Figure 2.13: (a) OpenMP program snippet with task-dependencies in a program with unrestricted usage of taskwaits. (b) The code snippet's OSPG. Each W-node is named after its corresponding serial function in the program.

Hence in programs with unrestricted use of taskwaits, the existence of a directed path between a pair of P-nodes is not sufficient to indicate that a pair of descendant W-nodes are serialized. We also need to consider the nesting level of the W-node and the `taskwait` constructs encountered by the program trace. This involves checking the sum of st_val values of the OSPG nodes on the path from the left W-node to the LCA node of the W-node pair. Thus, the W-node pair execute in series if this sum equals one, and there exists a directed path comprised of dependency edges between the two sibling P-nodes. Using ST-nodes with st_val values, the OSPG in Figure 2.13(b) can be used to correctly check the logical series-parallel relations between all pairs of W-nodes. For example, consider the logical series-parallel relation between the W-node pair, (W_D, W_E). Since the sum of st_val values from W_D to the pair's LCA node is equal to one, we identify that the pair of W-nodes execute serially. In contrast, for the W-node pair, (W_C, W_E), the OSPG correctly encodes that the W-node pair may execute in parallel since the sum of st_val values from W_C to the pair's LCA node is not equal to one.

2.5 Identifying Logical Series-Parallel Relations

A program's OSPG can be queried to identify the logical series-parallel relations between any fragments encountered so far in the execution trace. Each fragment in the program has a

corresponding W-node in the program's OSPG. To identify the logical series-parallel relations between a pair of fragments, we first identify their corresponding W-node pair.

Algorithm 1 describes the steps involved in identifying the logical series-parallel relations between a pair of W-nodes (W_l, W_r). W-node W_l is to left of W_r. At line 2, the algorithm computes the LCA node of the pair of input W-nodes. Next, the algorithm identifies the child of the LCA node on the path to W_l, labeled L_{lc}. The algorithm checks the type of the left child of the LCA to determine the series-parallel relations between W-nodes W_l and W_r. Since the OSPG has four node types, four possible outcomes exist.

First, if the left child of the LCA is an S-node, then the two W-nodes logically execute in series. The correctness of this result follows directly from the definition of S-nodes in Section 2.3. By definition, if the left child of the LCA node, L_{lc}, is an S-node, it encodes the serial relation between all W-nodes in its subtree and its right sibling W-nodes. Since W-node W_l is in the subtree rooted at S-node L_{lc}, and W_r is a descendant of a right sibling of L_{lc}, the two W-nodes execute in series. Line 5 in Algorithm 1 performs this check.

Second, if the left child of the LCA is a W-node, similar to the previous case, the two W-nodes execute in series. This scenario can only occur if the W-node W_l is the direct child of the LCA node since W-nodes can only appear as leaf nodes in the OSPG. This case indicates the lack of an internal node to encode a potential parallel relation. Hence, similar to the first case, the two W-nodes execute in series (line 5 in Algorithm 1).

Third, if the left child of the LCA on the path to W_l is a P-node, the two W-nodes logically execute in parallel in the absence of task dependencies. However, the pair of W-nodes under consideration may be serialized by dependency edges. To check for serialization via task dependencies, the algorithm identifies the child of the LCA on the path to W_r. If the right child of LCA is also a P-node, a possible task dependency may serialize the two W-nodes. Thus the algorithm checks if there exists a directed path between these two P-nodes and that the sum of st_val values of the OSPG nodes on the path from W_l to the LCA node equals to one. If all these conditions are satisfied, the pair of W-nodes execute in series due to a task dependency between them. Otherwise, the two fragments logically execute in parallel (lines 9-12 in Algorithm 1).

Algorithm 1: The ISPARALLEL function returns true if a pair of W-nodes, W_l and W_r logically execute in parallel. The DIRECTEDPATH function returns true if there is a path comprised of dependency edges between the input P-nodes. The COUNT(W_l, L) function returns the sum of the nesting depth values (*i.e.*, `st_val`) on the path from W_l to L.

```
1  function IsParallel (W_l, W_r)
2      L ← LCA(W_l, W_r)
3      L_lc ← L.childTo(W_l)
4      L_rc ← L.childTo(W_r)
5      if L_lc.type = S-node ∨ L_lc.type = W-node then
6          return False
7      end
8      c ←Count(W_l, L)
9      if L_lc.type = P-node then
10         if L_rc.type = P-node ∧
11         c = 1 ∧ DirectedPath(L_lc, L_rc) then
12             return False
               ▷ series due to dependency edges
13         end
14         return True
15     end
16     else
           ▷ the left child of the LCA is an ST-node
17         if c > 0 then
18             return True
               ▷ parallel due to non-fully nested taskwaits
19         end
20         return False
21     end
22 function Count (W_l, L)
23     P ← W_l.parent
24     c ← 0
25     while P ≠ L do
26         c ← c + P.st_val
27         P ← P.parent
28     end
29     if W_l.parent.type = ST-node then
30         c ← c − W_l.parent.st_val
31     end
32     return c
```

Lastly, if the left child of the LCA on the path to W_l is an ST-node, the two W-nodes may execute in series or parallel depending on the nesting level of W_l and the number of taskwaits encountered in the program. Thus the algorithm checks if the two nodes are serialized by fully nested taskwaits. This check is realized by determining the count of the st_val values on the path from W_l to the child of the LCA node. Following the invariant described in Section 2.4.6, if this count is greater than zero, then the two W-nodes execute in parallel. Otherwise, they execute in series (lines 16-18 in Algorithm 1).

2.6 Summary

The OSPG is a series-parallel graph that has been designed to capture the logical series-parallel relations between fragments in an OpenMP program. Moreover, a program's OSPG can be incrementally constructed in parallel. It can be queried during construction to identify the logical series-parallel between any pair of fragments in the program execution trace. We designed the OSPG data structure with properties that make it useful for designing profiling tools and on-the-fly data race detection, which we describe in Chapters 3 and 4.

Chapter 3

Performance Profiling with What-if Analysis using OMP-ADVISER

In the last two decades, hardware with parallel execution units has become ubiquitous. With approaching the limits of frequency scaling, adding more parallelism has been the primary strategy to utilize increasing transistor counts in hardware. However, serial applications will not benefit from increased hardware parallelism. Instead, explicit parallel programming is required to design software that effectively utilizes parallel hardware resources. To develop applications with scalable performance, developers must address different sources of bottlenecks in the application.

This chapter introduces OMP-ADVISER, a performance profiler with what-if analysis for OpenMP applications that aims to assist developers in creating high-performing and scalable OpenMP programs. OpenMP applications must have sufficient parallelism to have scalable performance. With OMP-ADVISER, we make a case for measuring logical parallelism in OpenMP programs to identify serialization bottlenecks for a given input. A program's logical parallelism represents the program's speedup when it is executed on an idealized system with an infinite number of processors. We motivate the need for a parallelism centric profiler in Section 3.1. The key enabler in OMP-ADVISER's design is the use of a novel performance model, which we describe the details of its construction in Section 3.3. This performance model consists of the program's OSPG and measurement of computation at each program fragment, enabling OMP-ADVISER to compute the program's logical parallelism and identify serialization bottlenecks at a fine-grained granularity as described in Section 3.6. Our proposed performance model enables OMP-ADVISER to estimate improvements in parallelism before designing concrete strategies to address serialization bottlenecks in a region of code, which we call what-if analysis. OMP-ADVISER's what-if analysis enables the developer to answer the question of what are the program regions that matter for improving the program's performance.

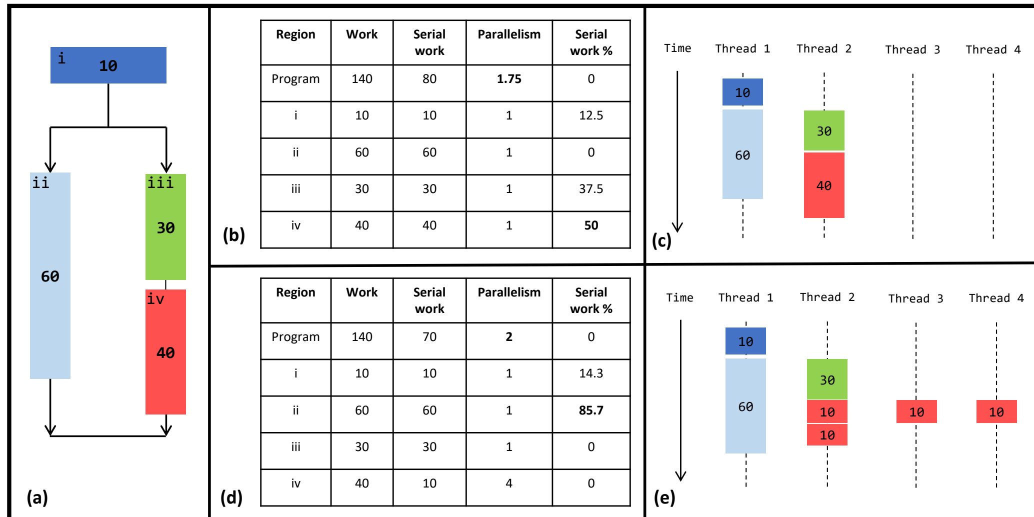
We describe the details of how our performance model is used to perform what-if analysis in Section 3.7. Even when the application has sufficient parallelism, it may not achieve good performance. First, we must ensure that the program has the right balance between parallelization and the runtime costs to create and orchestrate parallelism. We augment OMP-ADVISER's performance model by adding estimates of runtime overheads in Section 3.5, which enables OMP-ADVISER to pinpoint program regions with high tasking overheads. Second, a program may suffer from unintended resource contention caused by secondary effects of execution, such as false sharing. Such bottlenecks may manifest as parallel work inflation, where the parallel execution of the program performs additional computation for a given input compared to its serial execution. By comparing the performance model of the parallel execution to an oracle execution, OMP-ADVISER's differential analysis technique can pinpoint program regions with work inflation (Section 3.8).

3.1 Motivation

Different performance analysis tools have been proposed and are used by OpenMP application developers to identify performance bottlenecks [4, 5, 14, 50, 53, 71, 105, 176, 189]. The diversity among all these tools implies the importance and perhaps the difficulty of developing performant OpenMP applications. However, it may also raise the question of whether there is a need for yet another OpenMP performance analysis tool?

Most traditional performance analysis tools for OpenMP provide a thread-based view of the program's execution and identify bottlenecks based on which functions the program spends most of its time. While this approach effectively identifies some performance issues such as lack of thread utilization and load imbalance, it is less effective in identifying the sources of bottlenecks that are preventing the application from scaling to systems with more parallelism. Identifying such bottlenecks requires exploring dimensions in the design space that have received less attention.

Instead of providing a thread-based view of the execution, we advocate a design that is parallelism centric and examines the program's parallelism as the basis for identifying performance bottlenecks. The main idea behind the design of a parallelism centric performance



(b)

Region	Work	Serial work	Parallelism	Serial work %
Program	140	80	**1.75**	0
i	10	10	1	12.5
ii	60	60	1	0
iii	30	30	1	37.5
iv	40	40	1	**50**

(d)

Region	Work	Serial work	Parallelism	Serial work %
Program	140	70	**2**	0
i	10	10	1	14.3
ii	60	60	1	**85.7**
iii	30	30	1	0
iv	40	10	4	0

Figure 3.1: An example parallel application and using its parallelism profile to identify which program fragment to optimize first and the impact of first round of optimization (a) Illustration of a parallel program's fragments and the ordering of the fragment's execution. (b) Parallelism profile of the program. (c) A possible trace of the program when executed on a 4 core machine. (d) The program's parallelism profile after parallelizing fragment `iv` by a factor of 4. (e) The optimized program's trace when executed on a 4 core machine.

analysis technique is based on Amdahl's law[18], which states that the speedup of the parallel application is determined by the longest sequence of serial regions in the program[1].

A program's logical parallelism is defined as the ratio of the program's serial work over the work performed on its critical path. It represents the program's speedup when executed on an idealized system[2] with an infinite number of processors. Hence, to achieve scalable performance, a program must have sufficient parallelism. Alternatively, a program with serialization bottlenecks will have a low amount of logical parallelism.

While there exists prior research that uses logical parallelism measurements to identify performance bottlenecks in parallel applications [169, 192], the proposals in this space have focused on languages with structured parallelism or require the serial execution of the parallel application under test. In contrast, our work with OMP-ADVISER expands upon prior research by targeting OpenMP applications. An OpenMP application may use synchronization constructs (*i.e.*, task and task dependency) that are not available in structured task-parallel programs.

---

[1] Also referred to as the program's span or critical path.

[2] An idealized system with no scheduling or runtime overheads.

The key insight used to design OMP-ADVISER is that the OSPG, along with measurement of computation constitutes a performance model. This performance model can be used to compute the program's logical parallelism and identify serialization bottlenecks. After analyzing an OpenMP application, OMP-ADVISER uses its performance model to generate the program's parallelism profile. The parallelism profile includes the program's logical parallelism and the logical parallelism of each OpenMP directives used in the program. Further, the parallelism profile includes serial work contribution to the program's critical path for each OpenMP directive. Program regions with low parallelism on the critical path are potentially good candidates for optimization in the parallelism profile. For example, consider a parallel application trace, as illustrated in Figure 3.1(a), where each rectangle represents a fragment of execution[3]. The number at the center of each rectangle denotes the amount of computation in each fragment. Further, Roman numerals are used to identify each fragment.

For this simple example, the reader may quickly identify that the program has serialization bottlenecks and will not achieve scalable speedups by looking at its trace. However, as a program grows in complexity, identifying serialization bottlenecks by only looking at the program trace becomes more challenging. In contrast, the program's parallelism profile depicts the program's serialization bottlenecks in a report where the developer can identify serialization bottlenecks without examining large program traces. As shown in the program's parallelism profile (Figure 3.1(b)), the program has a parallelism of 1.75. Thus it will not be able to utilize more than two parallel execution units effectively.

While fragment `ii` performs the most amount of work in the application, the parallelism profile illustrates that it does not lie on the program's critical path. Hence the developer must optimize one of the other program fragments first. Let us assume that the developer has enough time to parallelize one program fragment. Which fragment should the developer prioritize? In this simple example, it is evident by looking at the program's parallelism profile that fragment `iv` must be optimized first since it performs the most amount of computation on the program's critical path. However, as a program grows in complexity, the impact of optimizing a program region on its parallelism becomes more challenging to assess. The developer may spend time and

[3]The longest sequence of instructions without encountering an OpenMP construct.

effort optimizing code that results in no improvement in parallelism. Furthermore, a programmer may want to know the impact of optimizing a region of code on upcoming hardware with a larger number of parallel execution units by executing the program on a smaller scale.

OMP-ADVISER's performance model captures the logical parallelism in an application. The logical series-parallel relations in a program are independent of the program's execution trace and the number of hardware threads the program is run with. Using this performance model, we devise a novel what-if analysis technique to answer the questions above. OMP-ADVISER's what-if analysis estimates the improvements in program parallelism and changes in its critical path by mimicking the effects of parallelizing a region of code. Thus, the developer can estimate the improvement in program parallelism before devising concrete optimization strategies and assess if the program will have scalable speedup without having access to higher core count systems. The results of using what-if analysis for hypothetically parallelizing fragment `iv` by $4\times$ is shown in Figure 3.1(d). The what-if analysis result illustrates that the program's parallelism will improve to 2. Additionally, the critical path will shift to fragment `ii`, which is now the limiting factor in the program's scalability.

While OMP-ADVISER's parallelism profile and what-if analysis proved useful in our initial experiments using it, we identified that we could improve what-if analysis in two directions. First, executing applications on actual hardware incurs runtime overheads to orchestrate the application's parallel execution. In practice, the amount of computation performed in an OpenMP construct must be sufficiently large to amortize the associated runtime costs. We augment OMP-ADVISER's performance model to estimate the runtime costs used for OpenMP constructs and report the runtime overhead ratio over the program computation for each OpenMP parallelism construct. We also incorporate this change when performing what-if analysis to identify if parallelizing a region further is no longer recommended due to high runtime costs.

To perform what-if analysis, OMP-ADVISER provides two options. In the first option, the developer chooses the program's regions to perform what-if analysis using program annotations. Sometimes the developer wants to parallelize program regions until a target parallelism is reached. In the second option, OMP-ADVISER identifies all regions that must be parallelized to increase the program's parallelism to a user-specified target value. We call this mode automatic what-if analysis. When using automatic what-if analysis, the user does not need to provide

any program annotations and instead specifies a target parallelism. OMP-ADVISER performs multiple iterations of what-if analysis and reports if reaching the target parallelism is feasible or not. If feasible, it provides a list of program regions that must be parallelized to reach the target parallelism.

Another common scalability bottleneck for parallel applications is undesired resource contention caused by secondary effects of execution. Developers use different types of synchronization in their OpenMP applications to ensure the correct ordering of operations in a parallel program. In contrast, undesired resource contention occurs as an artifact of executing on parallel hardware. This type of hardware resource contention may manifest as parallel work inflation in some metric of interest. Parallel work inflation is defined as additional time spent by threads in a multithreaded computation beyond the time required to perform the same computation in an oracle execution [150]. Identifying work inflation and its causes is difficult [13]. The application may need to run multiple, carefully designed performance experiments to identify them.

One of our technical contributions is devising a differential analysis technique that uses our performance model to identify program regions that suffer from different types of parallel work inflation at a fine granularity. The key insight is that our proposed performance model captures logical series-parallel relations in the program. These logical relations are the property of the program for a given input and do not change across executions. Hence our differential analysis technique identifies program regions that are experiencing unintended resource contention by comparing the performance model of the parallel execution with an oracle execution. OMP-ADVISER uses the single-thread OpenMP execution of the application as an oracle execution.

In this chapter, we make the following technical contributions. First, we propose a novel performance model that uses the OpenMP Series-Parallel Graph (OSPG) and fine-grained measurements to enable the measuring of the program's logical parallelism (Sections 3.3-3.4). Second, we augment the performance model to measure runtime overheads (Section 3.5). Third, we develop an algorithm that uses the performance model to identify serialization bottlenecks in the program. Further, we adapt the algorithm to work in an online setting during program execution (Section 3.6). Fourth, we devise a what-if analysis technique that enables developers to estimate performance improvements before designing concrete performance optimizations, identifying which program regions matter for improving parallelism. Our fifth contribution is the

design of a differential analysis technique that can identify program regions that are experiencing scalability bottlenecks caused by unintended resource contention (Section 3.8).

3.2 Overview of Profiling With OMP-ADVISER

This section provides an overview of OMP-ADVISER's performance model, parallelism profile, and what-if analysis. Figure 3.2(a) illustrates an example OpenMP program that uses tasking constructs to process a binary tree data structure in parallel. Further, the program creates a task that performs some background processing. The tree traversal function, `treeProcess`, uses a divide-and-conquer algorithm to perform a postorder traversal of the binary tree[4]. The program recursively creates more tasks to process each tree node's left and right subtree (lines 8-14 in Figure 3.2(a)). Once the nesting level reaches a user-defined `CUT_OFF` value (line 5 in Figure 3.2(a)), it stops creating more tasks and instead calls the serial function `treeProcessSerial`. The program uses OpenMP's taskwait construct to ensure that a parent tree node is processed after its left and right subtrees are processed[5]. When executing this program on a 4-core machine, the observed speedup is $2.1\times$ over serial execution.

Performance model. OMP-ADVISER constructs the program's performance model by executing the program in parallel for a given input. The performance model consists of two components. First, the OpenMP Series-Parallel Graph (OSPG) encodes the logical series-parallel relations between different all serial fragments of the program under test. Each W-node in the OSPG stores the fine-grained measurements of the computation performed by each serial fragment of dynamic execution, and each P-node corresponding to a tasking construct stores the runtime cost estimates associated with tasking constructs.

Figure 3.2(b) illustrates the example program's performance model. It is comprised of the program's OSPG, where each node is drawn with a circle, filled with an identifier denoting its type and a number to identify the node uniquely. Each OSPG edge is drawn with a solid arrow, its direction representing the parent-child relationship between the nodes. Each W-node, which represents a serial fragment of program execution, is accompanied by its corresponding

---

[4] A common parallel programming pattern to implement divide-and-conquer algorithms is the use of nested tasks [128].

[5] In this example, there is no ordering enforcement to process the left or the right subtree first.

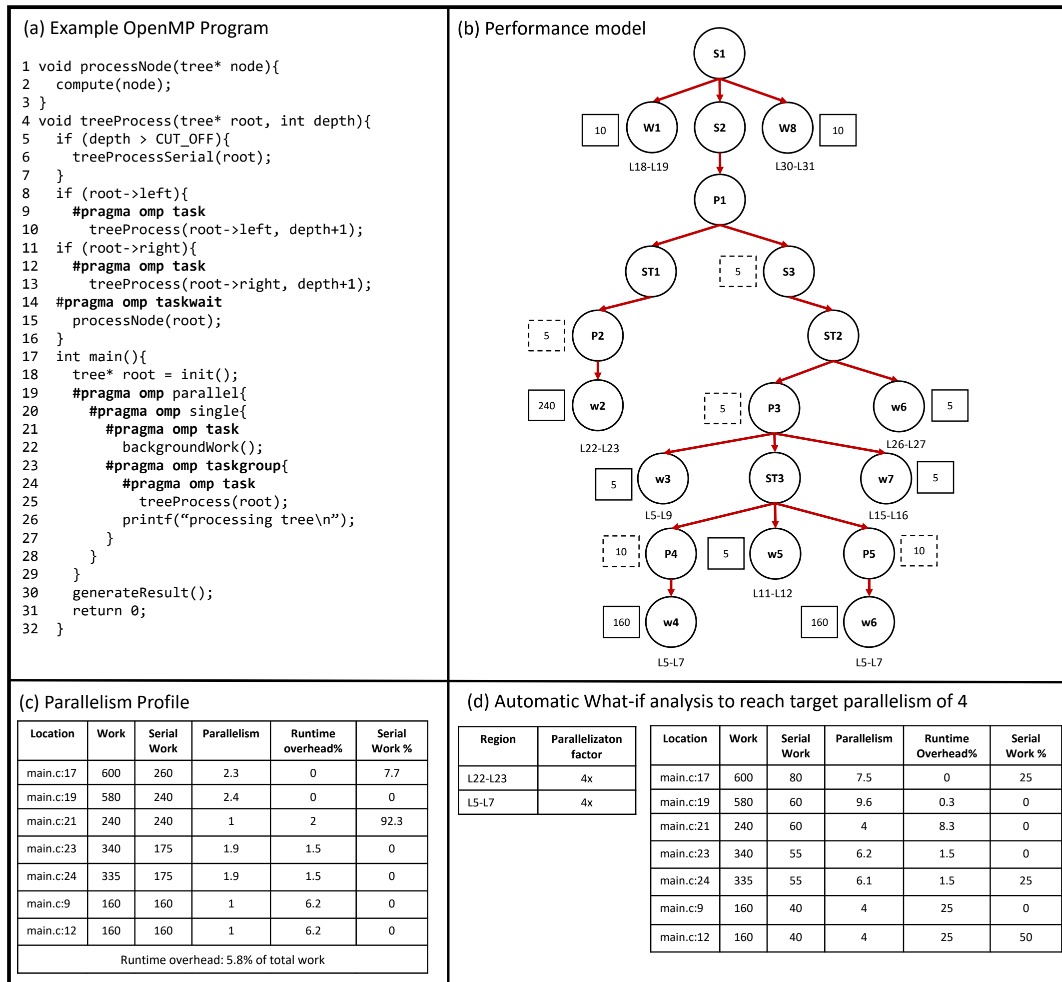


Figure 3.2: (a) An example OpenMP program. (b) Performance model of the example program in (a). (c) OMP-ADVISER's parallelism profile of the program. (d) OMP-ADVISER's automatic what-if analysis, identifying which regions to parallelize and the resulting what-if parallelism profile.

source code location. The amount of computation a W-node performs is shown in a solid rectangle to its side. The runtime costs attributed to OpenMP task-based parallelism constructs are depicted with dashed rectangles Figure 3.2(b). The internal nodes of the OSPG encode the logical series-parallel relation between the W-nodes (See Chapter 2 for more detail).

Parallelism profile. Using the information captured by the performance model, OMP-ADVISER constructs a parallelism profile which is reported back to the user. The parallelism profile aims to summarize the information stored in the performance model into a brief actionable report for the developer to identify performance bottlenecks. The example's program parallelism profile is illustrated in Figure 3.2(c). The parallelism profile aggregates the measurements stored

in each W-node to each static OpenMP directive (or the program itself if the fragment performs serial work outside of OpenMP parallel regions), which represents a row in the parallelism profile. Each row in the parallelism profile includes the following entries. (1) Location, which is the OpenMP directive's source code location in the program. (2) Work, which is the total computation attributed to the directive. (3) Serial work, which is the longest sequence of serial computation for the directive. (4) Parallelism, which is the ratio work over serial work and represents an upper bound on the region's speedup. (5) Runtime overhead, which is the ratio of estimated runtime costs over the work performed within the static directive represented in percentage. (6) Serial work percentage, this value denotes the directive's contribution to the program's critical path. If zero, the region does not lie on the program's critical path.

Intuitively, directives with low parallelism and high amounts of serial work percentage are potential candidates for future optimizations. For example, the third-row entry in the parallelism profile shown in Figure 3.2(c) satisfies both criteria. This entry corresponds to the task directive at line 21 in the program from Figure 3.2(a). Furthermore, a directive with high runtime overhead indicates that further parallelizing the directive's computation might not result in performance gains since the user-specified computation is not large enough to amortize the runtime costs associated with executing the computation in parallel.

Overall, the parallelism profile in Figure 3.2(c) denotes the example program has a logical parallelism of 2.3, which explains the observed speedup of 2.1× over serial execution on a 4-core machine. Moreover, the program's first serialization bottleneck is most likely the task created at line 21, which performs the majority of the program's serial work. Note that at this point, the program is spending more time in the `processNode` function (320 vs. 240 units). However, increasing the parallelism at function `treeProcessSerial` will not improve the program's parallelism and its speedup until the task at line 21 is optimized first. A profiling tool that reports the regions where the program spends the most time may incorrectly report the function `treeProcessSerial` as the program's current bottleneck.

What-if analysis. Before concretely optimizing the computation within the task at line 21 in Figure 3.2(a), the developer can use OMP-ADVISER's what-if analysis to generate its what-if parallelism profile. The main idea used by what-if analysis is that our performance model can be used to mimic effects parallelizing a region of code. Thus OMP-ADVISER can

estimate the changes in the program's parallelism and its critical path if selected program regions are parallelized. With what-if analysis, the developer chooses different program regions and a parallelization factor. Then OMP-ADVISER hypothetically parallelizes the user-selected regions and reports the program's parallelism profile with the hypothetical optimizations, which we refer to as the program's what-if profile. For example, performing what-if analysis on `backgroundWork` function (our candidate serialization from the task at line 21 in Figure 3.2(a)), the generated what-if parallelism profile estimates that the program's logical parallelism improves from 2.3 to 3. Further, the what-if profile will now indicate that the program's critical path has changed, and the tasks at lines 9 and 12 in Figure 3.2(a) now lie on the critical path and should be optimized next to increase the program's parallelism. To confirm whether parallelizing work within these tasks will increase the program's parallelism, we can perform another round of what-if analysis by including this region. The resulting what-if profile is illustrated in Figure 3.2(d), indicating that the program's parallelism improves from 3 to 7.5.

Instead of requiring users to choose what-if analysis regions by specifying program annotations, OMP-ADVISER provides an automatic what-if analysis mode. In this mode, the developer specifies their desired target parallelism and the program's input. OMP-ADVISER iteratively performs what-if analysis on program regions until it finds a list of regions that can be optimized to reach the target parallelism or determines that it is infeasible to reach the target parallelism. For example, performing automatic what-if analysis to reach the target parallelism of 4 for the example program in Figure 3.2(a) results in identifying the same two regions that we identified by performing two user-defined what-if analysis tests earlier. This mode's main benefit is that it does not require the user to select the what-if regions.

3.3 Performance Model Construction

To create a performance model to compute the program's logical parallelism and identify serialization bottlenecks, OMP-ADVISER constructs the program's OSPG incrementally in parallel. To limit performance perturbation and achieve low overheads during performance analysis, it is essential that the parallel construction of the OSPG is decentralized and makes limited use of synchronization. In this section, we first describe the OSPG construction algorithm

of a program during program execution. Further, we provide detail on OMP-ADVISER's usage of hardware performance counters to attribute fine-grained measurements to each serial fragment of the program.

3.3.1 OSPG Construction During Program Execution

The OSPG captures the logical series-parallel relations between each pair of serial program fragments. As we described in Chapter 2, for each program fragment, the OSPG construction algorithm creates a corresponding W-node. By construction, W-nodes are always leaf nodes in the OSPG. To encode the logical series-parallel relation between each pair of W-nodes, the OSPG uses different types of internal nodes (*i.e.*, S-nodes, P-nodes, and ST-nodes) and edges (*i.e.*, parent-child and dependency edges). The semantics of each OpenMP construct determines what nodes and edges are added to the OSPG as described in Section 2.4. In this section, rather than focusing on capturing the semantics of each OpenMP construct, we focus on the OSPG construction details as the program trace executes in parallel.

The OSPG is incrementally constructed online during program execution. Since OMP-ADVISER analyzes multithreaded OpenMP applications, we need to ensure that OMP-ADVISER scales with the applications' parallelism. To fulfill this requirement, the OSPG construction algorithm must execute in parallel with the application and avoid the use of synchronization primitives between threads when possible. Thus, the OSPG construction algorithm relies on thread-local data structures to limit the use of synchronization. During construction, each executing thread adds nodes to a subtree in the incrementally constructed OSPG. In the common case, each thread will operate on different subtrees of the OSPG, limiting the use of synchronization primitives. When the algorithm must update an OSPG node shared by two or more threads node in the OSPG, the use of some form of synchronization is unavoidable. However, for the applications we tested, such cases are the exception rather than the norm.

As a performance analysis tool, OMP-ADVISER is linked and runs in conjunction with the OpenMP program under test. The OSPG of a running program grows monotonically in proportion to the OpenMP constructs encountered during program execution. Thus, to analyze long-running applications with OMP-ADVISER, the OSPG construction algorithm should limit the amount of OSPG nodes kept in the main memory. Otherwise, the system will run out of

memory, and the profiling solution will fail to scale with long-running applications. To address this problem, whenever the program trace exits an OpenMP construct, the OSPG construction algorithm deallocates the nodes corresponding to the exited construct. Thus, limiting the memory usage of OMP-ADVISER to be proportional to the working set of the program.

Next, we describe the OSPG construction algorithm. Whenever the program trace encounters different OpenMP constructs, the construction algorithm updates the program's OSPG. The algorithm accomplishes this task by intercepting calls from the OpenMP runtime whenever an executing thread enters or exits an OpenMP construct. Intuitively, the algorithm incrementally constructs the OSPG using a per-thread stack and pushing unto it as the program enters different OpenMP constructs. In contrast, the algorithm removes OSPG nodes from memory whenever the program trace exits an OpenMP construct.

Intercepting calls to the OpenMP runtime can be accomplished in different ways, including instrumenting the OpenMP runtime with custom calls to the OMP-ADVISER library. We decided to use the OpenMP tooling (OMPT) interface. The OMPT interface is part of the OpenMP specification and is designed for tools that are linked and loaded directly in the OpenMP program [153]. Since OMPT is part of the OpenMP standard, it is guaranteed to be supported by current and future OpenMP implementation from different compiler vendors. Thus, when describing the OSPG construction, we also refer to the corresponding OMPT callbacks to ease the construction of OSPG for readers interested in using the OSPG in their OpenMP tools for other types of analysis.

Profiler Start/End

Profiler Start. When the profiler begins execution, OMP-ADVISER must initialize the data structures used for constructing the OSPG later in the execution when the program encounters OpenMP parallelism constructs. Each worker thread keeps track of a subset of the OSPG nodes using a per-thread stack, indexed by a unique thread identifier. Algorithm 2 illustrates the steps involved to initialize OMP-ADVISER to construct the OSPG. Note that OpenMP programs start execution with a serial thread, which for C/C++ programs is also called the main thread. This function takes the thread identifier (*i.e.*, $main_tid$) for the main thread and the number of worker threads that will be used during program execution (*i.e.*, $num_threads$) as an input.

Algorithm 2: OSPG construction at the start of the OMP-ADVISER profiler execution. The algorithm initializes the $nodes$ per-thread stack and creates the OSPG root node, S_1 and its W-node child, W_1.

```
1 procedure PROFILERSTART (main_tid, num_threads)
2     foreach tid ∈ num_threads do
          ▷ Initialize per-thread stack
3         nodes[tid] ← initializeStack(tid)
4     end
5     S_1 ← CreateSnode()
6     W_1 ← CreateWnode()
7     W_1.parent ← S_1
8     nodes[main_tid].push(S_1)
9     nodes[main_tid].push(W_1)
```

Algorithm 3: OSPG construction at the end of the profiler execution. The algorithm simply pops the last remaining active OSPG nodes in the main thread of execution.

```
1 procedure PROFILEREND (main_tid)
2     W_i ← nodes[main_tid].pop()
3     S_1 ← nodes[main_tid].pop()
4     Finalize()
```

The loop at line 2 initializes an empty stack for each worker thread. Next, the algorithm creates two OSPG nodes, the root node and its child W-node representing the first fragment of execution (lines 5-7 in Algorithm 2). Finally, the algorithm pushes the newly created nodes to the main thread's stack.

OMP-ADVISER uses the `ompt_initialize_callback` to implement Algorithm 2. This callback is guaranteed to be called after the OpenMP runtime is initialized, but before the runtime executes any OpenMP construct.

Invariant for the per-thread stack. For each stack, the OSPG nodes are stored in ascending depth order from the root node. This invariant ensures that popping the top node from a non-empty node stack results in either an empty stack or a stack with a new top node closer to the root node than the popped node. This invariant effectively bounds each stack's size to $O(h)$, where h is the height of the OSPG. Further, the top node in the stack after each operation stores the W-node corresponding to the current fragment of execution by an OpenMP worker thread. The second invariant is used to facilitate the attribution of measurements to W-nodes during OSPG construction.

Algorithm 4: OSPG construction when a parallel construct is encountered. The construction algorithm relies on two callbacks to update the per-thread view of the program's OSPG.

```
1  procedure ParallelBegin (master_tid)
2      W_i ← nodes[master_tid].pop()
3      S_par ← CreateSnode()
4      S_par.parent ← nodes[master_tid].top()
5      S_bar ← CreateSnode()
6      S_bar.parent ← S_par
7      nodes[master_tid].push(S_par)
8      nodes[master_tid].push(S_bar)
9  procedure WorkerBegin (tid, master_tid)
10     if tid ≠ master_tid then
11         S_bar ← nodes[master_tid].top()
12         S_par ← nodes[master_tid].top(−1)
13         nodes[tid].push(Copy(S_bar))
14         nodes[tid].push(Copy(S_par))
15     end
16     P_i ← CreatePnode()
17     P_i.parent ← nodes[tid].top()
18     W_i ← CreateWnode()
19     W_i.parent ← P_i
20     nodes[tid].push(P_i)
21     nodes[tid].push(W_i)
```

Profiler End. When the program ends, the OSPG has been fully constructed by the earlier calls to the OMP-ADVISER library. At this point in the execution of an OpenMP program, the only active thread is the main thread since no active parallel regions exist. Thus, all node stacks must be empty except the one corresponding to the main thread. The main thread's stack consists of two nodes. The top W-node refers to the last fragment. The last remaining internal node is the OSPG root node. Thus, as illustrated in Algorithm 3, OMP-ADVISER pops the remaining nodes from the main thread's stack before making a call to the `Finalize` function, which performs different operations depending on the profiling mode used. For offline analysis mode, the `Finalize` function serializes all the remaining unserialized OSPG nodes so that OMP-ADVISER's offline analyzer has information about all of the program's OSPG nodes. Alternatively, for online profiling mode, this function performs the final parts of the parallelism profiling algorithm (*i.e.*, computing the program's work, adding the last fragment of execution to the program's critical path, *etc.*.) to generate the parallelism profile for the program.

Parallel Construct

Entering a Parallel Region. When the program trace enters a parallel region, the serial (*i.e.*, master) thread, creates a team of worker threads that includes itself as a member of the team. Algorithm 4 shows the steps involved to update the program's OSPG. First, the algorithm is broken down into two procedures, denoting two different callbacks to the OMP-ADVISER library. Between the two callbacks, the OpenMP runtime spawns and prepares the worker threads for executing the code within the parallel region. The construction should use two distinct callback to avoid wrongly attributing the computation performed by the OpenMP runtime to the program's W-nodes. This approach ensures that the algorithm avoids inaccurate work measurements when entering a parallel region.

The first callback, `ParallelBegin`, occurs when the master thread encounters the parallel construct before the runtime creates the new worker threads. At this point in the execution trace, the W-node on top of the master thread's stack corresponds to the fragment of code executed by the main thread before the parallel region. First, this W-node is popped from the master thread's stack (line 2). Next, we add two S-nodes to its stack (lines 3-8). The first S-node is used to encode the serial relation between the code fragments within the parallel region and the master thread's continuation after exiting the parallel region. The second S-node is added in anticipation of OpenMP barriers within the parallel region if any.

The second callback, `WorkerBegin`, is called right before each worker thread starts executing the parallel region. For each worker thread, excluding the master thread, The Algorithm makes a copy of the barrier and parallel region S-nodes on top of the master thread's stack (lines 10-15 in Algorithm 4). By making a per-thread copy of these S-nodes, we postpone all shared updates to the S-node S_{par} to when the program encounters a global synchronization construct (*i.e.*, a barrier). All worker threads can update the nodes locally with the amount of work in their subtrees without synchronization and overlap the synchronized updates to S_{par} with the implicit barrier that exists at the end of all OpenMP parallel regions. Thus, limiting the impact of OMP-ADVISER on the critical path of the program under test.

After making copies of the S-nodes (lines 10-15), each worker thread, including the master thread, inserts a P-node into its stack to encode the parallel relation between each worker thread in

the parallel region. Finally, each worker thread adds a W-node to mark the fragment of execution at the beginning of a parallel region (lines 16-21). OMP-ADVISER utilizes the OMPT callbacks `ompt_callback_parallel_begin` and `ompt_callback_implicit_task` to implement the callbacks in Algorithm 4.

Algorithm 5: OSPG construction when exiting a parallel region. The construction algorithm relies on two callbacks to update the per-thread view of the program's OSPG.

```
1  procedure WorkerEnd (tid)
2      W_i ← nodes[tid].pop()
3      P_i ← nodes[tid].pop()
4      S_bar ← nodes[tid].pop()
5  procedure ParallelEnd (master_tid, num_workers)
6      S_par ← nodes[master_tid].pop()
7      foreach tid ∈ num_workers do
           ▷ aggregate results in master stack's S-node
8          if tid ≠ master_tid then
9              S_par_copy ← nodes[tid].pop()
10             UpdateNode(S_par, S_par_copy)
11         end
12     end
13     ProcessRegion(S_par)
14     W_i ← CreateWnode()
15     W_i.parent ← nodes[master_tid].top()
16     nodes[tid].push(W_i)
```

Exiting a Parallel Region. Upon exiting a parallel region, the worker threads execute Algorithm 5. At this point in the execution, the program trace is exiting the parallel region. Thus the construction algorithm must remove the remaining nodes associated with the parallel region. Algorithm 5 is divided into two callbacks. The first callback `WorkerEnd` is called whenever a worker thread finishes executing the parallel region. When called, the corresponding worker thread pops the W-node associated with the previous fragment of execution (line 2). Additionally, the algorithm pops the P-node and S-node that were either added at the beginning of the parallel region or the latest encountered barrier (lines 3-4). At this point, the top OSPG node in each thread-local stack is the S-node corresponding to the parallel region, and the thread-local copies that were added at the beginning of the parallel region, node S_{par} (Algorithm 4 line 13).

To keep the results under the subtree rooted at S_{par} consistent, it is necessary to propagate all the information stored in each thread-local copy of S_{par} to the master thread's copy. Updating

the master thread's copy at the end of function WorkerEnd results in a data race. Thus, this update is performed in the second callback, ParallelEnd, which is only executed by the master thread after all worker threads have completed the WorkerEnd callback. Updating S_{par} in lines 6-11 in Algorithm 5, results in S_{par} containing all the aggregate profiling information in its subtree. Hence, depending on the profiling mode (see Section 3.6), the function ProcessRegion, is used to either update the program's parallelism profile or serialize the subtree associated with the parallel region. By delaying this update to the end of the parallel region and piggybacking on the implicit barrier's synchronization at the end of all OpenMP parallel regions, the update to S_{par} is an example of lazy synchronization in the OSPG construction algorithm.

Lastly, Algorithm 5 adds a new W-node to the master thread's stack to represent the code fragment corresponding to the master's thread continuation after the parallel region. OMP-ADVISER utilizes the OMPT callbacks ompt_callback_parallel_end and ompt_callback _implicit_task to implement the callbacks shown in Algorithm 5.

Barrier Construct

An OpenMP barrier construct synchronizes the code fragments before the barrier with the code that executes after it. Thus, all worker threads must first finish executing outstanding code fragments before the program can continue executing the code after the barrier construct (See Section 2.4.3 for a description of how the OSPG captures the series-parallel relations between the fragments before and after a barrier). For a task-based OpenMP program, a barrier results in waiting for the completion of all tasks created before a barrier construct. Additionally, the worker threads entering a barrier construct become eligible to execute explicit tasks in the task queue, if any. The OSPG construction algorithm must ensure that it correctly updates the OSPG when the runtime schedules tasks to run while the worker thread has reached a barrier. Thus, to handle such situations and exclude waiting times within a barrier during work measurements of program fragments, the OSPG construction algorithm uses two callbacks to update the program's OSPG during a barrier construct.

Entering a Barrier. As illustrated in Algorithm 6, whenever a worker thread enters a barrier, the algorithm removes the W-node associated with the fragment of execution ending at

Algorithm 6: OSPG construction when the program trace encounters a barrier construct. The algorithm uses two callbacks to distinguish when a worker threads enters a barrier and when it exits a barrier construct. In `BarrierExit`, $first_tid$ denotes the worker thread that first exits the barrier construct.

```
1  procedure BARRIERENTER (tid, num_threads)
2  |   W_i ← nodes[main_tid].pop()
3  procedure BARRIEREXIT (tid, first_tid)
4  |   P_i ← nodes[tid].pop()
5  |   S_{bar_j} ← nodes[tid].pop()
6  |   if tid = first_tid then
7  |   |   S_{bar_{j+1}} ← CreateSnode()
8  |   |   S_{bar_{j+1}}.parent ← nodes[tid].top()
9  |   |   nodes[tid].push(S_{bar_{j+1}})
10 |   end
11 |   else
12 |   |   S_{bar_{j+1}} ← nodes[tid].top()
13 |   |   nodes[tid].push(Copy(S_{bar_{j+1}})
14 |   end
15 |   P_{i+1} ← CreatePnode()
16 |   P_{i+1}.parent ← nodes[tid].top()
17 |   W_{i+1} ← CreateWnode()
18 |   W_{i+1}.parent ← P_{i+1}
19 |   nodes[tid].push(P_{i+1})
20 |   nodes[tid].push(W_{i+1})
```

the current barrier. At this point, the algorithm makes no further changes to the OSPG, since the OpenMP runtime may still schedule explicit tasks, which results in updating aggregate measurements in the current per-thread barrier P-node and S-node.

Exiting a Barrier. The callback `BarrierExit` in Algorithm 6, is called whenever a worker thread is ready to exit the barrier. At this point, the top of worker thread's stack is comprised of a P-node and S-node that were created at a previous `BarrierExit` or `WorkerBegin` (See Subsection 21). Since all worker threads are ready to exit the barrier, the computation encoded under the subtree corresponding to the current S-node must be completed. Thus, the OSPG construction algorithm can safely remove the S-node and its child P-node from the per-thread stack(lines 4-5 in Algorithm 6).

Once these two nodes are popped from the worker thread's stack, the construction algorithm must add a new S-node in anticipation of the upcoming barrier. However, the algorithm must ensure that only one primary copy of the new barrier's corresponding S-node is created. Thus,

the algorithm uses a conditional branch to ensure that only the first worker thread exiting the barrier creates the new S-node, and other worker threads make thread-local copies of the newly created S-node (lines 6-14 in Algorithm 6). Next, each worker thread adds a new P-node to its stack to represent the worker threads' parallel relation. Lastly, the algorithm creates a new W-node to represent the fragment of execution starting after the barrier construct, pushing the newly created P-node and W-node to the worker thread's stack (lines 15-20).

We use the `ompt_callback_sync_region` OMPT callback to update the program's OSPG when the program encounters the barrier construct.

Worksharing Constructs

OpenMP worksharing constructs are used to distribute the computation within the worksharing construct among the worker threads that encounter it. The single and for loop constructs are among the most commonly used worksharing constructs. Next, we describe the OSPG construction details for these worksharing constructs.

A single construct specifies a structured block that is only executed by only one thread in the team. The other threads in the team go over the structured block and execute the rest of the enclose parallel region. A common use of the single construct is with OpenMP tasking constructs where a task is created within a single block to avoid creating redundant tasks by other worker threads in the team.

The worksharing for construct specifies that the associated loop's iteration will be distributed for parallel execution among the team's worker threads encountering the for construct. An OpenMP for loop accepts a schedule clause that specifies how the iteration space of the for loop is partitioned into chunks and divided for execution to each worker thread. OMP-ADVISER supports loops with static or dynamic scheduling. With static scheduling, the OpenMP runtime divides the iteration space by the number of worker threads in the team where each worker thread roughly executes the same number of iterations. With dynamic scheduling, the iteration space is divided into user-defined chunk sizes. Under dynamic scheduling, each worker receives one chunk for execution and requests another chunk when done until all chunks are executed.

Entering/Exiting a Single Construct. The OSPG construction algorithm for a single construct is simple. The parallel relation between the thread executing the single construct and

other worker threads in the team is already encoded in the OSPG, either from the P-nodes added when the program enters a parallel region (See Section 21) or the P-nodes added after exiting the latest encountered barrier (See Section 16). Hence, no new internal nodes are added to the program's OSPG when it encounters a single construct. When the single thread enters the single construct, it denotes the end of the single thread's program fragment. The construction algorithm pops the W-node corresponding to this fragment from its stack and pushes a new W-node to denote the program fragment's start within the single block. Alternatively, when the single thread exits the single construct, the construction algorithm pops the W-node corresponding to the last fragment of execution within the single block and pushes a new W-node to encode the start of the program fragment after the single block.

Entering/Exiting a Worksharing Loop Construct. The OSPG construction algorithm for loops with static scheduling is similar to the worksharing construct explained in the previous paragraph. However, for loops with a dynamic schedule, the OSPG construction algorithm must create a W-node for each loop chunk, and it must capture the logically parallel relation between all loop chunks. A pair of dynamic loop chunks may get executed by the same worker thread in a program trace. However, the two chunks logically execute in parallel. Hence, to capture this parallel relation even when the same thread executes the two chunks, the construction algorithm must add a P-node for each loop chunk.

Compared to static loops where each worker thread receives a callback at the start and the end of the loop, the OSPG construction algorithm receives an additional callback for dynamic loops whenever the OpenMP runtime dispatches a loop chunk to a worker thread for execution. The OSPG construction algorithm uses this callback to pop the W-node and P-node corresponding to the previous chunk if any. Moreover, it pushes a new P-node and W-node to encode the current loop chunk unto the worker thread's stack.

The OSPG construction algorithm uses the OMPT callback `ompt_callback_work` whenever a worker thread encounters the beginning/end of a worksharing construct. Further, it uses the callback `ompt_callback_dispatch` to update the program's OSPG whenever a new dynamic loop chunk is dispatched to a worker thread.

Algorithm 7: OSPG construction when the program encounters task creation constructs. The construction algorithm relies on different callbacks to both the parent and child tasks' view of the program's OSPG.

```
1  procedure TASKCREATE (tid, task_id)
2      W_i ← nodes[tid].pop()
3      if NODETYPE(nodes[tid].top()) ≠ ST-node then
4          ST_task ← CreateSTnode()
5          ST_task.parent ← nodes[tid].top()
6          nodes[tid].push(ST_task)
7      end
8      P_task ← CreatePnode()
9      P_task.parent ← nodes[tid].top()
10     TaskMap.Insert(task_id, P_task)
11     W_{i+1} ← CreateWnode()
12     W_{i+1}.parent ← nodes[tid].top()
13     nodes[tid].push(W_{i+1})
14 procedure TASKBEGIN (tid, task_id)
15     P_task ← TaskMap.get(task_id)
16     nodes[tid].push(P_task)
17     W_i ← CreateWnode()
18     W_i.parent ← nodes[tid].top()
19     nodes[tid].push(W_i)
20 procedure TASKEND (tid, task_id)
21     W_i ← nodes[tid].pop()
22     P_task ← nodes[tid].pop()
```

Tasking Constructs

OpenMP supports task-based parallelism by providing the task directive to spawn new tasks, a set of task-based synchronization constructs, such as taskwait and taskgroup, and user-defined ordering between sibling tasks using task dependency clauses. In this section, we describe the OSPG construction algorithm when these constructs are encountered during program execution.

The OpenMP task construct is used to create a new explicit task. The worker thread encountering a task construct creates a new explicit task that gets scheduled for execution by the OpenMP runtime by assigning it to any worker thread in the team. A task construct specifies a structure block. The new task's code is the code associated with this structured block. The OSPG construction algorithm distinguishes between a task creation event and when the task begins execution event. The thread that creates a task may not be the thread chosen by the OpenMP runtime to execute the task. The OSPG construction algorithm must update the program's OSPG

as described in Section 2.4.6 while updating the per-thread stacks to reflect a correct view of the program's OSPG.

A taskgroup construct enforces a serial ordering between the structured block associated with the taskgroup and the code fragments that execute after the taskgroup. Namely, the code following the taskgroup waits for the completion of all created tasks and their descendants within the taskgroup's structured block. The OSPG construction algorithm updates the program's OSPG when the thread encountering the taskgroup enters/exits the code block specified by the taskgroup construct. In contrast, a taskwait construct enforces a serial ordering between the thread encountering the taskwait and only its direct child tasks. Further, unlike a taskgroup, a taskwait does not specify a block of code. Instead, a taskwait may occur at any point within an OpenMP parallel region. Hence during program execution, we do not know a priori if a parent task will encounter a taskwait later in the execution, requiring our OSPG construction algorithm to create an ST-node in anticipation of encountering a taskwait.

Algorithm 7 illustrates the callbacks the OSPG construction algorithm receives for these task and task-specific synchronization constructs.

Creating a Task with the Task construct. When the execution trace encounters a task construct, the OSPG construction algorithm receives the TASKCREATE callback, shown in Algorithm 7, with the encountering thread's identifier (*i.e.*, tid) and the newly created task's identifier (*i.e.*, $task_id$). We first pop the W-node corresponding to the program fragment ending at the task construct (line 2). As described in Section 2.4.6, the OSPG construction algorithm must add an ST-node to the current thread's stack if the current task is creating its first child task after the most recent synchronization construct. The algorithm performs this check by checking if the top of the current thread's stack is an ST-node or not. If the top of the stack is not an ST-node, the algorithm creates an ST-node that indicates the parent task's first child task (lines 3-7). Thus, when the algorithm reaches line 8, it is guaranteed that the top of the stack includes an ST-node.

At this point, the OSPG construction algorithm creates a P-node to encode the parallel relation between the child task and the parent task's continuation. However, we do not add the newly created P-node to the current worker thread's stack since the child task may run by a different worker thread. Instead, we store it in a map data structure using its identifier $task_id$

Algorithm 8: OSPG construction when the program encounters different task synchronization constructs.

```
1  procedure TASKGROUPBEGIN (tid)
2      W_i ← nodes[tid].pop()
3      S_tg ← CreateSnode()
4      S_tg.parent ← nodes[tid].top()
5      nodes[tid].push(S_tg)
6      W_{i+1} ← CreateWnode()
7      W_{i+1}.parent ← nodes[tid].top()
8      nodes[tid].push(W_{i+1})
9  procedure TASKGROUPEND (tid)
10     W_i ← nodes[tid].pop()
11     S_tg ← nodes[tid].pop()
12     W_{i+1} ← CreateWnode()
13     W_{i+1}.parent ← nodes[tid].top()
14     nodes[tid].push(W_{i+1})
15 procedure TASKWAITBEGIN (tid)
16     W_i ← nodes[tid].pop()
17 procedure TASKWAITEND (tid)
18     ST_task ← nodes[tid].pop()
19     ST_task.st_val ← −1
20     W_j ← CreateWnode()
21     W_j.parent ← nodes[tid].top()
22     nodes[tid].push(W_j)
```

as the key (lines 8-10). Lastly, we create and push a W-node corresponding to the parent task's continuation after the task construct (lines 11-13).

Task Begin/End. Right before the OpenMP runtime starts executing the body of a newly created task, the worker thread that the runtime chooses to execute the task receives a TASKBEGIN callback, shown in Algorithm 7, including the task's identifier $task_id$. At this point, we check the task map to retrieve the P-node corresponding to the executing task (line 15). This P-node, P_{task}, is guaranteed to exist in this map since a task can only begin execution after it was created at an earlier TASKCREATE callback. Next, the algorithm pushes the P-node unto to the thread's stack. Lastly, a new W-node that represents the program fragment at the beginning of the task's body is created and pushed to the worker thread's stack (lines 17-19).

Right after a worker thread finishes executing an explicit task, the worker thread receives a TASKEND callback. This callback performs cleanup by removing the W-node corresponding to

the last code fragment in the task and subsequently removing the P-node corresponding to the completed task (lines 21-22 in Algorithm 7).

The OSPG construction algorithm uses the OMPT callback `ompt_callback_task_create` whenever a worker thread encounters a task construct. Further, to receive a callback whenever an explicit task starts/finishes execution, we use the callback `ompt_callback_task_schedule`.

Entering/Exiting a Taskgroup. When the current task encounters a taskgroup construct, before the worker thread starts executing the block specified by the taskgroup construct, it receives a TASKGROUPBEGIN callback. As shown in Algorithm 8 lines 1-8, the OSPG construction algorithm adds an S-node to the current thread's stack to encode the serial relation between the code fragments corresponding to the taskgroup block and its continuation (lines 3-5). The callback ends by creating and pushing a W-node to represent the first fragment of execution within the taskgroup block.

Once a worker thread finishes executing a taskgroup block, it receives a TASKGROUPEND callback, which is illustrated in Algorithm 8 lines 9-14. At this point in the execution, all the computation associated with the taskgroup block has completed execution. Hence, the algorithm pops the W-node corresponding to the fragment of execution ending at the end of the taskgroup block. Lastly, it pops the S-node created at the beginning of the taskgroup construct and pushes a newly created W-node to represent the continuation of the taskgroup block.

Entering/Exiting a Taskwait. When a worker thread encounters a taskwait construct, it receives two callbacks. The first callback, TASKWAITBEGIN, is called when the current task encounters the taskwait construct right before it starts waiting for the completion of its child tasks. The second callback, TASKWAITEND, is called after the wait for the children task's completion is over, and the current task can start executing the next program fragment.

The callback TASKWAITBEGIN performs a single operation, which is popping the W-node corresponding to the fragment of execution before the taskwait construct (line 16 in Algorithm 8). Note that this operation cannot be postponed to execute as the first operation in TASKWAITEND since the current task might be assigned by the OpenMP runtime to execute its child tasks while waiting at the taskwait. Without popping the W-node at the top of the current task's

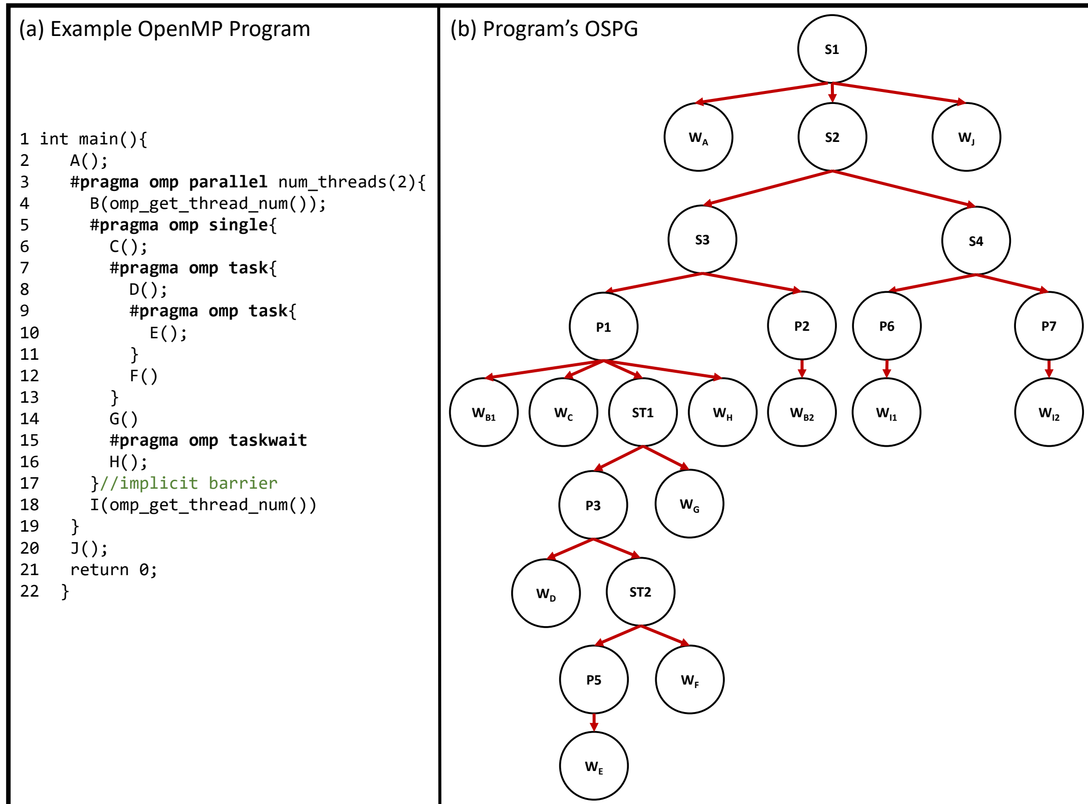


Figure 3.3: An example OpenMP program (a) Task-based OpenMP program. (b) The program's OSPG. Each W-node in the OSPG is named after the corresponding serial function in the program.

associated worker thread, the invariant that only the top node of the per-thread can be a W-node is invalidated by the OSPG construction algorithm.

When the current task finishes waiting at the taskwait, it receives the TASKWAITEND callback to pop the task ST-node added at a previous TASKCREATE callback from its stack (See Algorithm 7).

3.3.2 Example OSPG Construction

We end this section with an example OpenMP program and describe its incremental OSPG construction by applying the OSPG construction algorithm described earlier in Section 3.3.1. Figure 3.3 contains a task-based OpenMP program and its OSPG. The program has one parallel region (lines 3-19) that uses two worker threads. Within the parallel region, a single construct is used to create two explicit tasks (lines 7 and 9). The program uses synchronization primitives in

the form of a taskwait at line 15 and two implicit barriers, one at the end of the single block and one at the end of the parallel region. We assume that all functions in the example program in Figure 3.3(a) named with an upper case letter are serial functions, thus correspond to one W-node in the program's OSPG shown in Figure 3.3(b). Each W-node node in the OSPG is named after its associated function. If a function is called by multiple worker threads (*i.e.*, functions B and I), its W-node name uses the thread identifier to distinguish between its instances.

Figure 3.4 shows the OSPG construction algorithm that constructs the program's OSPG, shown in Figure 3.3(b), incrementally and in parallel. The figure shows the contents of each per-thread stack used by the construction algorithm and how they are updated as the program receives different callbacks to update each thread's view of the OSPG. The example program in Figure 3.3(a) uses two worker threads. The construction algorithm uses thread identifiers to distinguish between threads. Further, as described in Algorithm 7, a newly created explicit task via the task construct has a unique task identifier that is used as a key in a shared map.

When the program in Figure 3.3(a) starts executing the MAIN function, OMP-ADVISER receives a PROFILERSTART callback, creating the per-thread stacks and filling the serial thread's stack with the OSPG root node and the W-node corresponding to the upcoming fragment of execution, function A. Next, the serial thread encounters the parallel region where it updates the serial thread's stack with two S-nodes, S-node $S2$ encoding the serial relation at the end of the parallel region and S-node $S3$ in anticipation of an upcoming barrier, if any. In Figure 3.4(c) both worker threads are about to start executing the first fragment in the parallel region (*i.e.*, function B). Thread2 makes a local copy of S-nodes $S2$ and $S3$ and pushes them unto its stack. Next, worker thread creates a P-node and the W-node corresponding function B.

Next, shown in Figure 3.4(d), thread 1 reaches the single construct first, receiving a SINGLE-BEGIN callback. thread 1 updates its stack to include the W-node corresponding to the first code fragment in the single block (*i.e.*, function C). Once thread1 finishes executing function C, it creates a new explicit task (line 7 program in Figure 3.3(a)). This triggers the TASKCREATE callback, adding a P-node with the new tasks' identifier as the key in the task map. After this event, thread1 is going to execute the program continuation after the task block (*i.e.*, starting with the G function at line 14). This behavior is reflected in the OSPG construction algorithm in Figure 3.4(e) by adding W-node W_G to thread1's stack.

Since thread1 was chosen to execute the single block, it indicates that thread2 will skip over the single block and encounters the implicit barrier at the end of the single block at line 17. At this point in the execution trace, thread2 receives a BARRIERENTER callback (Figure 3.4(f)). The OSPG construction algorithm pops the W-node corresponding to function B from its stack. Further, after encountering the barrier, thread2 becomes eligible to run pending explicit tasks. In our example, thread2 starts executing the recently created task by thread1. Thus, in Figure 3.4(g), thread2 retrieves the task's P-node from the task map and updates with the W-node representing function D. The use of the task map and pushing the P-node to the worker thread that executes the task, rather than the thread that creates the task is essential in correctly measuring the amount of work performed in the task.

Concurrently, thread1 finishes execution function G and encounters the taskwait synchronization construct, which causes thread1 to wait until the child task executing at thread2 to complete execution (Figure 3.4(h)). While waiting, thread1 is eligible to execute any pending tasks. However, no pending tasks exist until thread2, In Figure 3.4(i), creates the explicit task at line 9 in Figure 3.3(a). Thread1 starts executing the newly created task, updating its stack, as shown in Figure 3.4(j). In figures Figure 3.4(k) and (l), both threads finish executing the explicit OpenMP tasks. At this point, thread1 exits the taskwait and continues executing the code fragment that comes after it (*i.e.*, function H). Meanwhile, thread2 continues to wait at the barrier for thread1. Note that in Figure 3.4(m), when thread1 exits the taskwait, the OSPG construction algorithm updates the `st_val` of the ST-node $ST1$. This operation is used to encode the encounter of a taskwait in the program. This information is later used by the parallelism profile computation algorithm, described in Section 3.6, to correctly compute the critical path in the presence of taskwaits in the program trace.

Once thread1 enters the barrier at the end of the single block, the execution can proceed since all worker threads have reached the barrier. As shown in Figure 3.4(o), we remove the S-node corresponding to this barrier from the worker threads' stack and add the S-node $S4$ and its thread-local copy in anticipation of the next barrier in the parallel region, if any. After both threads finish executing function I, they reach the end of the parallel region. Figures 3.4(p) and (q) show how the OSPG construction algorithm removes all the remaining S-nodes and P-nodes corresponding to the just-completed parallel region for both threads' stacks. Lastly, after thread1

finishes executing function J and before it finishes execution, the OSPG construction algorithm receives a PROFILEEND callback. OMP-ADVISER, our parallelism profiler, uses this callback to perform cleanup of the per-thread stacks. However, more importantly, depending on the profiling mode, OMP-ADVISER uses this callback to serialize the remaining OSPG for later analysis or uses it to generate the program's parallelism profile.

3.4 Fine-grained Measurements

The performance model used by OMP-ADVISER is comprised of two components. The first component is the program's OSPG. In section 3.3, we described in detail OMP-ADVISER's construction algorithm to incrementally construct a program's OSPG in parallel. The second component consists of fine-grained measurement of the program's computation for each serial fragment of execution. By using both parts, OMP-ADVISER can perform its parallelism computation and what-if analysis.

A key requirement in designing a performance analysis tool is to ensure that the tool is as lightweight as possible. Thus, avoiding perturbation that may result in result in skewed or incorrect results. Similar to performance analysis tools, OMP-ADVISER uses hardware performance counters to perform this type of fine-grained measurement. Hardware performance counters are registers in modern CPUs, designed to count difference hardware metrics with low overhead. During performance collection, OMP-ADVISER uses these hardware counters to measure the work performed in each worker thread of the program by programmatically querying the counters for different performance metrics (*i.e.*, instruction count, cycles, *etc.*)

To programmatically access hardware performance counters in Linux, OMP-ADVISER uses `perf_events`[6] [78] that is part of the Linux kernel since version 2.6.31. The perf interface provides a system call to access different performance metrics referred to as perf events. There are two types of perf events, sampled and counting. Sampled events are generated by sampling different types of measurements at fixed intervals. In contrast, counting events are used to count the aggregate number of a performance metric. Since OMP-ADVISER must measure the amount of computation for each fragment of the program, using sampled

[6] Also referred as Performance Counters for Linux (PCL) and Linux perf events (LPE).

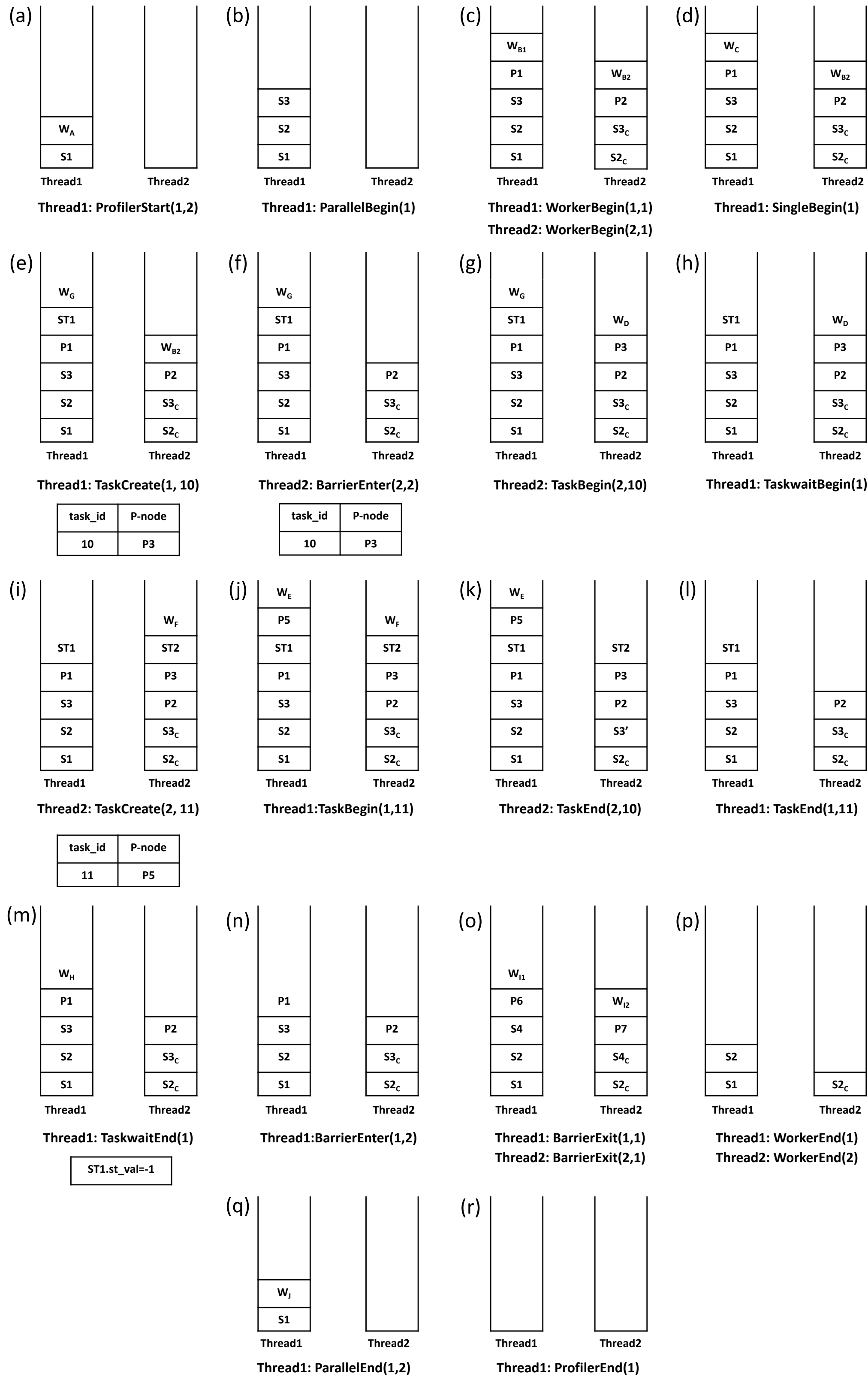


Figure 3.4: Incremental OSPG construction for the example program in Figure 3.3

events may lead to missing measurements if the given fragment ends before the sampling intervals. Thus, OMP-ADVISER uses the perf interface to access counting events. For example to measure the amount of computation as the number of CPU cycles, OMP-ADVISER uses the `PERF_COUNT_HW_CPU_CYCLES` event.

For counting events, the perf interface provides the option of starting, stopping, and resetting the value stored in the hardware performance counter. OMP-ADVISER uses these operations to measure the program's computation by resetting the value and starting the performance counter at the beginning of the fragment and reading its value at the end of a fragment. Alternatively, it is possible to measure a fragment's work without resetting the counter and reading its value at the beginning and end of a fragment. The fragment's work is then measured by subtracting the two values, resulting in lower overheads accessing performance counters by omitting the reset operation. We chose the first option since the second option does not work correctly whenever the performance counter overflows. While it is still possible for a performance counter to overflow even after resetting its value, the program fragment must run for an exceptionally long time to overflow the 64-bit counters available in current CPUs.

Attributing fine-grained measurements to program fragments. OMP-ADVISER's performance model is comprised of the program's OSPG and fine-grained attribution of work measurement to each program fragment. To measure the program's logical parallelism, we need to measure the program's computation and exclude the time spent in the OpenMP runtime (*i.e.*, time spent waiting on synchronization constructs such as barriers). By design, each program fragment is represented by a W-node in the OSPG. When constructing the program's OSPG, a fragment's corresponding W-node's lifetime match each other. Further, during its lifetime, a W-node resides on top of its per-thread stack. One approach to measure each program fragment's computation is to read the performance counters' values whenever its corresponding W-node is pushed and popped from the per-thread stack. The difference between the two values read indicates the computation performed by a program fragment.

Figure 3.5 illustrates the operations used by OMP-ADVISER to attribute work to each W-node. During OSPG construction, pushing a newly created W-node to the top of the per-thread stack denotes the start of a new fragment of execution. To measure the computation performed by the new fragment and exclude the computation performed in the runtime, we reset and start the

(a) Performance counter operations after the creation of a W-node Wi at thread j

```
nodes[j].push(Wi)
counter.reset()
counter.start()
return
```

(b) Performance counter operations after W-node Wi at thread j completes execution

```
c = counter.read()
Wi = nodes[j].pop()
Wi.work = c
return
```

Figure 3.5: Use of performance counters to measure the program's computation and attribution of work to corresponding W-node during OSPG construction.

performance counter at the start of a new program fragment right before OMP-ADVISER returns the execution to the program under test (Figure 3.5(a)). Alternatively, whenever a fragment ends, we read the performance counter and attribute it as the work of the newly popped W-node (Figure 3.5(b)). To maintain the invariant of having a W-node at the top of the per-thread stack during program execution, OMP-ADVISER creates a new W-node and pushes it to the stack before returning the execution to the program under test.

3.5 Estimating Runtime Parallelization Costs

Logical parallelism and critical path measurement in a program provide a hardware-agnostic view of the program's serialization bottlenecks. However, when running an application on actual hardware with a finite number of parallel execution units, there are runtime costs associated with executing and orchestrating the parallel execution of OpenMP constructs. Thus, to achieve scalable speedup, it is essential to ensure that the chosen parallelism granularity ensures that the amount of computation in each W-node amortizes the runtime costs to create and execute its corresponding OpenMP construct.

In practice, the developer has to spend time and effort to manually identify the right parallelism granularity for their programs to achieve good performance. Parallelism granularity and the parallelism of a program directive are related. Consider the program in Figure 3.2, the

`treeProcess` function is using the divide-and-conquer paradigm to process the nodes in a binary tree in parallel by creating new OpenMP tasks to solve the sub-problems in parallel. The programmer uses a `CUT_OFF` value in line 5 to adjust the granularity of work performed in each task. If we increase the `CUT_OFF` value by one, it results in doubling the amount of created tasks while reducing the amount of work in each leaf task by half. To achieve good performance, the developer manually chooses a `CUT_OFF` value that strikes the right balance between the program's parallelism and parallelism granularity. Thus, resulting in a manual trial and error process.

OMP-ADVISER provides actionable feedback based on the runtime costs incurred by the OpenMP runtime when creating dynamic instances of parallelism constructs. Specifically, we augment OMP-ADVISER's performance model by measuring the estimated cost of task creation directives (*i.e.*, task) in the program under analysis and attributing them to corresponding internal OSPG nodes. We then summarize this information in the program's parallelism profile by aggregating the tasking overhead to each static tasking directive in the program and reporting it back to the user.

By default, OMP-ADVISER measures the runtime cost associated with task creation constructs. While it is similarly possible to estimate the cost of worksharing constructs in an OpenMP program, task creation and scheduling costs are significantly higher than creating worksharing constructs [74]. Further, task-based regions in the OpenMP applications tend to have higher nesting depths than its worksharing parts, which results in higher runtime overheads if the wrong task granularity is chosen.

Measuring task creation overheads. An OpenMP task consists of a sequence of instructions. During execution, a task is comprised of an instance of executable code and a data environment that is scheduled by the OpenMP runtime for execution on any worker thread. Thus, the runtime overheads associated with tasking consist of two main components - task creation and scheduling. OMP-ADVISER estimates the task creation costs.

During program execution, whenever a task construct is encountered, the program enters the OpenMP runtime to create a task instance and its data environment. Next, the task instance is passed on to the runtime scheduler to be scheduled for execution before exiting the runtime to continue executing the user's code. To estimate task creation costs, we instrument the OpenMP

runtime to measure the computation costs from when the program enters the runtime to create a task instance until it returns to the user's application.

By default, we use hardware performance counters to measure this cost in terms of hardware cycles. This reliance on hardware counters results in additionally querying hardware performance twice for each newly created task. In our experiments, this additional overhead has been small enough not to perturb the OpenMP program under analysis.

Attributing Task Creation Measurements to OSPG Nodes. After measuring the cost of creating a new task, we need to attribute it to the corresponding OSPG node. For any newly created tasks, as described in Section 2.4.6, a new P-node is added to the program's OSPG. We attribute the creation costs of the newly created task to this P-node. Similarly, for any encountered taskgroup constructs, a new S-node is added to the program's OSPG. We use this S-node to attribute the taskgroup's cost.

Figure 3.2(b) illustrates the attribution of task creation costs to the OSPG of the example program from Figure 3.2(a) by depicting the task creation cost in a dashed rectangle next to the corresponding S-node or P-node. The program creates four task instances, which their costs are attributed to P-nodes [$P2$–$P5$]. Further, the cost of the program's taskgroup construct is attributed to the S-node $S3$.

3.6 Parallelism Profile Computation

OMP-ADVISER uses hardware performance counters to perform fine-grained measurement of the program's computation and attributes them to each W-node in the program's OSPG. This is a performance model from the perspective of computing the logical parallelism for a program and its input. OMP-ADVISER summarizes the parallelism and critical path captured by the performance model at each static OpenMP directive in the program. We refer to this report as the program's parallelism profile.

Definition 3.6.1. (**Parallelism Profile**) A program's parallelism profile summarizes the information captured in the program's OSPG and fine-grained measurements at a static program location (*i.e.*, per static OpenMP directive or the main function in the program) granularity. Each entry in the parallelism profile is comprised of the following members:

- **Location.** Represents a static program location where a location includes the OpenMP construct's filename followed by its line number in the program's source.
- **Work.** Denotes the aggregate computation attributed to the location.
- **Serial work.** Denotes the computation of the longest sequence of serial computation for a given location.
- **Parallelism.** Is the ratio of work over serial work which represents an upper bound on the region's speedup.
- **Runtime overhead.** Is the percentage of estimated OpenMP runtime costs over the total work attributed aggregated at a given location.
- **Serial work percentage.** this value denotes the location's contribution to the program's critical path. If zero, the corresponding program trace does not lie on the program's critical path.

To generate the program's parallelism profile, OMP-ADVISER operates in two modes, each with its set of trade-offs. Offline mode and on-the-fly mode. In both modes, OMP-ADVISER runs in conjunction with the program under test to incrementally construct OSPG and perform measurements. In offline mode, the performance model is written to log files whenever an OpenMP construct finishes execution (*i.e.*, the corresponding OSPG nodes are popped from the per-thread stack). Later, an analysis program uses the log files to reconstruct the entire OSPG in memory and generates the program's parallelism profile. Compared to on-the-fly mode, offline mode uses less memory during profiling since it can deallocate OSPG nodes as soon as a node is popped from a thread's stack. Further, since the offline analysis tool has access to the program's entire OSPG, what-if analysis (see Section 3.7) can be performed without the need to re-execute the program under test. This mode's main drawback is that the log files for long-running programs can become large, limiting its usability for long-running applications.

When using the on-the-fly profiling mode, the parallelism profile is generated on-the-fly while executing the program under test. On-the-fly profiling mode does not require access to the program's entire OSPG to compute the parallelism profile. Hence it can be used to analyze long-running applications and applications with large OSPGs that do not fit inside the system's

memory. However, in this mode, OMP-ADVISER maintains additional state with each OSPG node during program execution to compute its parallelism on-the-fly. Thus, compared to the offline mode's profiling run, it uses more memory for each OSPG node created. Additionally, performing multiple what-if analysis queries requires the program to be re-executed.

This section first provides the offline mode's details since it is less complicated than the on-the-fly mode. Next, we describe the on-the-fly mode's details, addressing the challenges of adapting the offline analysis algorithm to work in an online setting.

3.6.1 Offline Analysis to Compute Parallelism

A program's parallelism profile consists of the program's logical parallelism, the parallelism of each static OpenMP directive in the program, and its serial work contribution to the program's critical path. To generate the program's parallelism profile, the offline algorithm first performs a bottom-up traversal of its OSPG that computes each node's aggregate work and serial work. The algorithm visits each OSPG node and computes the node's work and serial work from the information stored in its child nodes. Thus at the end of the analysis, the algorithm computes the following two quantities for each OSPG internal node: (1) work (w) and (2) the set of W-nodes that constitute its critical path (S). After computing these quantities, the analysis algorithm summarizes this information to each static OpenMP directive to generate the program's parallelism profile.

Definition 3.6.2. (**Node Work**) An OSPG node's work (w), is defined as follows:

- **Leaf Nodes.** By construction, all leaf nodes in the OSPG are W-nodes. A leaf node's work is equal to the corresponding code fragment's fine-grained work measurement.
- **Internal Nodes.** An internal node's work is the aggregate sum of the work performed by its W-node descendants.

Definition 3.6.3. (**Node Critical Path**) An OSPG node's critical path (S) is the set of W-nodes in its subtree that execute in series and perform the largest amount of work.

Algorithm 9: Generate program's parallelism profile for OSPG G rooted at node R. D is the set of the program's directives used for aggregation. CHILDNODES returns all the child nodes of the input node. COMPUTESERIALWORK identifies the serial work contribution of each static directive to the program's critical path. AGGREGATEPERSTATICDIRECTIVE aggregates work, serial work, and tasking overheads to produce the parallelism profile similar in format to Figure 3.2 (c) and (d).

```
1 function PARALLELISMPROFILE (G, R, D)
2     foreach N in bottom-up traversal of G do
3         ⟨N.w, N.S⟩ ← COMPUTENODE(N)
4     end
5     H ← COMPUTESERIALWORK(G, R, N.S)
6     foreach N in bottom-up traversal of G do
7         AGGREGATEPERSTATICDIRECTIVE(G, H, D)
8     end
9     return
```

Definition 3.6.4. (**Node Serial Work**) An OSPG node's serial work (sw), is defined as the amount of work performed on its critical path. It can be computed by summing the work of its critical path W-node members.[7].

Algorithms 9 and 10 show the steps involved in generating the program's parallelism profile. Algorithm 9 performs a bottom-up traversal of the program's OSPG nodes and computes each node's work ($N.w$) and the set of W-nodes that constitutes its critical path ($N.S$) by invoking the COMPUTENODE function shown in Algorithm 10 lines 2-4.

Once this traversal completes, the OSPG's root node R contains the entire program's work in $R.w$. Further, the set of W-nodes that contribute to the program's critical path are stored in $R.S$. We use the results computed during this traversal to calculate the serial work contribution of each static directive to the program's critical path by invoking the COMPUTESERIALWORK function (line 5 in Algorithm 9). The COMPUTESERIALWORK function returns H, a mapping from a program location to its serial work contribution on the critical path. Finally, the algorithm aggregates the information computed in previous steps to generate the program's parallelism profile matching the Definition 3.6.1.

[7]Let N represent an OSPG Node. $N.sw = \sum_{C \in N.S} C.w$.

Computing Each Node's Work and Serial Work. The PROCESSNODE function in Algorithm 10 illustrates the computation of in input node's work and critical path. The PROCESSNODE function is invoked for each OSPG node in a bottom-up traversal. This bottom-up traversal enables the algorithm to compute a node's work and critical path from its children nodes' work and critical path instead of processing the entire subtree rooted at a given node.

The simple case for the algorithm occurs when processing a W-node (lines 2-5 in Figure 9). By construction, all W-nodes are leaf nodes. Thus, we can immediately compute a W-node's work and serial work. To process an internal OSPG node and compute its work and serial work, the algorithm processes its child nodes from left to right, based on the logical ordering of sibling operations (lines 9-40 in Algorithm 9). We describe how an internal OSPG node is processed next.

Consider internal OSPG node N which has k children. Each child node is denoted by $(C_1..C_k)$. In the PROCESSNODE function, the loop at line 10 iterates over all child nodes C_i such that $C_i \in C_{1..k}$. The algorithm maintains the loop invariant that at iteration i, $N.w$ and $N.S$ contain the node's work and critical work when considering its first i child nodes, $(C_1..C_i)$. By initializing $N.w$ and $N.S$ at lines 6-7 in Algorithm 9, the invariant is true before entering the loop. We need to show the invariant is maintained after each iteration of the loop. If the invariant is true after each iteration, at the end of the loop, $N.w$ and $N.S$, will hold the node's work and critical path by considering all of its child nodes, which by definition is the node's work and critical path.

At each iteration of the loop, we increment $N.w$ to include the work of the current child node being examined (line 11 in Algorithm 9). Thus, the invariant is maintained for $N.w$. We use inductive reasoning to show how the loop invariant for the node's critical path is maintained. Our inductive hypothesis assumes that at the start of the $j+1$ iteration, $N.w$ includes node N's critical path candidate comprised of its first j child nodes and their descendants. We want to show that at the end of the iteration, $N.w$ includes node N's critical path candidate comprised of its first $j+1$ child nodes.

Based on our inductive hypothesis, $N.S$ includes the set of W-nodes under the subtree of the first j child nodes that perform the largest amount of work serially. When considering the node's critical path comprised of the first $j+1$ child nodes, two cases may occur. In the first case,

Algorithm 10: Compute input node N's work ($N.w$) and and the set of W-nodes on its critical path ($N.S$).

```
1  function ProcessNode(N)
2     if NodeType(N) = W-node then
3        N.S ← {N}
4        return ⟨N.w, N.S⟩
5     end
6     N.w ← 0
7     N.S ← {}
8     LS ← {}
9     LP ← {}
10    foreach C ∈ ChildNodes(N) do
11       N.w ← N.w + C.w
12       cu ← Σ_{W∈N.S} W.w
13       cd ← Σ_{W∈LS} W.w + Σ_{W∈C.S} W.w
14       switch NodeType(C) do
15          case P-node do
16             if HasDependency(C) = True then
17                DP ← LongestDependenceChain(C)
18                DW ← ImmediateFragments(DP)
19                cd ← cd + Σ_{W∈DW} W.w
20                if cd > cu then
21                   N.S ← N.S \ LP
22                   N.S ← N.S ∪ C.S ∪ DW
23                   LP ← C.S ∪ DW
24             else if cd > cu then
25                N.S ← N.S \ LP
26                N.S ← N.S ∪ C.S
27                LP ← C.S
28          case S-node or W-node do
29             if cd > cu then
30                N.S ← N.S \ LP
31                N.S ← N.S ∪ C.S
32             LS ← LS ∪ C.S
33          case ST-node do
34             ⟨ST_p, ST_s⟩ ← PartitionNodes(C.S)
35             if cd > cu then
36                N.S ← N.S \ LP
37                N.S ← N.S ∪ C.S
38                LP ← ST_p
39             LS ← LS ∪ ST_s
40       end
41    end
42    return ⟨N.w, N.S⟩
```

the critical path does not include any W-node descendants from child node C_{j+1} . in this case, the critical path candidate remains the same as the previous iteration and will not require any changes. In the second case, the critical path includes W-node descendants from the C_{j+1}. In this case, we need to update $N.w$ with the new critical path candidate that performs more serial work. The algorithm maintains the current critical path's serial work in cu and the candidate critical path's serial work in cd (lines 12-13 in Algorithm 9). To handle these two cases, the algorithm compares the cu and cd. If the candidate serial work is larger than the current serial work (check performed at lines 19, 24, 28, and 35 in Algorithm 9), the algorithm updates $N.S$ with the candidate critical path, thus maintaining the loop invariant.

We still need to show how to compute the candidate critical path, cu, at each iteration. To compute the critical path candidate that includes the current child node, C_{j+1}, the algorithm maintains two additional sets of W-nodes. These sets, named LS and LP, are used to compute the critical path candidate without the need to reexamine the descendant nodes of the first j child nodes at each iteration. The first set, LS contains the set of W-node descendants of already examined child nodes (*i.e.*, nodes $C_1..C_j$) that execute in series with current child node's C_{j+1} descendant W-nodes. Intuitively, node N's critical path candidate is comprised of the union of LS and the current child node's critical path (*i.e.*, $C_{j+1}.S$). The second set, LP, contains the set of W-nodes that are descendant of previously examined P-node and ST-node child nodes, that lie on the currently computed critical path, $N.S$, and run in parallel with the current child node's descendant W-nodes. This set is used to update the current node's critical path whenever the critical path candidate performs more work than the current critical path. As the loop executes, these sets are updated based on the type of the current child node.

Based on OSPG node definitions in Section 2.3, in the absence of task dependencies, a P-node encodes a parallel relation between its W-node descendants and right sibling descendant W-nodes. In contrast, an S-node encodes a serial relation between its W-node descendants and right sibling descendant W-nodes. A W-node executes in series with all descendants W-nodes to its right. Lastly, an ST-node partition its descendant W-nodes into two sets, the former executing in series and the latter executing in parallel relative to the W-node descendants of its right siblings, which is slightly more complicated than the first three types of nodes. Algorithm 9 uses the properties of the currently examined child node to maintain the loop invariant. To

ease explanation, we first explain the algorithm when the program under test does not use task dependencies (*i.e.*, omitting lines 16-23 from Algorithm 9). In such cases, if the program critical path candidate includes the descendants of a P-node or parallel partition of an ST-node, it at most can contain one such P-node or ST-node at a time. For example, consider two sibling P-nodes, without task dependencies, their descendant W-nodes execute in parallel. Hence, if their parent node's critical path contains W-nodes descendants of either P-nodes, it cannot contain any W-nodes descendants from the other. Thus, the set LP is updated such that it contains up to one P-node or S-node descendant W-nodes at a time.

When encountering a child P-node that replaces the current critical path (lines 24-27 in Algorithm 9), the algorithm updates the parent node's critical path candidate, $N.S$, by first removing the elements of set LP from it and adding the current child P-node's critical path to its critical path (line 21-22 in Figure 9). Intuitively, all W-nodes in the set LP execute in parallel with the current child P-node's descendant W-nodes that comprise its critical path, $C.S$. Hence, $N.S$ is updated by removing the elements of LP and adding the elements $C.S$. Lastly, since the newly updated critical path candidate includes the current child P-node descendant W-nodes, LP is set to the current child P-node's critical path.

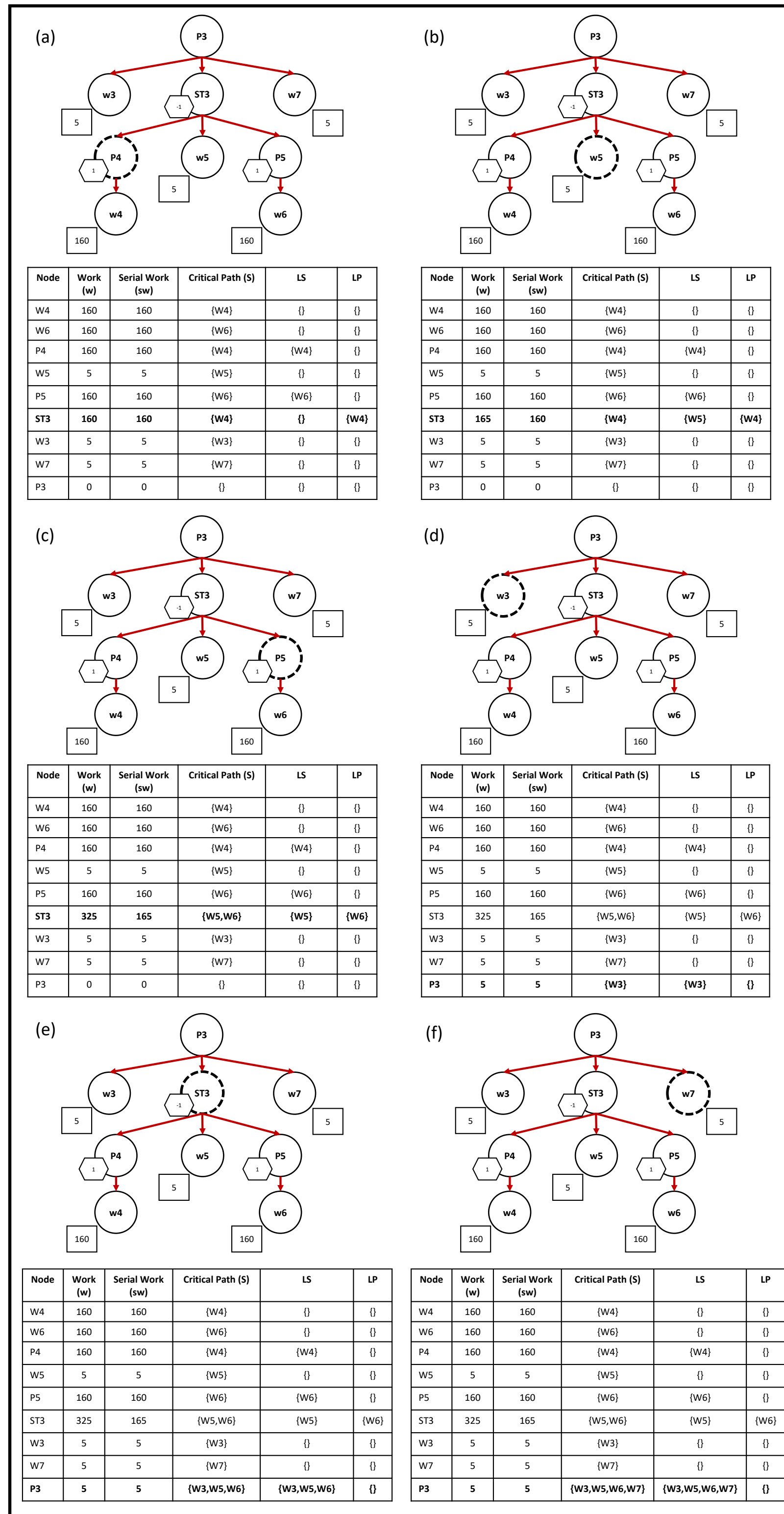
When processing a child S-node or W-node, if the candidate critical path, including the current child node, performs more work than the current critical path, it indicates that the algorithm has found a new critical path that does not include the elements of the set LP. Hence, similar to the previous case, $N.S$ is updated by removing the elements of LP and adding the elements $C.S$. Unlike P-nodes, an S-node or a W-node encodes a serial relation with the W-node descendants of its right siblings. Thus, the algorithm updates the set SP instead of LP (line 32). The algorithm updates SP by adding the critical path of the current child node to the set of W-nodes that execute in series with the W-node descendants of right siblings nodes examined in later loop iterations.

Compared to the previous cases, processing an ST-node node is slightly more involved (lines 33-39 in Algorithm 9). As discussed in Section 2.3, unlike P-nodes, S-nodes, or W-nodes, the W-node descendants of an ST-node may execute in series or parallel with its right sibling descendant W-nodes. The series-parallel relation of a W-node under the subtree of an ST-node and the W-node descendants of its right siblings depends on the nesting level and the sum of

`st_val` values on the path from the W-node to the ST-node. Thus an ST-node partitions the set of W-nodes under its subtree into two sets. A set of W-nodes that execute in parallel and a set of W-nodes that execute in series relative to the W-node descendants of the ST-node's right siblings. The algorithm calls the PARTITIONNODES function to retrieve the two partitions as ST_p and ST_s for the ST-node's critical path, $C.S$ (line 34). Intuitively, the algorithm treats the ST_p set as the descendant nodes of a P-node and ST_s as the descendants of an S-node. Hence, the rule to update the set LP and LS is similar to updating it for child P-nodes and child S-nodes, respectively.

Compute Critical Path in the Presence of Task Dependency. OpenMP task dependency clauses are used to specify a serial ordering between sibling tasks. In the absence of task dependencies, all sibling tasks logically execute in parallel. Thus when computing the critical path in the presence of task dependencies, we can no longer assume that a node's critical path can only pass through at most one child P-node. Instead, to compute the serial work for a P-node, we need to consider the serial work of all left P-node siblings that it depends upon since the child P-node cannot begin execution until all of its dependent P-nodes complete execution. Further, task dependencies are transitive. Thus, when computing a child P-node's serial work, we need to identify its longest chain of task dependencies (*e.g.*, if P3 depends on P2, and P2 depends on P1, then the longest chain of dependency is P3, P2, and P1). We handle this case in lines 16-23 in Algorithm 9. We use the helper function LONGESTDEPENDENCECHAIN to return the set DP, the child P-node's longest chain of dependent P-nodes. Since task dependencies only imply a serial ordering between the left P-nodes child fragment W-nodes and not all the W-nodes in its subtree (See example program in section 2.13), the algorithm uses another helper function IMMEDIATEFRAGMENTS to return the set of child W-nodes of the P-node depends upon as a result of task dependencies (line 17-18), name DW. Next, the algorithm updates the serial work candidate, cd, by adding the sum of work of W-nodes in the DW set to it (line 19). Consequently, if the algorithm updates the node's critical path, the algorithm adds members of DW to $N.S$ (line 22 in Algorithm 9).

Illustrative Example. Figure 3.6 illustrates the steps involved to compute the work, serial work, and critical path of OSPG nodes. The Figure highlights the bottom-up traversal of the OSPG subtree rooted at P-node $P3$ from the example OSPG in Figure 3.2(b) when executing



(a)

Node	Work (w)	Serial Work (sw)	Critical Path (S)	LS	LP
W4	160	160	{W4}	{}	{}
W6	160	160	{W6}	{}	{}
P4	160	160	{W4}	{W4}	{}
W5	5	5	{W5}	{}	{}
P5	160	160	{W6}	{W6}	{}
ST3	**160**	**160**	**{W4}**	**{}**	**{W4}**
W3	5	5	{W3}	{}	{}
W7	5	5	{W7}	{}	{}
P3	0	0	{}	{}	{}

(b)

Node	Work (w)	Serial Work (sw)	Critical Path (S)	LS	LP
W4	160	160	{W4}	{}	{}
W6	160	160	{W6}	{}	{}
P4	160	160	{W4}	{W4}	{}
W5	5	5	{W5}	{}	{}
P5	160	160	{W6}	{W6}	{}
ST3	**165**	**160**	**{W4}**	**{W5}**	**{W4}**
W3	5	5	{W3}	{}	{}
W7	5	5	{W7}	{}	{}
P3	0	0	{}	{}	{}

(c)

Node	Work (w)	Serial Work (sw)	Critical Path (S)	LS	LP
W4	160	160	{W4}	{}	{}
W6	160	160	{W6}	{}	{}
P4	160	160	{W4}	{W4}	{}
W5	5	5	{W5}	{}	{}
P5	160	160	{W6}	{W6}	{}
ST3	**325**	**165**	**{W5,W6}**	**{W5}**	**{W6}**
W3	5	5	{W3}	{}	{}
W7	5	5	{W7}	{}	{}
P3	0	0	{}	{}	{}

(d)

Node	Work (w)	Serial Work (sw)	Critical Path (S)	LS	LP
W4	160	160	{W4}	{}	{}
W6	160	160	{W6}	{}	{}
P4	160	160	{W4}	{W4}	{}
W5	5	5	{W5}	{}	{}
P5	160	160	{W6}	{W6}	{}
ST3	325	165	{W5,W6}	{W5}	{W6}
W3	5	5	{W3}	{}	{}
W7	5	5	{W7}	{}	{}
P3	**5**	**5**	**{W3}**	**{W3}**	**{}**

(e)

Node	Work (w)	Serial Work (sw)	Critical Path (S)	LS	LP
W4	160	160	{W4}	{}	{}
W6	160	160	{W6}	{}	{}
P4	160	160	{W4}	{W4}	{}
W5	5	5	{W5}	{}	{}
P5	160	160	{W6}	{W6}	{}
ST3	325	165	{W5,W6}	{W5}	{W6}
W3	5	5	{W3}	{}	{}
W7	5	5	{W7}	{}	{}
P3	**5**	**5**	**{W3,W5,W6}**	**{W3,W5,W6}**	**{}**

(f)

Node	Work (w)	Serial Work (sw)	Critical Path (S)	LS	LP
W4	160	160	{W4}	{}	{}
W6	160	160	{W6}	{}	{}
P4	160	160	{W4}	{W4}	{}
W5	5	5	{W5}	{}	{}
P5	160	160	{W6}	{W6}	{}
ST3	325	165	{W5,W6}	{W5}	{W6}
W3	5	5	{W3}	{}	{}
W7	5	5	{W7}	{}	{}
P3	**5**	**5**	**{W3,W5,W6,W7}**	**{W3,W5,W6,W7}**	**{}**

Figure 3.6: Execution of the PROCESSNODE when computing the work, serial work, and critical path on the OSPG subtree from Figure 3.2. The node currently being processed is highlighted by bold text in the table. The child node being examined is shown with dashed circles in the Figure.

PROCESSNODE on each node. The table in each subfigure shows the quantities that the algorithm updates after a call to the PROCESSNODE function. For brevity, the steps to process W-nodes and internal nodes with one child node are omitted. For these nodes, their critical path is comprised of the W-node or their single W-node child node.

Figures 3.6(a-c) show the steps to compute work, serial work, and critical path of the ST-node $ST3$ by processing its child nodes from left to right. As each child node is processed, the work and the critical path of $ST3$ is updated. In Figure 3.6(a), the algorithm updates the ST-node's critical path, $ST3.S$, to include the serial work of $P4$. Next, when processing W-node $W5$, the algorithm does not update the ST-node's critical path since W-node $W5$ executes in parallel relative to $P4$. When processing child node $P5$ in Figure 3.6(c), the critical path of $ST3$ is updated since the critical path candidate comprised of node $P5$ and its union with $W5$ performs more work than the current critical path stored in $ST3$. Thus, after processing all child nodes of $ST3$, its work, serial work, and critical path is correctly computed by the PROCESSNODE function. Similarly, Figures 3.6(d-f) show how the PROCESSNODE function computes the work and serial work on node $P3$ by processing its child nodes.

Generating the Program's Parallelism Profile.

Once we compute the work and critical path of all OSPG nodes, the root node's critical path $R.S$ consists of the W-nodes that constitute the program's critical path. To generate the program's parallelism profile, we need to summarize this information to static locations. Generally, this aggregation can be performed at different granularity levels. The two common options are aggregating per static location of the OpenMP parallelism directives used in the program or the calling context of each program fragment. In both cases, we need to maintain additional location information for the program's OSPG nodes. For OMP-ADVISER, we opted for the former option since it requires maintaining a constant amount of information (*i.e.*, filename and line number) per internal OSPG node. However, the AGGREGATEPERSTATICDIRECTIVE function used in Algorithm 9 can be adopted to aggregate at other granularities. For example, aggregation at a calling context granularity is possible if we maintain the calling context of each program fragment, which can be accomplished by using a stack unwinding library whenever a new W-node is created.

Computing serial Work per Static Directive. When constructing a program's OSPG, as described in Chapter 2, whenever the program encounters an OpenMP parallelism directive (*i.e.*, `parallel`, `for`, `task`, *etc.*), it creates one or more internal OSPG nodes. We attribute the location of the directive to the first OSPG node that is created in response to the encountered directive. If a W-node represents a program fragment outside all OpenMP parallel regions, we attribute it to an entry representing the program's main function.

Algorithm 11 takes the program's critical path, CP, and the set static locations, LC, and computes the critical path contribution of each program directive by returning map H, which maps each static location to the amount of work it performs on the critical path. The algorithm first initializes the entries of map H to zero by iterating over the set of static locations, LC. Next, the algorithm iterates over each W-node on the program critical path and attributes its work to the corresponding entry in H. To find which location a W-node corresponds to, the algorithm examines the nodes on the path towards the root node. We attribute the W-node's work to the location of the first internal node visited that has a non-empty location, ensuring that the W-node is not attributed to more than one location. Intuitively, the W-node resides under the subtree of the encountered internal node. Hence it represents a fragment of dynamic execution within the body of the location it corresponds to (lines 7-13 in Algorithm 11. The OSPG root node corresponds to the location of the main function in the program. Thus, the algorithm ensures that each W-node will be attributed to exactly one location.

Aggregating per Static Directive. The final step in generating the program's parallelism profile is to aggregate work, serial work, and task creation overheads to each static directive in the program. The main challenge to correctly perform this aggregation is that we must ensure we do not attribute the measurements corresponding to a dynamic instance of an OpenMP directives to more than one static directive. For example, in a program that uses nested tasks, we must be careful not to double count work and serial when the subtree under the current node includes another node with the same directive location information.

Similar to Algorithm 11, the AGGREGATEPERSTATICDIRECTIVE function (line 7 in Algorithm 9) maintains a map with each static directive entry as its set of keys. In contrast, it maps each static entry to its work, serial work, and aggregate task creation overhead. To update the values in the map, the algorithm performs another bottom-up traversal of the program's OSPG.

Algorithm 11: Compute the critical path contribution of each static location in the program

```
1  function ComputeSerialWork(G, R, CP, LC)
2      foreach L ∈ LC do
3          H(L) ← 0
4      end
5      foreach N ∈ CP do
6          w ← N.w
7          do
8              N ← Parent(N)
9              L ← Loc(N)
10             if L ≠ ∅ then
11                 break
12         while N ≠ ∅
13         H(L) ← H(L) + w
14     end
15     return H
```

For work and serial work values, whenever an internal node with a corresponding static directive location is encountered, we update the map's entry by adding the node's work and serial work if it is not subsumed by another node with the same location information. When examining a node, a simple way to perform this check is to check for a node with the same location information on the path to the root node. If such a node exists, the algorithm defers adding the work and serial work value until the ancestor node is visited later in the bottom-up traversal of the program's OSPG. In contrast, aggregating the tasking overhead per static directive is simpler. We increment the corresponding entry in the map during the bottom-up traversal for every internal node with a tasking overhead measurement.

3.6.2 On-the-fly Profiling Mode to Compute Parallelism

In the on-the-fly profiling mode, OMP-ADVISER generates the program's parallelism profile based on Definition 3.6.1 without the need to generate log files and using additional analysis mode to generate the parallelism profile. The main advantage of the on-the-fly mode over the offline mode is that it does not need the entire OSPG to compute the program's parallelism profile.

To compute the program's parallelism profile on the-the-fly, we need to devise solutions to the following challenges:

1. In an online setting, nodes may be removed from the OSPG as soon as the computation under their subtree finishes. Further, for parallel fragments of execution, their corresponding nodes may appear in different valid ordering in each execution of the program.

2. To generate the parallelism profile, we need to track the program's critical path. However, in an online setting, it is not clear a priori which fragments contribute to the program's critical path. Hence the algorithm should be able to deduce the critical path as the program continues execution.

We devise two different solutions to address these challenges for on-the-fly profiling that each have their trade-offs. The first solution supports the same programs supported in the offline profiling mode, yet it requires keeping OSPG nodes in memory longer than the second solution. The second solution deallocates OSPG as soon as they go out of scope[8]. However, this solution supports a subset of OpenMP programs that do not have any ST-nodes in their OSPG (*i.e.*, they do not use taskwaits). This requirement simplifies identifying the program's critical path when having access to only a small slice of the program's OSPG.

Online Analysis in the Presence of ST-nodes

This solution is highly inspired by the offline profiling mode algorithm described in Section 3.6.1. It addresses the first challenge caused by profiling in an online setting by keeping the entire subtree representing the current parallel region in memory until the parallel region finishes execution, which delays OSPG node removal until the corresponding parallel region completes execution.

The behavior of ST-nodes necessitates the need to keep OSPG nodes in memory until all computation under its subtree completes execution. Unlike S-nodes, P-nodes, or W-nodes where they establish either a series or a parallel relation between the W-nodes in their subtree and the descendant W-nodes of their right sibling, for a descendant W-node on an ST-node, its nesting

[8] *i.e.*, The node is popped from the per-thread stack as described in Section 3.3.1

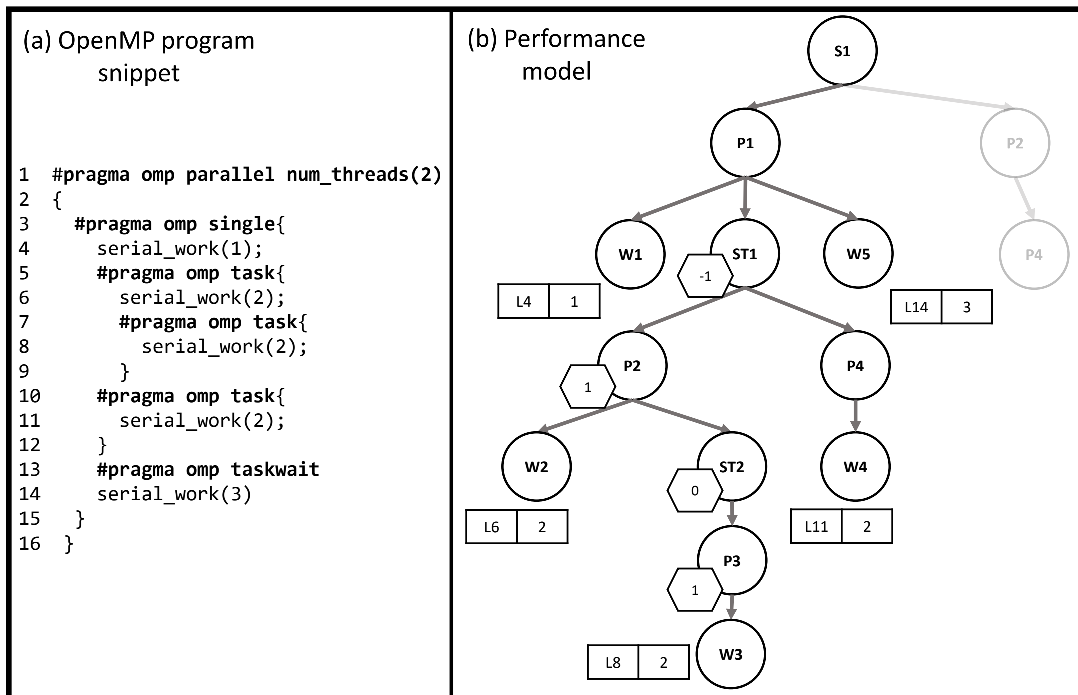


Figure 3.7: (a) An OpenMP program snippet that uses tasking and taskwaits. (b) The program's performance model. the table under each W-node contains the corresponding line in the program and its amount of work. the `st_val` of nodes is shown with a hexagon each to relevant nodes.

level and the encountering of taskwaits in ancestor tasks changes the series-parallel relation between it and the descendant W-nodes the ST-node's right sibling.

Thus, to correctly identify the critical path of an internal OSPG node with ST-node descendants, we need to maintain its subtree until all its W-node fragments complete execution. After this condition is met, we can be certain that the `st_val` of ST-nodes may no longer change, and the series-parallel relation between all W-nodes remains constant. Thus, we can safely process each internal OSPG node and compute the parallelism profile. The example, Figure 3.7 shows an example OpenMP code snippet and its performance model where removing the subtree under the P-node $P1$ as soon as its child ST-node completes execution will result in computing the wrong critical path for an internal node in the OSPG.

First consider computing the work and critical path of P-node $P1$ using the offline mode (see Algorithm 10). The algorithm processes node $P1$ child nodes from left to right, and determines that the critical path of node $P1$ is comprised of the set of W-nodes $\{W1, W2, W5\}$ performing five units of work. Specifically, the algorithm identifies that even if W-node $W3$ is part ST-node

$ST1$ critical path, it is not a part of node $P1$ critical path since W-nodes $W3$ and $W5$ execute in series. Now consider an online algorithm that keeps track of $P1$ children work and critical path by storing them in $P1$ as soon as their subtree completes execution. For some program traces the subtree under ST-node $ST1$ completes execution before the fragment corresponding to W-node $W5$ completes execution. When node $ST1$ finishes execution we store its critical path, $\{W2, W3\}$, in its parent node $P1$. Later in the execution trace, $W5$ completes execution. The information stored is $P1$ is insufficient to determine the series-parallel relation between W-nodes $W5$ and $\{W2, W3\}$. If we simply assume that they execute in series, the online algorithm generates an incorrect critical path $\{W1, W2, W3, W5\}$ performing eight units work for P-node $P1$. However, if we maintain the subtree under ST-node $ST1$, an online algorithm will have enough information to correctly deduce the critical path.

Our online solution in the presence of ST-nodes takes a straightforward approach to generate the program's parallelism profile. The algorithm steps are listed as follows:

- Construct the program's OSPG incrementally based on the description in Section 3.3.
- After executing a parallel region, perform the offline analysis algorithm on the entire subtree corresponding to the completed parallel region. Summarize the information in the subtree by updating the entries of a map (Similar to the map in Section 3.6.1). Subsequently, deallocate the entire OSPG subtree.
- Repeat the above steps until the program completes execution. Use the map to generate the parallelism profile.

Compared to offline profiling, the extra computation to update the parallelism profile at the end of each parallel region adds additional run-time overheads. However, since this computation is performed after the parallel region ends, it does not impact worker threads' parallel execution. This additional computation appears as if the barrier at the end of the parallel region has a long waiting duration in the program's execution trace. Thus, our proposed on-the-fly profiling addresses the three challenges mentioned with profiling in an online setting.

Online Analysis in the absence of ST-nodes

In programs with OSPG's that contain no ST-nodes, a node's serial work can be deduced from its child nodes' serial work computation. This property holds since unlike ST-nodes, the logical series-parallel relations between the W-node descendants of S-nodes and P-nodes, and the descendant W-nodes of their right siblings, once established, does not change. Hence, we can leverage this property to devise an on-the-fly profiling algorithm that can deallocate OSPG nodes as soon as they complete execution. Effectively, the profiling algorithm piggybacks on the OSPG construction algorithm described in Section 3.3 by keeping additional state for each OSPG node and updating them whenever the nodes are pushed and popped from the per-thread stack used by OMP-ADVISER to construct the program's OSPG.

In the absence of ST-nodes, we claim that the problem of computing the critical path of an internal OSPG node from its children nodes has the optimal substructure property. Hence, enabling a solution that can incrementally construct the node's critical path while examining its child nodes from left to right. Recall that a node's critical path is the set W-nodes in its subtree that execute serially and perform the largest amount of work. In the absence of task dependency, a parent's node critical path has two forms: (1) it does not include any W-node descendant from its child P-nodes. In this case, the parent node's critical path is comprised of the union of the critical path of all its S-node and W-node children and (2) it includes one child P-node in its critical path. In this scenario, the parent's node critical path is the union of the P-node's critical path with the critical path of all left sibling S-nodes and W-nodes.

Based on OSPG nodes properties, in the absence of task dependency, two sibling P-nodes execute in parallel. Hence the parent node's critical path can only include the descendants of one child P-node in its critical path. Algorithm 12 uses the above properties to compute a node's critical path with a single left to right scan of its child nodes. Each node maintains three sets, LW is the set of the current child node's left W-node siblings. LS is the set of W-node descendants on the critical path of left S-node siblings. Lastly, CP is the set of the W-node that constitute the current critical path candidate comprised of all already processed child nodes. At each iteration, if the critical path candidate that includes the current child node performs more

Algorithm 12: Algorithm to compute an internal OSPG node's critical path form its child nodes

```
1  function FastProcessNode (G, N)
2      N.LW ← {}
3      N.LS ← {}
4      N.CP ← {}
5      foreach C ∈ ChildNodes(N) do
6          cu ← Σ_{S∈N.CP} S.w
7          cd ← Σ_{S∈N.LS} S.w + Σ_{S∈N.LW} S.w + Σ_{S∈C.CP} S.w
8          if cd > cu then
               ▷ update critical path
9              N.CP ← N.LW ∪ N.LS ∪ C.CP
10         switch NodeType(C) do
11             case S-node do
12                 N.LS ← N.LS ∪ C.CP
13             case W-node do
14                 N.LW ← N.LW ∪ C.w
15         end
16     end
17     return
```

work than the previously-stored critical path, the algorithm updates the current critical path with the new candidate. At the end of the loop, the set CP will contain the node's critical path.

Algorithm 12 assumes that the child nodes can be examined based on the left-to-right ordering of OSPG nodes. We can use this algorithm in an online setting if we restrict OMP-ADVISER online profiling to execute the application under test serially. However, executing the application under test serially does not utilize the underlying hardware parallelism and can result in high overheads. When an application executes in parallel, depending on the program's schedule and underlying hardware parallelism, sibling P-nodes may run in parallel. Thus sibling P-nodes can complete in any permutation and go out of scope. Therefore, we need to adapt Algorithm 12 for use in parallel execution of the application.

In addition, our online profiling algorithm must update the per static location map entries with the amount of work, serial work, and tasking overhead measurements. Our online profiling algorithm (shown in Algorithm 13) piggybacks on OMP-ADVISER's OSPG construction to compute the program's parallelism profile on-the-fly. Whenever the construction algorithm

Algorithm 13: On-the-fly profiling mode algorithm for program's with no ST-nodes. The function PUSHNODE occurs when adding a new OSPG node N as the child node of P. The function POPNODE is called after the entire subtree under node N completes execution and node N updates its parent node's attributes before getting deallocated.

```
1  procedure PUSHNODE(P, N, t)
2      N.w ← 0
3      N.t ← t
4      N.LSW ← {}
5      N.CP ← {}
6      N.PSW ← P.LSW
7  procedure POPNODE(P, N)
8      switch NODETYPE(N) do
9          case W-node do
10             N.w ← READCOUNTER()
11             N.CP ← {N}
12             P.LSW ← P.LSW ∪ {N}                    // atomic update
13         case S-node do
14             P.LSW ← P.LSW ∪ N.CP                   // atomic update
15         case P-node do
16             L ← TASKLOC(N)
17             T(L) ← T(L) + N.t                      // atomic update
18     end
19     P.w ← P.w + N.w                                // atomic update
20     P.CP ← max(P.CP, N.CP ∪ N.PSW)          // atomically update
        parent's critical path
21     if LOC(N) ≠ ∅ then
22         D ← D ∪ LOC(N)
23         AGGREGATENODEPERSTATICDIRECTIVE(N, H, D)
24     end
```

pushes a new node on the per-thread stack, the algorithm initializes the attributes of the new node (lines 1-6 in Function PUSHNODE). Each node maintains the following five attributes:

1. Work ($N.w$). The node's work. Upon node creation, it is set to zero.

2. Task creation cost ($N.t$). The node's task creation cost. We attribute the estimated cost of creating a task to a task P-node. For all other nodes, this value is set to zero.

3. Left Sibling Serial Work ($N.LSW$). The critical path of all S-node and W-node child nodes that have completed execution. It is initialized as the empty set.

4. Critical Path ($N.CP$). The node's critical path. It is initialized as the empty set.

5. Parent's Serial Work ($N.PSW$). A snapshot of the parent node's serial work at the time of the child node's creation. Effectively, this is the union of the critical path of all of the child node's left sibling S-node and W-node.

Whenever a node completes execution, it updates some of its attributes before summarizing the information under its subtree by atomically updating some of its parent node's attributes. The key idea used by the on-the-fly profiling algorithm for computing a node's critical path in a parallel execution is to make a copy of the parent's current serial work (*i.e.*, the chain of serial W-node descendants of the parent node's completed child S-node and W-nodes) and store it in whenever a child node is created (line 6 in Algorithm 13). The child node later uses this copy to update the parent node's critical path whenever the child node completes execution (line 14 in Algorithm 13). Intuitively, this mimics the offline algorithm to update a node's critical path (line 9 in Algorithm 12). By making this copy whenever a child node is created, OMP-ADVISER can continue executing the OpenMP application under test without waiting for the computation under the newly created child node's subtree to complete. Whenever the child node completes execution, it has access to sufficient information to update its parent node's critical path if the computation in the subtree rooted at the child node is part of the parent's critical path.

Whenever a node is popped, OMP-ADVISER first checks if the current node is a W-node. If so, it updates the node's work directly by calling the function READCOUNTER. Next, the algorithm sets the W-node's critical path to a single element set comprised of itself. Lastly, since the W-node has completed execution, we update its parent node's left sibling serial work attribute (line 11 in Algorithm 13) to include the current W-node as its newly completed child W-node. Since the update at line 11 can be done by multiple child nodes in parallel, we need to perform this update atomically to avoid data races. Similarly, suppose the node being popped is an S-node. In that case, we update its parent node's left sibling serial work attribute by adding the current S-node's critical path to its parent completed left sibling serial work attribute (line 13-14 in Algorithm 13).

For P-nodes, the on-the-fly profiling algorithm aggregates the tasking overhead per static tasking directive. The function TASKLOCK is used to identify the P-nodes corresponding task

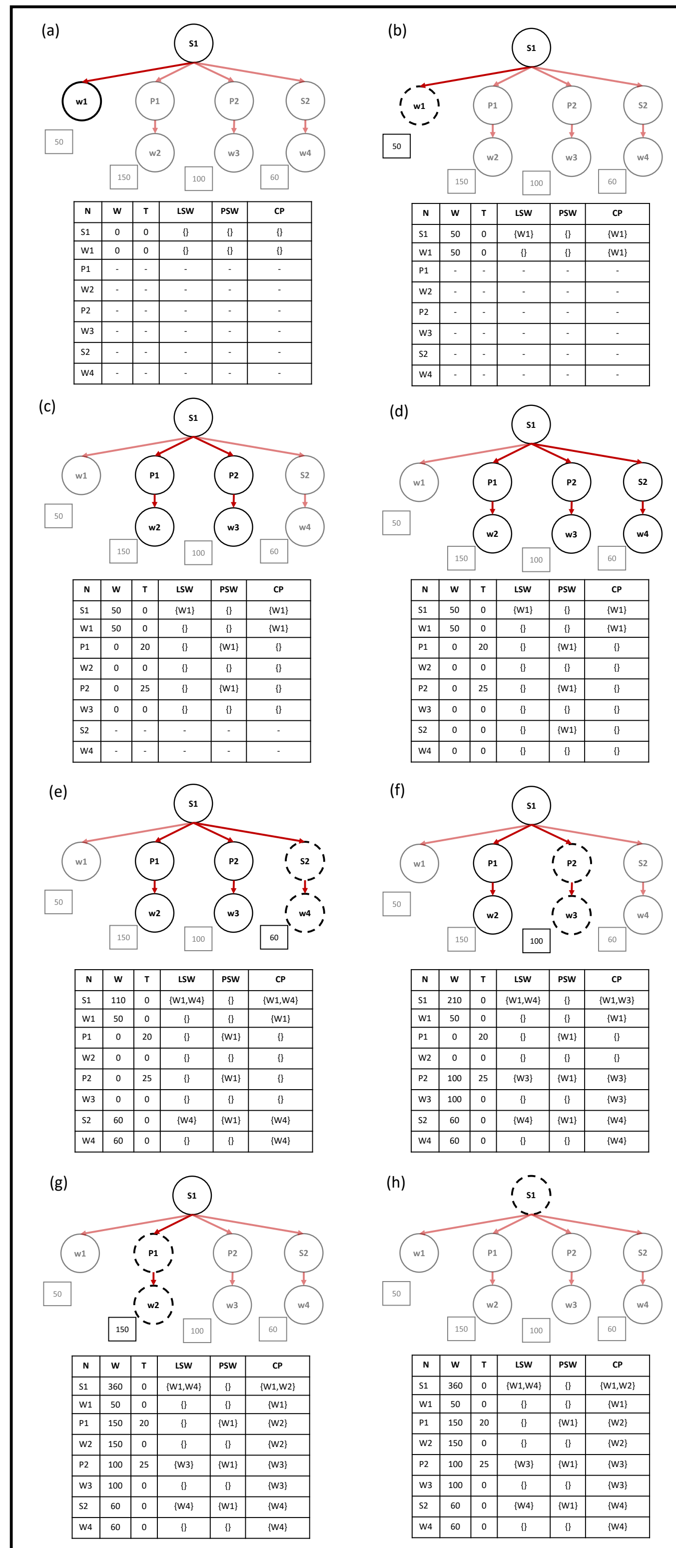
directive. Next, the task directive location is used to index a map that aggregates the tasking overhead per static location (lines 15-18 in Algorithm 13).

Next, OMP-ADVISER updates the parent node's work by adding the current child node's work atomically (line 19 in Algorithm 13). Afterward, we conditionally update the parent node's critical path by comparing the parent node's current critical path with the child node's critical path candidate. The critical path candidate is computed by taking the union of the child node's critical path with the parent node's serial work, which is copied to the child node at the time of its creation (line 20 in Algorithm 13).

Lastly, OMP-ADVISER updates each static directive's aggregate work and serial work if the current node corresponds to any OpenMP directives. The function AGGREGATENODEPERSTATICDIRECTIVE ensures that we do not double count the aggregation of these quantities. Before adding the current node's work and serial work to the aggregation map, the function checks the path from the current node towards the root node. Suppose the path contains any node with the same corresponding location. In that case, the algorithm does not include the current node's work and serial work to ensure that each program fragment is attributed to a static directive exactly once.

On-the-fly Profiling Example. Figure 3.8 shows the execution of OMP-ADVISER's on-the-fly profiling algorithm (shown in Algorithm 13) for a program's OSPG. The first image (shown in Figure 3.8(a)) shows the initial state of the OSPG, which is comprised of a root S-node and a W-node child node. Next, the following images illustrate the incremental changes in the program's OSPG. When a node completes execution and is about to be popped from the OSPG (shown with a dashed circle in Figure 3.8), it updates the attributes of itself and its parent node (the attribute changes are recorded in the table under each OSPG in Figure 3.8).

The next images highlight that the parallel computation under the root node can continue execution without any interruptions. For example, the computation under the parallel P-nodes complete after their right sibling S-node $S2$. When $S2$ completes execution (shown in Figure 3.8(e)), depending on the program's schedule, P-nodes $P1$ and $P2$ may not have completed execution yet. Thus, the longest sequence of serial W-nodes observed in the dynamic execution so far is $\{W1, W4\}$, which is not the root's critical path of $\{W1, W2\}$. However, as the child P-nodes complete execution, they can update the root node's critical path using the parent serial



(a)

N	W	T	LSW	PSW	CP
S1	0	0	{}	{}	{}
W1	0	0	{}	{}	{}
P1	-	-	-	-	-
W2	-	-	-	-	-
P2	-	-	-	-	-
W3	-	-	-	-	-
S2	-	-	-	-	-
W4	-	-	-	-	-

(b)

N	W	T	LSW	PSW	CP
S1	50	0	{W1}	{}	{W1}
W1	50	0	{}	{}	{W1}
P1	-	-	-	-	-
W2	-	-	-	-	-
P2	-	-	-	-	-
W3	-	-	-	-	-
S2	-	-	-	-	-
W4	-	-	-	-	-

(c)

N	W	T	LSW	PSW	CP
S1	50	0	{W1}	{}	{W1}
W1	50	0	{}	{}	{W1}
P1	0	20	{}	{W1}	{}
W2	0	0	{}	{}	{}
P2	0	25	{}	{W1}	{}
W3	0	0	{}	{}	{}
S2	-	-	-	-	-
W4	-	-	-	-	-

(d)

N	W	T	LSW	PSW	CP
S1	50	0	{W1}	{}	{W1}
W1	50	0	{}	{}	{W1}
P1	0	20	{}	{W1}	{}
W2	0	0	{}	{}	{}
P2	0	25	{}	{W1}	{}
W3	0	0	{}	{}	{}
S2	0	0	{}	{W1}	{}
W4	0	0	{}	{}	{}

(e)

N	W	T	LSW	PSW	CP
S1	110	0	{W1,W4}	{}	{W1,W4}
W1	50	0	{}	{}	{W1}
P1	0	20	{}	{W1}	{}
W2	0	0	{}	{}	{}
P2	0	25	{}	{W1}	{}
W3	0	0	{}	{}	{}
S2	60	0	{W4}	{W1}	{W4}
W4	60	0	{}	{}	{W4}

(f)

N	W	T	LSW	PSW	CP
S1	210	0	{W1,W4}	{}	{W1,W3}
W1	50	0	{}	{}	{W1}
P1	0	20	{}	{W1}	{}
W2	0	0	{}	{}	{}
P2	100	25	{W3}	{W1}	{W3}
W3	100	0	{}	{}	{W3}
S2	60	0	{W4}	{W1}	{W4}
W4	60	0	{}	{}	{W4}

(g)

N	W	T	LSW	PSW	CP
S1	360	0	{W1,W4}	{}	{W1,W2}
W1	50	0	{}	{}	{W1}
P1	150	20	{}	{W1}	{W2}
W2	150	0	{}	{}	{W2}
P2	100	25	{W3}	{W1}	{W3}
W3	100	0	{}	{}	{W3}
S2	60	0	{W4}	{W1}	{W4}
W4	60	0	{}	{}	{W4}

(h)

N	W	T	LSW	PSW	CP
S1	360	0	{W1,W4}	{}	{W1,W2}
W1	50	0	{}	{}	{W1}
P1	150	20	{}	{W1}	{W2}
W2	150	0	{}	{}	{W2}
P2	100	25	{W3}	{W1}	{W3}
W3	100	0	{}	{}	{W3}
S2	60	0	{W4}	{W1}	{W4}
W4	60	0	{}	{}	{W4}

Figure 3.8: Execution of the on-the-fly profiling algorithm (shown in Algorithm 13) to compute the parallelism profile of an example OSPG. Nodes that haven't been created or have completed execution are greyed out. Nodes being popped from the OSPG are marked with a dashed circle.

work (PWS) attribute. When $P1$ completes execution (shown in Figure 3.8(g)), it compares the longest chain of serial work with the candidate critical path in the root node (line 20 in Algorithm 13). It replaces it since the candidate critical path performs more serial work (200 vs. 150).

The last image (Figure 3.8(h)) shows the OSPG when the program is about to complete execution. At this point in the execution, the root node's attributes correctly summarize the computation in the entire OSPG. Further, sufficient information has been recorded by the algorithm to generate the program's parallelism profile[9].

Algorithm Analysis

Offline Parallelism Profiling. During the profiling phase, For each program fragment, the OSPG construction algorithm creates a constant number of OSPG nodes (*i.e.*, one or two new nodes per fragment) and performs $O(1)$ operations per node creation. Hence, the offline algorithm's time overhead is $O(1)$ for each program fragment. The space overhead of our offline algorithm is $O(TN)$, where T is the maximum number of worker threads used during program execution and N is the nesting level of the program. This upper bound is derived by considering the OSPG construction algorithm described in Section 3.3.1: the OSPG construction algorithm maintains a slice of the OSPG per worker thread in memory as a stack of nodes. Each stack size is bounded by the height of the program's OSPG, which is proportional to the program's nesting level, $O(N)$. Hence, the space overhead of the offline profiling algorithm is $O(TN)$. Whenever a node is popped from the per-thread stack, the offline algorithm must serialize the node. While this serialization does not impact offline profiling's asymptotic overhead, serializing nodes to log files has a large overhead. We use standard buffering techniques to amortize the cost of initiating IO requests.

Offline Analysis Algorithm. The offline analysis algorithm loads the entire program's OSPG in memory and performs a bottom-up traversal of the OSPG to generate its parallelism profile. The number of W-nodes in the program's OSPG is proportional to the number of program fragments, $O(F)$. In the worst case, the number of OSPG nodes is $O(FN)$. For

[9] Updates to the per static location map is not shown in the figure due to space constraints.

example, a task-based OpenMP program where each task creates two child tasks results in an OSPG with approximately $FN/2$ nodes. Loading the OSPG memory and traversing all nodes to generate the parallelism profile indicates that the offline analysis algorithm's time and space overhead is $O(FN)$.

Online Parallelism Profiling. Since we described an optimized version of the on-the-fly profiling mode for programs that do not have an ST-node in their OSPG, the algorithm's asymptotic cost differs depending on the existence of ST-nodes in the program's OSPG. In the absence of ST-nodes, since the critical path of an OSPG can be inferred by only inspecting its child nodes, the online analysis algorithm can safely deallocate OSPG nodes as soon as they are popped from the worker thread's stack. Thus, indicating that the asymptotic time and space overhead of on-the-fly profiling mode in the absence of ST-nodes is similar to offline parallelism profiling, $O(1)$ per OSPG node and $O(TN)$ space overhead. In contrast, in the presence of ST-nodes, the on-the-fly profiling algorithm must keep all the OSPG nodes in the subtree rooted at the S-node corresponding to the current parallel region in memory. This requirement results in increased asymptotic memory overhead for this algorithm. For example, consider a program where all the computation is solely performed within a single parallel region. For such example programs, similar to the offline analysis profiling algorithm's space overhead, the space overhead is $O(FN)$. Despite performing more operations per OSPG node, the asymptotic running time of the algorithm remains at $O(1)$ per OSPG node in the presence of ST-nodes.

In practice, most OpenMP programs use more than one instance of a parallel region during execution, thus providing opportunities to reclaim OSPG nodes after each parallel region to save memory. Our observations with testing OpenMP benchmarks indicate that the amount of space needed by the on-the-fly profiling algorithm is small and is a fraction of the working set size of the OpenMP application under test (See Section 3.9 for empirical results).

3.7 What-if Analysis

Once a profiling tool identifies program bottlenecks, the developer must decide which bottlenecks to optimize. A standard solution to optimize program bottlenecks is to parallelize them. However, parallelizing a program bottleneck requires domain knowledge and expertise. It is

often considered a time-consuming process to devise concrete parallelization strategies for a program's serialization bottlenecks. Further, a program may have many candidate regions for optimization. The developer may spend time optimizing a region that does not reduce the program's critical path. Ideally, it is desirable to avoid such situations. In this section, we describe OMP-ADVISER's what-if analysis that aims to assist developers in identifying regions that matter for performance in the OpenMP application.

As described in Section 3.6, a program's parallelism profile highlights the program's scalability bottlenecks regions. Program regions with low parallelism and performing significant amount of work on the critical path are potential candidates for optimization. However, by just looking at the program's parallelism profile, it may not be clear which regions must be optimized to resolve the program's scalability bottlenecks. For example, optimizing a program region on the critical path might shift the critical path to other regions that perform similar amounts of work, thus not improving the program's parallelism.

OMP-ADVISER's what-if analysis uses its performance model to estimate the impact on a program's parallelism and its critical path when user-defined program regions are hypothetically parallelized. The key idea that enables what-if analysis is that the OSPG, along with fine-grained measurements of computation, can be used to quantify the change in parallelism in a program. After using what-if analysis, OMP-ADVISER reports a what-if parallelism profile. The what-if parallelism profile is similar to the program's parallelism profile. Except, it estimates the parallelism profile of the program when user-defined regions are parallelized. The what-if parallelism profile enables the developer to infer which regions must be prioritized for parallelization to address the program's serialization bottlenecks.

OMP-ADVISER supports what-if analysis in two modes. In the first mode, named user-defined what-if analysis, the developer chooses the program regions for what-if analysis. After a what-if profiling execution, OMP-ADVISER reports the program's what-if parallelism profile. By comparing the what-if profile with the program's parallelism profile, the developer can assess the impact of parallelizing different program regions on the program's parallelism, and focus on concretely parallelizing them.

In the second mode, named automatic what-if analysis, instead of requiring the developer to choose program regions for what-if analysis, they specify their desired target parallelism. With

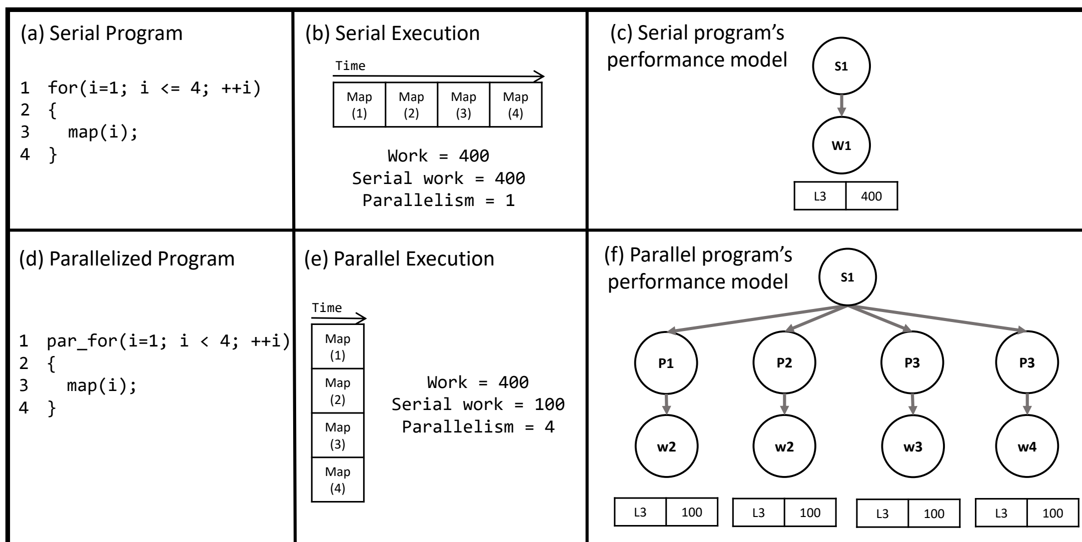


Figure 3.9: A data-parallel for loop executed serially and in parallel. (a) Serial program with data-parallel loop (b) The serial program's trace. (c) Performance model of the serial program. (d) Parallelelized version of the program (e) The parallel program's trace. (f) Performance model of the parallelized program.

automatic what-if analysis, OMP-ADVISER iteratively performs what-if analysis on different program regions until it finds a list of program regions that must be concretely parallelized to achieve the target parallelism, or it determines that it is infeasible to reach the target parallelism for the current input. This mode can improve developer productivity by automating the process of choosing program regions for what-if analysis, and providing a list of regions that must be parallelized to achieve the target parallelism. Next, we describe how OMP-ADVISER uses our performance model to mimic the effects of parallelizing program regions.

Using OMP-ADVISER's Performance Model to Estimate the Impact of Parallelizing Program Regions. In an ideal system, parallelizing a program region by a given factor is equivalent to reducing its serial work by the same factor. For example, Figure 3.9 shows the effects of parallelizing a data parallel serial loop on its performance model. Effectively, parallelizing the program by a factor of four, quadruples the amount W-nodes in the program. The parallelized program performs the same amount of total work. However, each fragment will perform a quarter of its original work.

OMP-ADVISER's performance model captures the logical series-parallel relations between different program fragments as W-nodes and attributes their work to each W-node. Inspired

by the example in Figure 3.9, OMP-ADVISER's approach to estimating the effects of parallelizing a program region by a given factor, is to identify the set W-nodes corresponding to the program region, reduce their serial work while keeping their total work the same, and to recompute the program's parallelism profile. OMP-ADVISER's parallelism profile reports an upper bound on the program's speedup. We maintain this property in our what-if parallelism profiles by assuming that a program region for what-if analysis corresponds to an embarrassingly parallel [128] workload, where program fragments can parallelized into smaller tasks without incurring additional communication. Hence, OMP-ADVISER assumes that a program region can be parallelized into a bag-of-tasks, where the new task executes independently (Shown in Figure 3.9(f)).

As described in Section 3.5, to better characterize performance bottlenecks in OpenMP applications, its necessary to augment the performance model with OpenMP runtime costs. Similarly, when performing what-if analysis, it is pertinent to take the increased runtime costs associated with creating finer-grained parallelism into account. When computing the program's parallelism profile, OMP-ADVISER computes the average task creation cost per tasking directive. When hypothetically parallelizing a program region for performing what-if analysis, since OMP-ADVISER assumes that the region is parallelized into a bag-of-tasks, it attributes the average task creation to each hypothetical new task and reports the what-if estimate for the average tasking overhead per static location. Further, OMP-ADVISER puts a realistic limit of the granularity of what-if parallelism profiling by not reducing the serial work of a W-node below a certain empirical threshold that is a function of the average costs of task creation on the system.

3.7.1 User-defined What-if Analysis

In this mode, the developer selects the program regions used for what-if analysis. To select a program region, the OMP-ADVISER library provides a pair of annotation calls `What-if-Begin` and `What-if-End`, for the programmer to annotate a static program region with their specified parallelism factor. In the profiling execution, OMP-ADVISER keeps track of program fragments corresponding to the static region. When performing what-if analysis, OMP-ADVISER computes the what-if parallelism profile of the application by reducing the serial work of the fragments of

Algorithm 14: OSPG construction when a user-defined what-if annotation is encountered. The annotation includes its location in the program source.

```
1  procedure WHAT_IF_BEGIN (tid, start_loc)
2      W_i ← nodes[tid].pop()
3      W_r ← CreateWnode()
4      W_r.start_loc ← start_loc
5      W_r.parent ← nodes[tid].top()
6      nodes[tid].push(W_r)
7      return
8  procedure WHAT_IF_END (tid, end_loc)
9      W_r ← nodes[tid].pop()
10     W_r.end_loc ← end_loc
11     W_j ← CreateWnode()
12     W_j.parent ← nodes[tid].top()
13     nodes[tid].push(W_j)
14     return
```

each what-if region by the user-specified parallelism factor while keeping their total work the same.

To distinguish a program fragment within a what-if region, during OSPG construction, OMP-ADVISER creates a new W-node whenever the dynamic trace enters or exits a what-if region. Creating additional W-nodes does not impact the series-parallel relation in the program. Based on the definition of a fragment and W-nodes, two sibling W-nodes in place of a W-node with the same amount of total are equivalent. However, this distinction between W-nodes ensures that each static what-if region corresponds to a set of non-overlapping W-nodes, simplifying tracking of what-if regions during profiling execution.

Offline What-if Profiling. To enable offline what-if profiling, the primary change during the performance model's construction is to track the W-nodes corresponding the a what-if region. OMP-ADVISER tracks a W-node's corresponding what-if region by maintaining two additional attributes for each W-node. The `start_loc` and `end_loc` are used to identify which what-if region a W-node belongs to, if any. They are set whenever the program's execution encounters a what-if region. Algorithm 14 shows OMP-ADVISER's operations whenever a what-if region is entered or exited during program execution. Later, when the performance model is serialized for offline parallelism profile analysis, OMP-ADVISER records the what-if location of all W-nodes and the parallelism factor of each what-if region.

Algorithm 15: Compute input node N's work ($N.w$) and and the set of W-nodes on its critical path ($N.S$) in what-if profiling mode.

1 **function** ProcessNodeWhatIf (N)
2   **if** NodeType(N) $=$ `W-node` **then**
3     $N.S \leftarrow \{N\}$
4     $R \leftarrow$ GetWhatIfRegion(N)
5   **if** $R \neq \varnothing$ **then**
6       $N.sw \leftarrow N.w/R.f$
7   **end**
8   **else**
9       $N.sw \leftarrow N.w$
10   **end**
11   **return** $\langle N.w, N.sw, N.S \rangle$
12   **end**
13   $N.w \leftarrow 0$
14   $N.sw \leftarrow 0$
15   $N.S \leftarrow \{\}$
16   $LS \leftarrow \{\}$
17   $LP \leftarrow \{\}$
18   **foreach** $C \in$ ChildNodes(N) **do**
19     $N.w \leftarrow N.w + C.w$
20     $cu \leftarrow \sum_{S \in N.S} S.sw$
21     $cd \leftarrow \sum_{S \in LS} S.sw + \sum_{S \in C.S} S.sw$
22   **switch** NodeType(C) **do**
23   **case** `P-node` **do**
24   **if** HasDependency(C) $= True$ **then**
25           $DP \leftarrow$ LongestDependenceChain(C)
26           $DW \leftarrow$ ImmediateFragments(C)
27           $cd \leftarrow cd + \sum_{W \in DW} W.w$
28   **if** $cd > cu$ **then**
29             $N.S \leftarrow N.S \setminus LP$
30             $N.S \leftarrow N.S \cup C.S \cup DW$
31             $LP \leftarrow C.S \cup DW$
32   **else if** $cd > cu$ **then**
33           $N.S \leftarrow N.S \setminus LP$
34           $N.S \leftarrow N.S \cup C.S$
35           $LP \leftarrow C.S$
36   **case** `S-node` $\vee$ `W-node` **do**
37   **if** $cd > cu$ **then**
38           $N.S \leftarrow N.S \setminus LP$
39           $N.S \leftarrow N.S \cup C.S$
40         $LS \leftarrow LS \cup C.S$
41   **case** `ST-node` **do**
42         $\langle ST_p, ST_s \rangle \leftarrow$ PartitionNodes(C.S)
43   **if** $cd > cu$ **then**
44           $N.S \leftarrow N.S \setminus LP$
45           $N.S \leftarrow N.S \cup C.S$
46           $LP \leftarrow ST_p$
47         $LS \leftarrow LS \cup ST_s$
48   **end**
49   **end**
50   **return** $\langle N.w, N.sw, N.S \rangle$

Offline What-if Analysis. The offline algorithm to generate a program's what-if parallelism profile is similar to the Algorithms 9 and 10 described in Section 3.6.1 with some modifications to reduce the serial work of W-nodes corresponding to annotated what-if regions. The main changes occur in Algorithm 10. Algorithm 15 illustrates the node processing algorithm with what-if analysis. In this algorithm, each node maintains an additional serial work attribute ($N.sw$). When processing a W-node, the algorithm uses the GETWHATIFREGION function to determine the what-if region it corresponds to. If the W-node node is part of any what-if region, the algorithm reduces the node's serial work by the region's parallelism factor (lines 5-7 in Algorithm 15). If not, we set the W-node's serial work to be the same as its work, resulting in the same behavior as Algorithm 10 for W-nodes that do not belong to any what-if regions. In addition, to incorporate the what-if analysis W-nodes' reduced serial work unto the internal nodes of the OSPG, computing the current critical path and the critical path candidate (lines 20-21 in Algorithm 15) is based on each node's serial work attribute instead of its work attribute.

On-the-fly What-if Profiling. Similarly, we can support On-the-fly what-if profiling by slightly modifying Algorithm 13. Similar to offline what-if analysis, we need to add a serial work attribute for each node. Further, when a W-node corresponding to a what-if region finishes executing and is popped from the per-thread stack (line 9-12 in Algorithm 13, we need to reduce its serial work by the user-specified parallelism factor (similar to lines 5-7 in Algorithm 15).

3.7.2 Automatic What-if Analysis

In this mode, the developer provides a target parallelism that they wish to achieve when they devise concrete strategies to parallelize the program. With automatic what-if analysis, OMP-ADVISER iteratively chooses different regions for what-if analysis and assesses the changes in parallelism. After completion, OMP-ADVISER returns a list of program regions, sorted by their impact on the program's parallelism, that must be parallelized to reach the user's target parallelism, or it reports that it could not identify such a list.

Recall that to perform what-if analysis, we need to know the corresponding static source location for W-nodes. In user-defined what-if analysis, we track this correspondence by using manual annotations. For automatic analysis, it is unrealistic to expect the user to annotate all program regions manually. Instead, OMP-ADVISER tracks the start and end location for each

W-node in the performance mode, which effectively makes it possible to apply what-if analysis to all static locations in the program.

A naive solution to our problem is for OMP-ADVISER to perform what-if analysis on all program regions, check if the target parallelism is met, and return the list of all program regions. However, returning such a list provides no new insights for the developer. Ideally, we want the returned list to be in descending order of impact on the program's parallelism. Further, we want the list to contain the minimal set of regions required to reach the target parallelism. Intuitively, when designing concrete optimization strategies, other things being equal, a developer would want to prioritize optimizing a region that has more impact on the program's parallelism and achieve the target program parallelism by optimizing the smallest number of regions.

In a program with N candidate regions for what-if analysis, there exists $N!$ permutations to choose the ordering of program regions for what-if analysis. One way to sort the regions based on their impact on the program's parallelism is to iterate over all possible permutations, apply what-if analysis based on the ordering in the permutation, and report the permutations that reach the target parallelism with the least number of regions. However, due to the fast growth of factorial function, this solution does not scale for programs with more than twenty regions.

To scale OMP-ADVISER's automatic what-if analysis algorithm for larger applications, we use a heuristic to greedily determine the ordering of program regions. Our heuristic prioritizes program regions that perform the largest amount of work on the program's critical path. The rationale with this heuristic is that in an ideal system (*i.e.*, a system with no runtime overhead), parallelizing the region that performs the most amount of serial work on the critical path is guaranteed to reduce the program's critical path the most. Hence, resulting in the most improvement in the program's parallelism. To reach a target parallelism, iteratively choosing the next program region that performs the most work on the critical path results in a list with a minimal number of regions to parallelize on an ideal system. The chosen regions are sorted in descending order of impact on the program's parallelism.

Algorithm 16 illustrates the steps involved to automatically generate a list of program regions to reach a target parallelism. First, the algorithm takes the program's performance model M, and the target parallelism we wish to achieve t as input and initializes the list of program regions to an empty list. Next, the algorithm generates the program's parallelism profile, updating the

Algorithm 16: Automatically generate a list of program regions that, if parallelized, the program's parallelism becomes greater or equal to the target parallelism. Conversely, the algorithm may terminate early without being able to produce such a list.

```
1  function AutomaticWhatIfProfile (M, t)
2      L ← InitList()
3      ⟨M, C⟩ ← ParallelismProfile(M)
4      do
5          ⟨M, C, p⟩ ← WhatIfProfile(M, C, L)
6          if C = ∅ then
7              return ⟨fail, p, L⟩
8          L ← AppendList(L, C)
9          if p ≥ t then
10             return ⟨success, p, L⟩
11     while True
```

program's performance model, and returning the program region[10] C that currently performs the most amount of serial work on the program's critical path (line 3 in Algorithm 16). Next, the algorithm's main loop is used to iteratively perform what-if analysis on the program region currently performing the most amount of work on the critical path.

At each iteration, the algorithm generates the program's what-if profile, updating the program's performance model, choosing a new program region that performs the most amount of serial work on the critical path, and the program's parallelism after performing what-if analysis. We pass the list of program regions to the what-if profile generator to ensure that the function returns a new program region C that has not been added to L in previous iterations, avoiding a potential infinite loop. Subsequently, if the current round of what-if analysis identified such a new region, the algorithm adds it to the result list L. If not, the algorithm reports that it has failed to find a list of regions to reach the target parallelism. At the end of an iteration, the algorithm checks if the program's parallelism is greater or equal to the target parallelism. If so, the algorithm successfully completes, returning the list of regions to optimize to reach the target parallelism and the possible parallelism that may be achieved after optimizing them. Otherwise, the program continues to the next iteration to identify a new region to add to the list.

[10] A program region is comprised of a start and end location, as described in Section 3.7.1 and is used to specify a what-if analysis region.

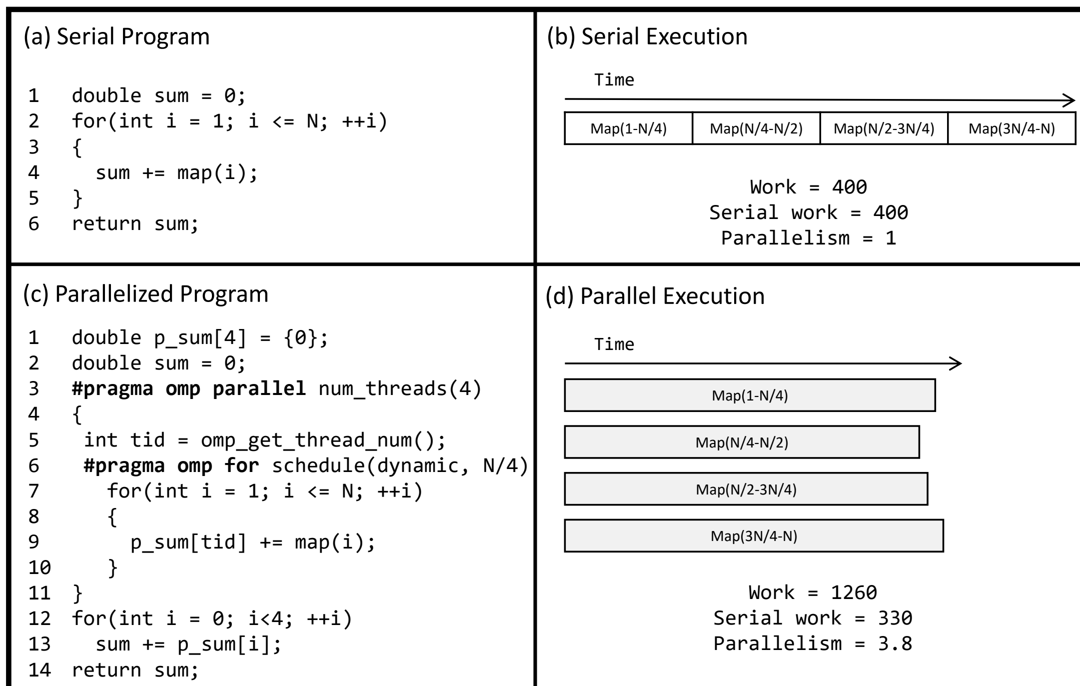


Figure 3.10: (a) A data-parallel for loop performing a simple map-reduce computation. (b) the serial for loop execution trace. (c) An OpenMP conversion of the serial application. The program has work inflation caused by false sharing on variable `p_sum`. The execution trace of the parallel for loop (lines 6-10 in Figure 3.10). Because of false sharing, the parallel version performs more work compared to the serial program.

3.8 Differential Analysis with OMP-ADVISER

The OpenMP API enables developers to add incremental parallelism to their applications. Different worksharing and tasking constructs are used to divide serial fragments of execution between different worker threads to reduce the program's critical path. Commonly, OpenMP constructs are used to break down a problem into smaller subproblems and then solve them in parallel. For data-parallel computation, it might seem reasonable to the developer that by breaking down the problem into N subproblems, the critical path to reduce by a factor of N, thus achieving scalable speedup.

However, when executing a parallelized program on actual hardware, the developer may not observe their expected speedups. One primary source of bottlenecks in parallel applications is caused by unintended resource contention caused by executing the application on hardware with parallel execution units. This unintended resource contention may be a result of the underlying

microarchitecture design choices on modern parallel hardware. Unintended resource contention on parallel hardware manifests as additional work performed by each thread. In such cases, the parallel execution performs more total work than the serial execution for performing the same computation on the same input.

Increased work performed by the parallel execution of an application is a well-known cause for a lack of scalability in parallel applications. It is commonly referred to as parallel work inflation [13, 150]. For example, consider the data-parallel for loop shown in the code snippets in Figure 3.10(a) and (c). The code snippet represents a data-parallel computation where the elements of the `map` function are summed up and returned as the result of the computation. Figure 3.10(c) shows an attempt at parallelizing the code with OpenMP's parallel for construct to execute on a 4-core system. Since the calls to the `map` function are independent, and the computation is embarrassingly parallel, the developer expects to see scalable speedup close to $4\times$ over serial execution. However, when executed on a modern multi-core processor, the program has a speedup of $1.2\times$ over serial execution. The lack of scalable speedup is caused by unintended resource contention on multiprocessors cache lines known as false sharing [40].

In modern multiprocessors, each processor has a local cache. The cache coherency protocol ensures a consistent view of memory. Whenever a processor writes to a cache line, the cache coherency protocol invalidates the cache line of other processors. When the other processors access variables from the invalidated cache line, they incur a performance hit caused by a cache miss of the invalidated cache line. In the example program in Figure 3.10(c), the developer has avoided true sharing by using per-thread partial sums variables `p_sum[i]`, however the four elements of the `p_sum` array are mapped to the same cache line. Thus, the writes to elements of `p_sum` result in false sharing.

The false sharing in the example program in Figure 3.10(c) results in each processor spending more cycles to perform the same computation from the serial execution from the program in Figure 3.10(a), leading to parallel work inflation. When looking at the two programs' execution traces in Figure 3.10(b) and (d), we observe that compared to the serial execution, the parallelized version achieves an almost $4\times$ improvement in parallelism. However, since parallel work inflation has resulted in increased total work and serial work in the parallelized program, it does not achieve scalable speedup. Thus, we recognize that looking at the program's parallelism

alone is not sufficient and may even be misleading in identifying the source of performance bottlenecks in programs that suffer from parallel work inflation.

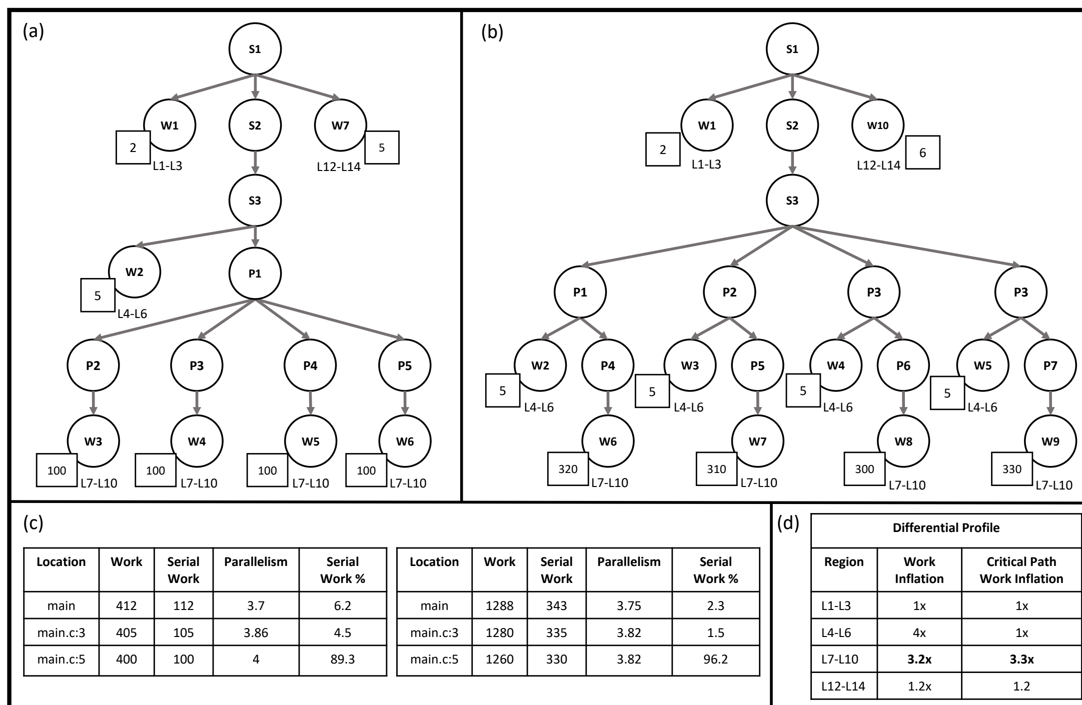
The importance of identifying and lowering parallel work inflation in OpenMP programs has been explored in prior tools [150]. However, these tools primarily focus on identifying work inflation at the program level. In this section, we utilize our performance model to extend OMP-ADVISER with a differential analysis technique aimed at identifying and pinpointing program regions that exhibit work inflation.

Differential profiling [130, 173] is an effective technique to identify performance issues in applications. Typically, performance analysis tools produce a performance report for a single run of the application. Suppose the user wishes to assess the impact of changing a parameter on an application's performance characteristics. In such cases, they must run the performance analysis tool twice and compare the generated performance reports' manually. Manually comparing reports generated by performance analysis tools could be time-consuming and not helpful in pinpointing the bottleneck's source. The basic idea behind differential profiling is to automate this process. With differential profiling, performance measurements are performed multiple times under different conditions (*i.e.*, varying program or system parameters). Afterward, The measurements are compared to gain additional insight into performance issues in the application, reporting a differential profile to characterize the impact of the changed parameter on the application's performance profile. OMP-ADVISER's differential analysis uses its performance model to pinpoint program regions that are likely suffering from unintended resource contention.

3.8.1 Differential Analysis Approach

To use differential profiling to identify parallel work inflation, we need to compare the performance model of the parallel execution with the performance model of an execution that does not suffer from parallel work inflation. Second, we need to ensure that the two performance models remain comparable. If we satisfy these two conditions, we can apply differential analysis to pinpoint which program regions suffer from parallel work inflation by comparing the two performance models.

For a given input, assuming the program is deterministic, the logical series-parallel relations between program fragments do not change for different program interleavings or when varying



Location	Work	Serial Work	Parallelism	Serial Work %
main	412	112	3.7	6.2
main.c:3	405	105	3.86	4.5
main.c:5	400	100	4	89.3

Location	Work	Serial Work	Parallelism	Serial Work %
main	1288	343	3.75	2.3
main.c:3	1280	335	3.82	1.5
main.c:5	1260	330	3.82	96.2

Differential Profile		
Region	Work Inflation	Critical Path Work Inflation
L1-L3	1x	1x
L4-L6	4x	1x
L7-L10	**3.2x**	**3.3x**
L12-L14	1.2x	1.2

Figure 3.11: (a) Performance model of the single thread execution of the program in Figure 3.10(c). (b) Performance model of the parallel execution of the program in Figure 3.10(c) with four worker threads.

the number of worker threads. Thus, program fragments and their series-parallel relations are encoded similarly by the program's OSPG when executed serially by a single worker thread or in parallel by multiple threads. This property is the key enabler for our differential profiling approach to identify parallel work inflation. Namely, we choose the single-threaded OpenMP execution of the program to estimate an oracle execution with no work inflation. Our differential profiling technique compares the performance model of the single-threaded OpenMP execution of the application with its parallel execution. Since the two performance models are comparable (*i.e.*, they identify the same set of fragments and critical path), we can compare them to identify program regions that suffer from work inflation. Specifically, our differential profiling technique enables fine-grained identification of regions with parallel work inflation. Thus, the addition of differential profiling to parallelism profiling and what-if analysis enables OMP-ADVISER to identify the primary sources of bottlenecks in parallel applications[11].

[11] Lack of parallelism, runtime overheads, and work inflation.

For example, consider the performance models of the OpenMP program in 3.10(c) when executed with one worker thread and when executed in parallel with four worker threads. The two performance models are shown in Figure 3.11(a) and (b). Note that the two performance models contain W-nodes that represent the same set of program regions[12]. Further, for a pair of W-nodes in the single thread execution of the application, there always exists a corresponding pair of W-nodes in the parallel execution that encodes their logical series-parallel the same as the serial execution. For instance, the pair of W-nodes $(W3, W7)$ that execute serially in the single thread execution have a corresponding pair of $(W6, W10)$ in the parallel execution. Hence, there is a mapping from each W-node in the single thread execution to one or more W-nodes in the parallel execution. We use this mapping between W-nodes of the two executions to identify the W-nodes experiencing work inflation in the parallel execution.

The parallelism profile of the single thread execution and the parallel execution with four threads is shown in Figure 3.11(c). Despite work inflation in the parallel execution, the two profiles appear similar since the work inflation has occurred uniformly across all worker threads. Our differential analysis aims to summarize the difference between the two performance models and highlight work inflation to the user. The result of applying differential analysis on the program from Figure 3.10(c) is presented in the differential profile in Figure 3.11(d). For each program region, the differential profile reports: (1) the amount of work inflation, which is the ratio of the region's work in parallel execution over its single thread execution. (2) Critical path work inflation: the ratio of a region's critical path work in the parallel execution over its single thread execution. From the example program's differential profile, we can identify that the program region corresponding to the worksharing loop at lines 7-10 in Figure 3.10(c) suffers from work inflation. Despite performing the same number of iterations in both executions, there is a large increase in the work performed in the parallel execution compared to the single thread execution. Overall, program regions that perform a large amount of work on the critical path and exhibit larger than 1 work inflation are good candidates for optimizing program performance.

It is worth noting that the parallel execution may have additional W-nodes and internal nodes compared to the single thread execution. The increased W-nodes in the parallel execution's

---

[12] We assume that the program is deterministic for a given input. Hence both executions take the same branches in their dynamic trace.

performance model is caused by the SPMD (single program, multiple data) execution of the worker threads within a parallel region that are not enclosed inside a worksharing or a tasking construct. For example, the code region that uses `omp_num_threads` function to get the current worker thread's number falls under this category (lines 4-6 in Figure 3.10(c)). In such scenarios, a W-node in the serial execution maps to N W-nodes in the parallel execution with N worker threads. In the performance model of the program in Figure 3.10(c)), W-node $W2$ in the single thread execution corresponds to W-nodes $(W2, W3, W4, W5)$ in the parallel execution with four worker threads (as shown in Figure 3.11). For the example program, this scenario causes the differential profile to report a $4\times$ work inflation for the program region at lines 4-6 in the program from Figure 3.10(c). When performing differential analysis, we keep the program input the same across both executions to observe strong scaling in the application (*i.e.*, keeping total work the same across executions while increasing hardware parallelism). However, for SPMD fragments, the number of worker threads in the program is part of the program input and changes the amount of work performed by each worker thread depending on the number of worker threads. For such SPMD regions, the work inflation metric is not a good indicator of scalability bottlenecks. Instead, we need to examine the critical path work inflation to assess if the region is suffering from work inflation. If the critical path work inflation is close to 1, it indicates that the SPMD region does not have work inflation, as shown for our example program in Figure 3.11(d).

Still, an SPMD fragment that performs work proportional to the number of worker threads can produce misleading differential profiling results for its regions. For example, the SPMD fragment can use the OpenMP library function to `omp_get_num_threads` to perform more or less work worker-thread compared to its single thread execution. Hence, reading the differential profiling results for SPMD regions must be done with caution. In practice, with the introduction of worksharing constructs, this type of ad hoc work distribution among worker threads is no longer common in OpenMP applications. However, since such programs are still valid OpenMP programs, the takeaway point of treating the differential profile values corresponding to SPMD program regions remains the same.

3.9 Evaluation

In this section, we discuss the evaluation of OMP-ADVISER. We start by providing a brief description of the OMP-ADVISER prototype and our experimental setup. Next, we report the results of our experimental evaluation. We designed these experiments to answer the following research questions: (1) Can OMP-ADVISER effectively identify performance bottlenecks? (2) Does addressing the performance bottlenecks reported by OMP-ADVISER result in improved performance? (3) What are the analysis overheads of using OMP-ADVISER with different OpenMP applications? (4) How does OMP-ADVISER compare to other state-of-the-art OpenMP performance analysis tools?

As a performance analysis tool, OMP-ADVISER can report performance bottlenecks. However, it does not attempt to address these bottlenecks automatically. For our experimental evaluation, we were able to use OMP-ADVISER as a guide to finding performance bottlenecks. In some cases, we were able to address the identified bottlenecks and observed improved speedups. However, addressing all performance bottlenecks reported by OMP-ADVISER is challenging and requires domain-specific knowledge.

Prototype. The OMP-ADVISER prototype can analyze C/C++ OpenMP applications. Our prototype is comprised of a library that is linked with the application under test, supporting both offline and on-the-fly profiling modes and a standalone analyzer to perform offline-analysis. As described in Section 3.3.1, OMP-ADVISER utilizes OMPT callbacks to intercept OpenMP runtime events and construct the program's OSPG and perform fine-grained measurements. Our prototype works with LLVM-10 and its OpenMP runtime implementation. To perform fine-grained measurements, OMP-ADVISER uses `perf_events` module on Linux to measure work using different metrics such as execution cycles or dynamic instruction. Our prototype is publicly available [34].

Experimental Setup. All experiments in this section are performed on an Ubuntu 16.04 machine with a 16-core Xeon 6130 processor running at 2.1GHz and with 32GB of memory. We disabled dynamic frequency scaling and hyper-threading to ensure that our results remain consistent across different runs and among tested tools. Further, all tests were performed with the LLVM-10 release branch with the included OpenMP runtime.

Table 3.1: List of OpenMP benchmarks used for evaluating OMP-ADVISER

Benchmark	Application	Description	Input
PBBS	BFS	Run a breadth first search on a directed graph	40M
PBBS	CompSort	Sort a sequence given a comparison function	100M
PBBS	Convex Hull (CH)	Compute the convex hull of a set of points in 2D plane	100M
PBBS	Delaunay Triangulation (DT)	Compute Delaunay triangulation of a set of points in 2D plane	10M
PBBS	Delaunay Refinement (DR)	Refine a Delaunanay triangulation given in input angle	5M
PBBS	Dictionary	Perform dictionary operations	100M
PBBS	Integer Sort (ISORT)	Sort fixed-length unsigned integer keys	100M
PBBS	Maximal Independent Set (MIS)	Compute maximal independent set of the input graph	10M
PBBS	Maximal Matching (MM)	Compute maximal matching of the input graph	40M
PBBS	Nbody Forces (NBODY)	Solve n-body force calculation problem for a set of points in 3D space	1M
PBBS	K-Nearest Neighbors (KNN)	Find k nearest neighbors of a set of points	10M
PBBS	Ray Cast	Find the intersection of input rays and triangles	default
PBBS	Remove Duplicates	Remove duplicate items from a sequence of input objects	100M
PBBS	Spanning Forest (SF)	Compute a valid spanning forest for input undirected graph	10M
PBBS	Minimum Spanning Forest (MSF)	Compute minimum spanning forest of for input weighted undirected graph	10M
PBBS	Suffix Arrays (SA)	Generate suffix array of input string	100M
BOTS	Alignment	Align a sequences of proteins	prot.100
BOTS	FFT	Compute a Fast Fourier Transformation	2^{25}
BOTS	Floorplan	Compute the optimal placement of cells in a floorplan	input.20
BOTS	Fibonacci	Compute Nth Fibonacci number	47
BOTS	Knapsack	Solve the Knapsack problem	input.040
BOTS	Health	Simulates a country health system	large
BOTS	Nqueens	Finds solutions of the N-Queens problem	15
BOTS	Sort	Use a mixture of sorting algorithms to sort a vector	2^{25}
BOTS	SparseLU	Compute the LU factorization of a sparse matrix	100×100
BOTS	Strassen	Compute matrix multiplication with Strassen's method	$2^{12} \times 2^{12}$
Coral	AMGmk	Algebraic multigrid solver microkernel	50k
Sequoia	IRSmk	27-point stencil loop kernel	input_50
Coral	Lulesh	Performs a hydrodynamics stencil calculation	default
SPEC	KDTree	Build a k-d tree using random coordinate points	ref
Coral2	Quicksilver	Monte Carlo transport benchmark	Coral2_P_1

Benchmarks. We evaluate OMP-ADVISER with a set of different OpenMP application chosen from different benchmarks suites from Coral [1], Sequoia [107], PBBS [178], BOTS [63], Kastros [2], and SPEC [179]. The benchmarks are listed in Table 3.1, including a description and the input used to analyze the application. Some applications, such as Lulesh and Quicksilver, use MPI + OpenMP for parallelism. To test them with OMP-ADVISER, we built and ran them with MPI disabled. By default, the PBBS applications are not OpenMP applications. We ported them to OpenMP. Our ported PBBS applications exist in two variants, one using static loop scheduling and the other using dynamic loop scheduling. All other OpenMP benchmarks were left unchanged. To run an OpenMP application with a single thread for OMP-ADVISER's differential analysis, we set the number of worker threads to one using the `OMP_NUM_THREADS` environment variable.

3.9.1 Effectiveness of OMP-ADVISER in Identifying Scalability Bottlenecks

We used OMP-ADVISER with the set of benchmarks and report the initial program speedup[13], parallelism, and tasking overhead percentage[14] in Table 3.2. Since our test setup uses a 16-core processor, we consider speedup values close to $16\times$ as good program speedups. To achieve scalable speedup, the program should have more parallelism than the target core count. Further, a higher parallelism value also helps with reducing load imbalance among worker threads. However, having a very high parallelism value can negatively impact the program speedup if the aggregate cost of task creation becomes comparable to the amount of work performed by a task or results in parallel work inflation. For benchmarks that do not use OpenMP tasking constructs, the amount of tasking overhead is reported as 0% in Table 3.2.

Benchmarks with Serialization Bottlenecks

After computing the parallelism profile of the applications listed in Table 3.2 and observing their speedup, the benchmarks described next exhibit low logical parallelism and low speedup. Thus, indicating that these applications could have serialization bottlenecks or load imbalance. In the use cases that follow, we improved the program's parallelism by addressing some of the bottlenecks reported by OMP-ADVISER and improved the application's speedup on our test system. Unless stated otherwise, the parallelism profiles shown in this section use execution cycles as their work measurement unit. Further, the parallelism profiles shown in the upcoming figures do not show all regions in the program's parallelism profile. Instead, we only show the relevant program regions to keep the figures concise.

AMGmk. The AMGmk benchmark from the LLNL Coral benchmark suite [1] 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 an axpy vector operation. Initially, the speedup over serial execution on a 16-core machine was $7.16\times$. To understand bottlenecks, we used OMP-ADVISER to generate its parallelism profile, which is shown in Figure 3.12(a). The entire program has a parallelism of 10.46 and regions

[13] Speedup over serial execution is measured as the ratio of the serial program execution over parallel execution.

[14] Defined as the percentage of aggregate task creation costs over the program's aggregate work.

Table 3.2: List of OpenMP benchmarks used for evaluating OMP-ADVISER

Application	Parallelism	Tasking Overhead%	Initial Speedup
BFS	47.68	0%	5.56×
CompSort	13.01	0%	3.77×
Convex Hull (CH)	2.31	0%	2.10×
Delaunay Triangulation (DT)	1.32	0%	1.23×
Delaunay Refinement (DR)	19.63	0%	4.00×
Dictionary	719.86	0%	7.55×
Integer Sort (ISORT)	40.21	0%	3.36×
Maximal Independent Set (MIS)	42.9	0%	6.13×
Maximal Matching (MM)	250.4	0%	7.19×
Nbody Forces (NBODY)	1.13	0%	1.07×
K-Nearest Neighbors (KNN)	72.05	0%	4.11×
Ray Cast	5.74	0%	4.16×
Remove Duplicates	150.2	0%	4.76×
Spanning Forest (SF)	8.92	0%	4.19×
Minimum Spanning Forest (MSF)	2.93	0%	1.94×
Suffix Arrays (SA)	2.68	0%	1.93×
Alignment	501.75	8.53e−3%	15.87×
FFT	114.99	9.01%	9.10×
Floorplan	130.6	0.11%	15.08×
Fibonacci	122.98	6.53e−4%	14.75×
Knapsack	9913.71	43.82%	118.92×
Health	168.56	8.67e−2%	14.13×
Nqueens	851.63	5.82e−4%	15.78×
Sort	8929.28	2.33%	13.32×
SparseLU	119.93	3.30e−2%	14.73×
Strassen	31.52	5.93e−4%	8.69×
AMGmk	10.46	0%	7.16×
IRSmk	13.56	0%	3.06×
Lulesh	12.86	0%	4.56×
KDTree	550.78	20.01%	1.14×
Quicksilver	13.05	0%	11.07×

Location	Parallelism	Serial Work %
main	10.46	35.46
relax.c:91	15.89	36.36
csr_matvec.c:176	15.79	20.16
vector.c:383	14.81	3.32

(a) Initial parallelism profile

Location	Parallelism	Serial Work %
main	12.83	21.44
relax.c:91	15.93	45.54
csr_matvec.c:176	15.86	24.74
vector.c:383	14.88	4.32

(b) What-if profile

Location	Parallelism	Serial Work %
main	13.01	17.70
relax.c:91	15.89	44.11
csr_matvec.c:179	15.71	20.16
vector.c:383	14.72	4.05

(c) Final parallelism profile

Figure 3.12: Parallelism profile for AMGmk. (a) parallelism profile on the original program. (b) what-if profile after manually annotating selected regions. (c) The parallelism profile of the program after concretely parallelizing the regions selected using what-if analysis.

Location	Parallelism	Serial Work %
main	13.05	7.53
mc_omp.hh:4	14.03	92.47

(a) Initial parallelism profile

Location	Parallelism	Serial Work %
main	87.02	58.07
mc_omp.hh:4	206.18	41.93

(b) What-if profile

Location	Parallelism	Serial Work %
main	89.35	43.50
mc_omp.hh:4	190.99	46.50

(c) Final parallelism profile

Figure 3.13: Parallelism profile for Quicksilver. (a) parallelism profile on the original program. (b) what-if profile after manually annotating selected regions. (c) The parallelism profile of the program after concretely parallelizing the regions selected using what-if analysis.

that are not associated with any OpenMP constructs constitute 35.46% of the program's critical path (*i.e.*, lines corresponding to main in the tables shown in Figure 3.12(a)). Upon examining the source code, we observed that parallel regions were interspersed with serial code fragments. We used user-defined what-if analysis to annotate the first of these four regions to estimate the increase in parallelism when this region is optimized by 16×. The what-if profile showed a slight increase in parallelism from 10.46 to 10.95. To further increase the program's parallelism, we annotated one more serial region and generated the program's what-if profile when both regions are hypothetically parallelized on 16-cores. Figure 3.12(b) shows the what-if parallelism profile where the parallelism of the program increases to 12.83. Subsequently, we examined these regions closely to parallelize them. The first region in the sparse matrix multiply kernel contains a for loop that initializes a vector and performs the algebraic operation on each vector element. Similarly, the second region contained the vector operation kernel with for loops. We parallelized both regions with an OpenMP worksharing for loop construct. The parallelism profile after concrete parallelization of the program is shown in Figure 3.12(c), which almost attains the same parallelism as the estimate from what-if analyses. Moreover, the program's speedup increased from 7.16× to 8.93× on a 16-core machine.

QuickSilver. Quicksilver [3] is a Coral application that solves a simplified dynamic Monte Carlo particle transport problem. We configured it to run on a single node machine by disabling MPI and used the Coral2_P_1 input to profile the program. The program has a parallelism of 13.05 (Figure 3.13(a)). The program is comprised of a single kernel that is parallelized using an OpenMP worksharing 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 3.13(b) presents the what-if profile that shows the increase

Location	Parallelism	Serial Work %
main	1.32	98.60
delaunay.c:258	25	1.15
sequence.h:275	1.99	0.17

(a) Initial parallelism profile

Location	Parallelism	Serial Work %
main	15.08	85.26
delaunay.c:258	24.67	12.23
sequence.h:275	2.22	1.53

(b) What-if profile

Location	Parallelism	Serial Work %
main	33.1	33.04
delaunay.c:243	71.71	34.45
delaunay.c:258	23.42	27.30
sequence.h:275	2.21	3.18

(c) Final parallelism profile

Figure 3.14: Parallelism profile for Delaunay Triangulation. (a) parallelism profile on the original program. (b) what-if profile after manually annotating selected regions. (c) The parallelism profile of the program after concretely parallelizing the regions selected using what-if analysis.

in parallelism for both the worksharing loop (to 206.18) and the program (to 87.02). We noticed that the loop was using static scheduling, which resulted in some load imbalance between the worker threads. We changed the loop scheduling clause to use dynamic scheduling and increased the chunk size by a factor of 16. Figure 3.13(c) shows the final parallelism profile after this optimization. The program's speedup increased from 11.07× to 12.45× on a 16-core machine.

Delaunay Triangulation. This PBBS application [178] computes the Delaunay triangulation of a given a set of points in the 2D plane. The program has an initial speedup of 1.23×. Figure 3.14(a) shows that the low speedup can be explained by the program's low parallelism of 1.32 and that most of the program's work (*i.e.*, 98.6%) is being performed serially. For its parallelism profile, we see that the OpenMP constructs at `delaunay.c:258` adds some parallelism to the program. Upon examining the source code, we confirmed that this OpenMP construct is located in function `incrementallyAddPoints`, the main kernel of the Delaunay triangulation computation. However, it only parallelizes a small region of code within this function (lines 258-260). Using OMP-ADVISER's user-defined what-if analysis, we annotated the serial portion of this function and generated its what-if profile (*i.e.*, a 16× hypothetical optimization), as shown in Figure 3.14(b). The program's what-if profile indicates that the program's parallelism would significantly increase if we can parallelize this serial region. We identified a bounded for loop within the what-if region that finds the nearest points for each point to compute their Delaunay Triangulation. Since the loop appears to be data-parallel, we

Location	Parallelism	Serial Work %
main	2.92	97.82
seq.h:403	5.87	1.45
seq.h:429	20.61	0.48
spec.h:72	14.81	0.12

(a) Initial parallelism profile

Location	Parallelism	Serial Work %
main	88.56	14.64
sampleSort.h:126	51.80	45.54
sampleSort.h:100	59.81	24.74
seq.h:443	20.57	13.45

(b) What-if profile

Location	Parallelism	Serial Work %
main	94.53	9.39
sampleSort.h:126	52.01	33.63
sampleSort.h:100	59.63	30.57
seq.h:443	20.84	13.68

(c) Final parallelism profile

Figure 3.15: Parallelism profile for Minimum Spanning Forest. (a) parallelism profile on the original program. (b) what-if profile after manually annotating selected regions. (c) The parallelism profile of the program after concretely parallelizing the regions selected using what-if analysis.

parallelized the loop using an OpenMP worksharing loop. The optimized program's speedup improves from 1.23× to 9.53×[15].

Minimum Spanning Forest. This program computes the minimum spanning forest of the input undirected weighted graph using the parallel Kruskal algorithm. Initially, it has a speedup of 1.94× over serial execution. Its parallelism profile shows that the program has low parallelism (*i.e.*, 2.92 in Figure 3.15(a)). OMP-ADVISER's what-if analysis helped us find two regions which, when hypothetically optimized by 16×, can increase the program's 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 1.94× to 7.54×. When we ran the modified program with OMP-ADVISER's what-if analysis, it showed that optimizing regions in the functions called by parallel sort can improve parallelism to 88.56 (Shown in Figure 3.15(b)). We parallelized the recursive block transpose function within this region using OpenMP tasks. The final program's increased to 95.83 as shown in Figure 3.15(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. With our changes to the program, we increased the speedup from 1.94× to 8.91× over serial execution.

NBody. The NBody benchmark computes the gravitational force vector of each point, given a set of points in 3D space. The OpenMP port of this PBBS application primarily uses

[15]It appears that this serial loop was a performance bug in the PBBS benchmarks since in later versions of the benchmark, this loop is parallelized.

Location	Parallelism	Serial Work %
main	1.13	98.87
seq.h:123	1.30	0.52
seq.h:403	1.82	0.19
CK.c:226	9.31	0.16

(a) Initial parallelism profile

Location	Parallelism	Serial Work %
main	15.06	84.64
seq.h:123	1.33	6.93
seq.h:403	1.88	2.50
CK.c:226	9.06	2.36

(b) What-if profile

Location	Parallelism	Serial Work %
main	77.53	17.95
seq.h:123	1.31	35.75
seq.h:403	1.84	12.83
CK.c:226	9.13	11.58

(c) Final parallelism profile

Figure 3.16: Parallelism profile for NBody. (a) parallelism profile on the original program. (b) what-if profile after manually annotating selected regions. (c) The parallelism profile of the program after concretely parallelizing the regions selected using what-if analysis.

Location	Parallelism	Serial Work %
main	2.31	86.30
seq.h:150	5.54	8.87
seq.h:277	13.02	4.38
seq.h:106	51.88	0.23

(a) Initial parallelism profile

Location	Parallelism	Serial Work %
main	22.07	57.76
seq.h:397	246.25	1.34
seq.h:392	40.83	0.17
seq.h:380	21.82	10.45

(b) What-if profile

Location	Parallelism	Serial Work %
main	219.61	0.14
seq.h:200	16.06	33.32
seq.h:152	59.77	16.28
seq.h:378	21.75	15.23

(c) Final parallelism profile

Figure 3.17: Parallelism profile for Convex Hull. (a) parallelism profile on the original program. (b) what-if profile after manually annotating selected regions. (c) The parallelism profile of the program after concretely parallelizing the regions selected using what-if analysis.

OpenMP worksharing constructs, which does not parallelize the program sufficiently. Hence, its initial speedup was 1.08× and its parallelism was 1.13, as shown in the parallelism profile in Figure 3.16(a). OMP-ADVISER's what-if analysis suggested that hypothetically parallelizing certain code regions in the serial parts of the application by 16× can increase the parallelism to 15.06 (Figure 3.16(b)). Upon examining these regions, we identified a recursive serial function that was invoked on a disjoint set of vertices. We parallelized this function by converting it to spawn tasks for each recursive call. This change in the program increased its parallelism to 77.53 (Figure 3.16(c)). Resulting in an increased speedup from 1.08× to 13.28×.

Convex Hull. This PBBS application computes the convex hull of points in a 2D space. Initially, it was parallelized with OpenMP worksharing for loop constructs. It had a low parallelism of 2.31 and a speedup 2.10× over serial execution. We used OMP-ADVISER's what-if analysis to find two regions that if optimized, could improve the program's parallelism as shown in the what-if parallelism profile shown in Figure 3.17(b). parallelism. We were able to increase parallelism in one region located at the `filter` function in the source file `seq.h`

by using an OpenMP for loop. Further, we increased the parallelism in the main kernel of the convex hull function by using recursive decomposition with tasks. These changes improved the program's parallelism from 2.51 to 220, increasing the program's speedup 2.10× to 11.13×.

Strassen. This benchmark performs a matrix multiplication using the Strassen algorithm. It uses OpenMP tasking to solve the recursive submatrix multiplications. The OpenMP program has a speedup of 8.69× over the serial execution. When we generated its parallelism profile, we noticed that the program has a parallelism of 31.52 while its aggregate tasking overheads are low. From this observation, we posited that increasing the program's parallelism might reduce any load imbalance in the program. Since the program's aggregate tasking overhead is negligible, the task granularities should remain large enough even if we increase the program's parallelism by increasing the recursive algorithm's nesting depth to create more tasks. We changed the nesting cut-off value of the program by one. This change improves the program parallelism to 69.43 and improves the program's speedup from 8.69× to 9.64× over serial execution.

We also used an updated variant of this program that uses task-dependency from the Kastros [2] benchmark suite. This variant omits some taskwait constructs in the program and replaces them with finer granularity task-dependencies. Because of this change, this variant's parallelism profile reports higher parallelism values compared to the original program. We also see higher speedups over serial execution (*i.e.*, 13.39× compared to 8.69×). We used the same idea of improving the BOTS program's parallelism for the Kastros benchmark. The modified task dependency program achieves parallelism of 78.54, and its speedup increases from 13.39× to 14.78× over serial execution on our 16 test machine.

Benchmarks with High Tasking Overheads

Some of our tested applications use OpenMP use tasking constructs. As shown in Table 3.2, the parallelism profile of these applications has a non zero tasking overhead percentage. These benchmarks predominantly use OpenMP tasking to solve recursive divide-and-conquer algorithms. As shown in the table, the benchmarks with a large amount of parallelism and tasking overheads of less than one percent achieve a speedup of over 14× over serial execution, which is close to the linear scaling expected on our 16-core test setup. In contrast, OMP-ADVISER reports high tasking overheads in some applications. The following use cases describe the

Location	Parallelism	Tasking overhead%	Serial Work %
main	550.79	19.43	89.89
kdtree.cc:545	14.88	8.58	1.23
kdtree.cc:567	14.86	10.98	0.73
kdtree.cc:636	3871.78	0.01	0.02
kdtree.cc:649	3834.68	0.01	0.01

(a) Initial parallelism profile

Location	Parallelism	Tasking overhead%	Serial Work %
main	77.41	0.98	85.75
kdtree.cc:418	24.14	9.23	1.96
kdtree.cc:448	24.79	8.38	0.95
kdtree.cc:636	165.60	0.03	3.51
kdtree.cc:649	159.44	0.03	4.08

(b) Final parallelism profile

Figure 3.18: Parallelism profile with tasking overhead measurements for KDTree. (a) parallelism profile of the original program. (b) The parallelism profile of the program after reducing its tasking overheads.

applications where we could lower these overheads and observed improved running times in the modified application.

KDTree. This benchmark is from the SPEC OMP 2012 [179] benchmark suite. The program takes a set of random points in 3D or 4D space. It constructs a k-d tree and finds the points that are close to each point in the tree. The benchmark suite provides three sets of inputs for this application ranging from smaller to larger inputs[16]. The program uses recursive traversals to search for a point and then recursively find the points adjacent to it, which is parallelized using OpenMP tasks. When we ran the program, we observed little to no speedups over the serial execution on our 16-core test setup across all three program inputs.

We generated the program's parallelism profile to diagnose the issue. The parallelism profile is shown in Figure 3.18(a) indicates that the program has a large tasking overhead of 19.48%. We also see that the tasking constructs located at the source file `kdtree.cc` at lines 636 and 649 (located in function `sweepkdtree`) have a large parallelism of 3800. After looking at the source code of the `sweepkdtree` function, we learned that the high amount of parallelism at this OpenMP construct is caused by how the `sweepkdtree` is written. `sweepkdtree` is a recursive function that uses OpenMP tasking, and in its base case, it calls to another recursive function, `searchkdtree`, that is also parallelized using OpenMP tasks. The source locations of the tasking constructs of the `searchkdtree` are at lines 545 and 567 in the source file `kdtree.cc`. In the parallelism profile in Figure 3.18(a), we see that these two locations suffer

[16]From small to large; the inputs are named test, train, and ref.

Location	Parallelism	Tasking overhead%	Serial Work %
main	9913.71	43.82	9.51
knapsack.c:166	5032.76	22.59	1.08
knapsack.c:170	4740.77	21.23	80.83
knapsack.c:305	11443.20	0	1.08

(a) Initial parallelism profile

Location	Parallelism	Tasking overhead%	Serial Work %
main	35.00	19.01	0.34
knapsack.c:166	24.14	10.01	0.03
knapsack.c:170	24.79	9.20	99.31
knapsack.c:305	165.60	0	0.03

(b) Final parallelism profile

Figure 3.19: Parallelism profile with tasking overhead measurements for BOTS knapsack. (a) parallelism profile on the original. (b) The parallelism profile of the program after reducing its tasking overheads.

from high tasking overheads, indicating that the task granularity at these locations might be too small to amortize the cost of creating and executing them. We changed the program by removing the tasking constructs at the function `searchkdtree` (*i.e.*, tasking constructs at lines 545 and 567 in file `kdtree.cc`). As expected, this change reduces the program's parallelism profile, as shown in the modified program's parallelism profile shown in Figure 3.18(b), from 550.79 to 77.41. However, this amount of parallelism is still sufficient to utilize all cores in our 16-core test system. Further, the aggregate tasking overhead of the program is significantly reduced. After this change, the program's speedup improves for all three inputs of the KDTree benchmark. For example, for the largest input, ref, the speedup increases from 1.14$\times$ to 6.14$\times$.

Knapsack. This benchmark solves the knapsack problem using tasks. Unlike the other benchmarks we tested, this application's parallel version performs less computation for the same input than its serial variant. This is because the optimization problem solved by this benchmark prunes out child tasks as soon as an intermediate result becomes worse than the best solution found by the program so far. Hence we observe a superlinear speedup of 118.92$\times$ over the serial program. However, when we generated the program's parallelism profile, as shown in Figure 3.19(a), we saw that the program has a high aggregate tasking overhead. Prompted by this observation, we changed the program's tasking cut-off value from 24 to 16. With this change, the program's task nesting level is lowered, which results in an increase in task granularity and lower tasking overheads.

The parallelism profile of the modified program with lowered aggregate tasking overheads is shown in Figure 3.19(b). The modified program finishes the knapsack computation faster and

Table 3.3: List of OpenMP applications with low initial parallelism used with OMP-ADVISER's automatic what-if analysis.

Application	Parallelism	No of Regions	Automatic What-if Parallelism
Convex Hull (CH)	2.31	5	38.58
AMGmk	10.46	3	34.51
IRSmk	13.56	1	132.92

increases the program's speedup from 118.92× to 1496.13× compared to its serial execution. By comparison, the modified parallel version of the program is 12.58× faster than the original parallel program. Since the parallel version of the program can perform less work based on how it prunes the task subtrees, we also checked the amount of total work performed by the modified program to explain the increased speedup over the original parallel program. Our measurements indicate that the program with a lower tasking cut-off performs 10.20× less aggregate work[17] than the original parallel program. Hence, even when considering the lower amount of work performed by the modified program, we still observe some speedup that may be attributed to the lowered aggregate tasking overhead in the modified knapsack program.

3.9.2 Effectiveness of OMP-ADVISER's Automatic What-if Analysis

To assess the effectiveness of OMP-ADVISER's automatic what-if analysis technique, we tested the applications in our benchmark suite that exhibited low logical parallelism and set OMP-ADVISER to report program regions that, if parallelized by a factor of 16×, would improve the program's parallelism to 32. The parallelism factor and target parallelism are user-defined OMP-ADVISER parameters. We chose a parallelism factor of 16× to match the hardware core count of our experimental setup. We chose the target parallelism of 32, a slightly higher value than our core count, to account for potential load imbalances in the optimized program. Further, OMP-ADVISER's automatic what-if analysis never chooses the same region twice. This setting achieves two goals. First, OMP-ADVISER's automatic analysis is guaranteed to terminate. Second, based on the automatic what-if analysis algorithm, the regions reported by automatic what-if analysis are ordered based on the amount of work performed on the critical path. Hence, if a region performs a significant amount of serial work on the critical path, it is likely that it

[17] 4.8e8 cycles compared to 4.9e9 cycles in the original program.

will be among the first regions reported by automatic what-if analysis, and there is no reason to report it again later during the analysis.

Understanding the inner working of some tested OpenMP benchmarks can be challenging and require domain-specific knowledge. Hence, we do not have ground truth knowledge for all tested benchmarks to validate the results of OMP-ADVISER's automatic what-if analysis. Further, it may not be possible to parallelize a program region reported by OMP-ADVISER's what-if analyses, or parallelizing it after the region has sufficient parallelism on the current test setup may not result in noticeable speedups. One way to empirically check if the results of OMP-ADVISER's automatic what-if analysis are credible is to check for overlap between the regions reported by OMP-ADVISER and the regions we used with user-defined what-if analysis in Section 3.9.1 to increase some benchmark's speedup. Hence, we include the PBBS applications we sped up using what-if analyses from Section 3.9.1 to compare the results reported by automatic what-if analysis.

Table 3.3 shows the list of tested benchmarks and the number of regions reported by OMP-ADVISER's automatic what-if analysis, illustrating the parallelism of the application, the number of regions identified by the automatic what-if analyzer, and the final parallelism reached by the automatic what-if analyzer. In some applications, such as Delaunay Triangulation, the final parallelism reached by automatic what-if analysis is below the target parallelism 32. For these applications, after an initial improvement in parallelism by optimizing a few regions, the program's hypothetical parallelism reaches the target parallelism by reporting tens of program regions where each region slightly improves the program's parallelism as each region contributes a small fraction to the program's critical path. For this evaluation, attempting to parallelize tens of program regions concretely, each providing a small improvement in parallelism, was not possible. Hence, we report the lower automatic what-if parallelism, where we were able to manually examine the regions reported by OMP-ADVISER.

AMGmk. Using OMP-ADVISER's user-defined what-if annotations, we were able to identify two regions on the program's critical path, that when parallelized, would improve the program's parallelism from 10.46 to 13.01, resulting in an improvement in speedup from 7.16× to 8.93× over serial execution as described in Section 3.9.1. However, the program's final parallelism is still lower than the number of cores in our test system. Thus, using the

unmodified program, we applied OMP-ADVISER's automatic what-if analysis to see what other program regions can be optimized to improve the program's parallelism. OMP-ADVISER's what-if analysis reports three program regions that, if parallelized, can improve the program's parallelism to 34.51. The first region reported is the serial region starting from the main function and ending before the matrix-vector kernel, which OMP-ADVISER estimates would improve the program's parallelism to 12.85. This serial region includes covers the same regions we identified when using user-defined what-if analyses.

The next two regions reported by OMP-ADVISER are within the relax kernel of AMGmk. This result is consistent with the parallelism profile of the optimized program shown in Figure 3.12(c) as it indicates that the OpenMP parallel region corresponding to `relax.c:91` performs the most amount of work on the critical path. However, these two regions are already parallelized using OpenMP worksharing loops. Increasing the logical parallelism of this region by reducing the loop's chunk size does not result in notable speedups in the program on our 16-core test setup.

Convex Hull. To improve this benchmark's parallelism to over the target parallelism of 32, OMP-ADVISER's automatic what-if analysis identifies five different regions. The unmodified application has a parallelism of 2.31. Among the regions reported, two regions `seq.h:150-275` and `seq.h:277-278` closely match the serialization bottlenecks we identified using user-defined what-if analysis and later optimized to increase the program's speedup from 2.10× to 11.13×. These two regions include two macros, `maxIndex` and `filter`. These two macros are located within the serialization bottleneck region we identified using user-defined what-if annotations in the main kernel of the Convex Hull. The other three regions reported by OMP-ADVISER are located in `geometryIO.h` and `IO.h` files. Both these regions include functions that are used for performing I/O and reading the input file for the Convex Hull program in parallel using worksharing for loops. While hypothetically increasing the parallelism of these regions improves the program's parallelism and reduces the program's critical path, in practice, increasing the parallelism for these regions does not result in a noticeable effect on the program's performance since the performance in these regions is I/O bound. Overall, our results

with Convex Hull indicate that OMP-ADVISER's automatic what-if analysis effectively identifies serialization bottlenecks in this benchmark, alleviating the need for user-defined what-if annotations.

To check the validity of the regions identified by automatic what-if analysis, we also developed a suite of small microbenchmarks. Each microbenchmark is comprised of an OpenMP program with less than 100 lines of code. We identified the program's logical parallelism in each microbenchmark and manually verified the results of OMP-ADVISER's automatic what-if analysis. In all microbenchmarks, the results reported by automatic what-if analysis matches our expected results.

3.9.3 Effectiveness of OMP-ADVISER's Differential Analysis

When testing our benchmark application with OMP-ADVISER and using parallelism profiling and what-if analysis, we still observe that some applications do not achieve scalable speedup despite having sufficient parallelism. In this section, we describe using OMP-ADVISER's differential analysis with one of these applications.

Lulesh. This benchmark is part of LLNL's proxy applications, and it is used to approximate the hydrodynamics equations discretely by partitioning the spatial problem domain into a collection of volumetric elements defined by a mesh [102]. The application is parallelized to the OpenMP API by using worksharing loop constructs with static scheduling. As reported in Figure 3.20(a), Lulesh has a parallelism of 12.86, which is higher than its speedup of 4.56×. Such a discrepancy for a data-parallel application indicates that the program may be experiencing parallel work inflation. Hence, we used OMP-ADVISER's differential analysis to identify which regions of the application are experiencing parallel work inflation, if any.

OMP-ADVISER's differential analysis, shown in Figure 3.20(b), identifies that the program has parallel work inflation. Specifically, Lulesh's parallelism profile indicates that the worksharing for loops in source file `lulesh.cc` at lines 832, 2364, 2123, 2121 perform 36.31% of the work in the program's critical path, and they suffer from parallel work inflation when measuring program cycles and local HITM[18]. A large number of HITM events in these program regions in

[18] HITM events count the cache hits in a modified cache line. A high number of HITM events in the parallel execution indicates true or false sharing among worker threads

Location	Parallelism	Serial Work %
Main	12.86	12.48
lulesh.cc:832	15.87	8.79
lulesh.cc:2364	19.66	7.57
lulesh.cc:2131	12.43	7.46
lulesh.cc:2121	12.43	7.35

(a) Parallelism profile

Region	Inflation Cycles	Inflation Local HITM
Main	1.76x	607.88x
lulesh.cc:832-1016	1.04x	28.54x
lulesh.cc:2364-2366	18.89x	67950x
lulesh.cc:2133-2148	1.51x	2344.26x
kdtree.cc:2123-2129	1.79x	1307.11x

(b) Differential profile

Figure 3.20: Parallelism profile and differential analysis for Lulesh. (a) parallelism profile of the original program (b) Differential analysis result.

Lulesh may indicate that these regions have large cache line contention, limiting the scalability of the parallel execution of Lulesh. Inspecting these parallel loops' body, we attempted to reduce cache line contention by using OpenMP runtime and guided loop scheduling instead of static. However, this simple change does not result in a noticeable reduction in work inflation, and the added cost of choosing runtime loop scheduling ends up slightly reducing the program's speedup. Reducing or rearranging the work performed in these loops can reduce work inflation. However, with our limited domain knowledge, we could not change the computation within these loops without impacting the application's correctness.

To check that OMP-ADVISER's differential analysis can pinpoint regions suffering from parallel work inflation in some metric of interest, we developed a suite of small microbenchmarks. Each microbenchmark is less than 100 lines of code and has a program region experiencing unintended resource contention that manifests as parallel work inflation in a performance metric. In all microbenchmarks, OMP-ADVISER's differential analysis can identify these program regions.

3.9.4 OMP-ADVISER Performance Overhead

In addition to effectively identifying performance bottlenecks, a performance analysis tool should be practical to use. That is, the tool should have low run-time and memory overheads. Further, it should be able to analyze long-running applications. Table 3.4 shows the run-time and memory overheads of OMP-ADVISER's on-the-fly profiler compared to the original OpenMP

Table 3.4: OMP-ADVISER' s performance overheads. The first two columns show the on-the-fly profiler's run-time overhead compared to the unmodified serial and OpenMP application. The third column compares the program's resident memory usage in profiling mode over parallel execution.

Application	Run-time overhead (serial)	Run-time overhead (parallel)	Memory overhead
Alignment	0.06×	1.01×	1.51×
Floorplan	0.09×	1.35×	2.46×
Fibonacci	0.07×	1.09×	1.18×
Health	1.10×	15.51×	1.35×
Sort	1.47×	10.12×	1.83×
NQueens	1.39×	18.48×	1.20×
Sparselu	0.08×	1.13×	1.27×
Strassen	0.13×	1.14×	1.01×
BFS	0.89×	4.95×	1.13×
CompSort	0.70×	2.63×	1.18×
Delaunay Refinement (DR)	0.56×	2.26×	1.03×
Dictionary	1.08×	8.17×	1.11×
Integer Sort (ISORT)	1.20×	4.03×	1.29×
Maximal Independent Set (MIS)	1.13×	6.91×	1.13×
Maximal Matching (MM)	1.06×	7.65×	1.14×
Geo Mean	0.44×	3.66×	1.28×

application. Since OMP-ADVISER's on-the-profiling mode runs with the application under test, it leverages the application's parallelism and OpenMP worker threads to construct the performance model in parallel. Thus, as seen in the first column of Table 3.4, OMP-ADVISER can analyze some applications faster than the original serial application. On average, OMP-ADVISER's run-time overhead compared to serial applications is 0.44×, resulting in a 2.27× faster execution times.

Compared to the 16-core OpenMP execution, OMP-ADVISER on-the-fly profiling mode is, on average, 3.66× slower. OMP-ADVISER's slowdown compared to the parallel execution can be attributed to the additional cost of OSPG construction, accessing hardware performance counters for measurements, and generating the program's parallelism profile. Among these sources, accessing hardware performance counters to perform fine-grained measurements has a large impact on the profiler's slowdown. To test this, we ran a modified version of OMP-ADVISER that, instead of accessing performance counters, attributes the value of one to each program fragment. For example, in the BFS application, this change reduces OMP-ADVISER's slowdown from 4.95× to 1.78× compared to the parallel execution, indicating that investigating solutions to reduce the overhead of accessing hardware counters or reducing the number of calls to them can reduce OMP-ADVISER's run-time overheads.

Further, similar to how sampling-based performance analysis tools can reduce their run-time overhead by reducing the sampling frequency, in worksharing applications, using static scheduling instead of dynamic can significantly reduce the number of loop chunks. This change reduces the program's OSPG size and the number of accesses to hardware counters[19], which in applications that use many worksharing loops can significantly reduce OMP-ADVISER's overheads. For example, for the BFS application, when using our port that uses static for loops instead of dynamic, its run-time overhead decreases from 4.95× to 1.05× compared to the parallel execution.

To empirically assess the memory overheads of OMP-ADVISER, we measured the resident memory used by its on-the-fly profiling mode. Recall that OMP-ADVISER does not maintain the entire program's OSPG in memory and reclaims OSPG node as soon as the program exits a parallel region, to reduce its memory overheads. On average, OMP-ADVISER's on-the-fly profiling mode uses an additional 28% of memory compared to the original OpenMP application. Since OMP-ADVISER maintains a small slice of the program's OSPG in memory at any point in time, it can also be used with long-running OpenMP applications.

To test OMP-ADVISER with long-running applications, we used it to analyze several benchmarks from the Coral suite, including Pennant, Kripke, Lulesh, and Quicksilver. These benchmarks take some standard inputs that let them run for several hours. In all cases, OMP-ADVISER's on-the-fly profiling mode can successfully analyze and generate the program's parallelism profile. In contrast, the offline profiling mode is not practical for long-running applications since the log sizes can grow to up to hundreds of gigabytes for these benchmarks.

3.9.5 Comparison With Other Performance Analysis Tools

Next, we evaluate the effectiveness of OMP-ADVISER by comparing it with state-of-the-art Intel Parallel Studio XE [49] suite of performance analysis tools. This suite provides several tools for analyzing the performance and parallelism of OpenMP applications. For this comparison, we use Intel Advisor [48] and Intel VTune [50]. Intel Advisor provides insight into the program's parallelism within a core (*i.e.*, vectorization) and among cores (*i.e.*, multithreading). It provides

[19] When using this mode, OMP-ADVISER measures the parallelism of the program instead of its logical parallelism.

Table 3.5: List of applications used with Intel Advisor to analyze.

Application	No of Regions	Maximum Program Gain for All Sites	Improved Speedup After Optimization
BFS	3	2.75×	N
Convex Hull (CH)	1	1.02×	N
Delaunay Triangulation (DT)	5	13.70×	Y
Nbody Forces (NBODY)	5	2.73×	Y
Minimum Spanning Forest (MSF)	3	4.18×	N
Knapsack	0	1×	N
Strassen	3	1.09×	N
AMGmk	2	2.15×	N
Lulesh	1	11.17×	N
KDTree	2	3.19×	N
Quicksilver	3	6.95×	Y

a suitability analysis that estimates program speedup after parallelizing different user annotated regions of code. We compare Intel Advisor's suitability analysis with OMP-ADVISER's what-if analysis. Intel VTune is a profiling tool that is used to identify performance bottlenecks. It provides different analysis types, including computation and memory hotspot identification.

Comparison With Intel Advisor. Intel Advisor's primary use is for tuning the vectorization and threading performance of the application. We focus on its threading workflow results for comparison with OMP-ADVISER. The starting point of using Intel Advisor is to run a survey analysis of the application. The threading workflow's survey analysis results include the following information: (1) a list of the program's loops ordered by the time spent by the program and (2) recommendations for improving loops' performance. After the survey analysis, Intel Advisor's threading workflow provides an optional analysis to compute loop trip counts. This information is used by Intel Advisor to generate a roofline model for each loop to indicate how much room for performance improvement exists on the current system under test. Similar to what-if analysis, Intel Advisor provides special program annotations to assess the impact of parallelizing user-selected program regions. After running the program with the added annotations, Intel Advisor generates a suitability report to indicate the potential speedup gains under different thread counts and threading models, including OpenMP and TBB. Lastly, the threading workflow includes a dependency analysis pass to check if the annotated loops carry data dependencies. Intel Advisor's dependency analysis requires another execution of the program and has significantly higher runtime overheads (*i.e.*, 50-100× more than other analysis passes with Intel Advisor).

We test eleven applications from our benchmark suite using Intel Advisor's threading workflow, which includes the applications that we could optimize using OMP-ADVISER. We start testing these applications using survey analysis. Next, we annotate program sites (*i.e.*, regions) and tasks within each site to hypothetically run in parallel based on survey analysis' recommendations. After adding the annotations, we use Intel Advisor's suitability analysis to get the estimated improvement on speedup on a 16 thread system using the OpenMP threading model. Lastly, in some applications where it was not clear if the recommended scalar loops were parallelizable, we use Intel Advisor's dependency analysis to find potential data dependencies. Subsequently, we attempt to parallelize Intel Advisor's recommendation with OpenMP constructs and measure the modified program's speedup. Table 3.5 summarizes our results with Intel Advisor. First, it shows the number of scalar loops reported by Intel Advisor for suitability analysis. The second column reports Intel Advisor's suitability analysis maximum speedup gain estimate after parallelizing all annotated regions. The final column indicates if we observed improved speedups after parallelizing Intel Advisor's recommended program regions.

Intel advisor recommends regions for parallelization in all tested applications except Knapsack. For this benchmark, attempting to annotate program regions results in a crash. For the remaining applications, we attempted to parallelize the reported regions. We could not parallelize many of the reported regions due to data dependencies that result in a data race if the region is parallelized. Intel Advisor's dependency analysis confirms that such data dependencies exist in some of the reported regions. In three applications (Delaunay Triangulation, NBody, and Quicksilver), we observe improved speedup after parallelizing Intel Advisor's recommendations. In these applications, the regions reported by Intel Advisor included the regions we identified using OMP-ADVISER. For the rest of the applications, the performance bottlenecks highlighted by Intel Advisor do not match OMP-ADVISER.

Comparison With Intel VTune. VTune is the flagship performance profiler in Intel's Parallel Studio suite of tools. For parallel applications, the recommended workflow is to use VTune after parallelizing the program with Intel Advisor to identify the application's remaining performance bottlenecks. VTune offers several predefined collections of analysis for the program under test. To test OpenMP benchmarks with VTune, we used the following analysis types: (1) hotspot analysis to find the performance hotspots in the application, (2) HPC

Table 3.6: Summary of Intel VTune's analysis report for different OpenMP applications.

Application	No of Hotspots	CPU Utilization	High Runtime Overhead	Memory Access Issues
BFS	3	3.48	N	Y
Convex Hull (CH)	3	8.35	N	Y
Delaunay Triangulation (DT)	2	5.33	N	Y
Nbody Forces (NBODY)	2	11.99	N	N
Minimum Spanning Forest (MSF)	3	4.47	N	Y
Knapsack	1	8.27	Y	Y
Strassen	2	6.38	Y	Y
AMGmk	3	9.47	N	Y
Lulesh	5	5.18	N	Y
KDTree	3	2.82	Y	N
Quicksilver	5	9.27	Y	Y

performance characterization analysis to perform an OpenMP aware analysis, and (3) memory access analysis to identify memory-related issues with the application. VTune uses hardware sampling techniques to provide a low overhead and low perturbation performance profiling tool. We ran each OpenMP benchmark three times to generate the three analysis reports listed.

After using Intel Advisor to address threading issues in the programs in Table 3.5, we use VTune to identify additional performance issues. Table 3.6 shows the results of profiling these applications using Intel VTune. The first column indicates the number of performance hotspots reported by VTune[20]. The second column indicates VTune's effective CPU utilization measurements. This metric is execution specific and differs from OMP-ADVISER's logical parallelism measurement. The third and fourth columns indicate if VTune flags the application for having high runtime overheads or memory access issues. The hotspots reported by VTune overlaps with the serialization bottlenecks highlighted by OMP-ADVISER. However, in some applications (AMGmk and NBody), some of the bottlenecks identified by OMP-ADVISER are not listed as performance bottlenecks in VTune. Among the tested applications, VTune identifies four applications with high runtime overheads. OMP-ADVISER reports high task creation overheads in two of these applications (Knapsack and KDTree). In these applications, OMP-ADVISER's parallelism profile identifies the task directive suffering from high task creation overhead. In contrast, VTune's report does not specify the exact task directive causing high runtime overheads.

In nine of the tested applications shown in Table 3.6, VTune reports memory subsystem issues (*i.e.*, True and False sharing, NUMA issues, *etc.*). Similarly, OMP-ADVISER's differential

[20]We did not include regions that belong to system libraries or the OpenMP runtime.

analysis identifies program regions with parallel work inflation in these applications. However, we could not determine these reports' effectiveness without having some ground truth on how addressing the reported bottlenecks may impact the program's performance. Overall, Intel Advisor and VTune are effective in identifying threading issues, program hotspots, runtime overheads, and unintended contention. However, in some tested applications (AMGmk, Knapsack, and KDTree), OMP-ADVISER's feedback pinpoints the region in the program with scalability bottlenecks. Further, while execution specific parallelism measurements are effective in identifying bottlenecks in the current execution, OMP-ADVISER's logical parallelism measurements provide additional insight into the program's scalability.

3.10 Limitations

The task creation measurements by OMP-ADVISER is a first-order approximation. Our measurements include the runtime costs to allocate and initialize a task. However, it does not include the runtime cost to check for task dependency and the cost associated with the task scheduler. Given this limitation, OMP-ADVISER has still proven useful in identifying programs with high tasking overheads. However, our analysis may underestimate the tasking overheads for some applications, especially OpenMP applications that use many dependencies per task. Constructing a more accurate estimate may prove useful in identifying tasking overhead issues in such applications.

While OMP-ADVISER can identify regions that must be optimized to address program bottlenecks, ultimately, it is the programmer that has to devise strategies to parallelize reported regions. Parallelizing a reported region may not be possible due to inherent dependencies in the application. In practice, parallelizing two different program regions may require vastly different amounts of effort. It would be interesting to explore expanding OMP-ADVISER to consider the feasibility of parallelizing different regions and incorporate it in the regions reported by automatic what-if analysis. We have currently incorporated a simple block listing feature where the developer can block program regions considered for what-if analysis if they deem it too challenging to parallelize concretely.

Further, after the user parallelizes a region, it is their responsibility to check that their changes have introduced no new bugs. At the moment, we can provide some assistance by running a suite of bug detection tools (*i.e.*, data race detector) on the optimized program. An interesting direction for future work is to equip OMP-ADVISER with the ability to synthesize parallelization strategies for some serialization bottlenecks automatically.

3.11 Summary

In this chapter, we presented the details of OMP-ADVISER, a novel performance analysis tool for OpenMP applications. OMP-ADVISER's performance model is comprised of the program's OSPG along with fine-grained measurements using hardware performance counters. Using its performance model, OMP-ADVISER computes the program's parallelism profile, including the program's inherent parallelism and the parallelism of each static OpenMP directive in the program. OMP-ADVISER parallelism-centric view of the program simplifies identifying serialization bottlenecks. OMP-ADVISER's performance model enables its what-if analysis technique. Using what-if analysis, OMP-ADVISER can estimate improvements in program parallelism if selected regions in the program are hypothetically parallelized. Further, OMP-ADVISER's automatic what-if analysis techniques alleviate the programmer's need to identify which regions to parallelize to reach a target parallelism. Our performance model includes measurements to estimate runtime task creation costs, which ensures that the optimization candidates reported by OMP-ADVISER perform work at a high enough granularity that can amortize the costs of increasing the region's parallelism. Lastly, OMP-ADVISER's differential analysis technique is based on the observation that applications with unintended resource contention may exhibit it as secondary effects of execution that do not manifest in the application's oracle. Our proposed differential analysis technique can identify scalability bottlenecks by comparing the performance model of the application's parallel execution with its oracle execution.

Chapter 4

On-the-fly Apparent Data Race Detection with OMP-RACER

OpenMP applications, similar to other multithreaded programs, are susceptible to various correctness bugs caused by nondeterministic execution order of threads. Bugs such as deadlocks, livelocks, and data races belong to this category. In particular, the execution of a parallel application may have many possible thread interleavings, and the bug may only manifest in a small fraction of those interleaving. The number of thread interleavings grows exponentially with the number of threads and instructions in the application, which makes it challenging to identify and reproduce such bugs using standard software testing techniques.

This chapter introduces OMP-RACER, a novel on-the-fly apparent data race detector for OpenMP applications. Apparent data races are those races that manifest in a program considering the logical series-parallel relations of the execution. By identifying apparent races, OMP-RACER can detect races that occur not only in the observed schedule but also in other schedules for a given input. OMP-RACER maintains information about previous accesses with each memory access and uses the OSPG to check if they can logically execute in parallel.

4.1 Background

Data races are a common source of bugs in multithreaded applications, including OpenMP programs. A program has a data race if, during program execution, there exist two parallel memory accesses to a shared memory location, where at least one of them is a write. Programming languages such as C/C++ [6, 180] consider a data race as undefined behavior. An undefined behavior removes all restrictions from the compiler and the code it generates, which may result in unintended behaviors and potentially even lead to catastrophic consequences. Further, data races cause the program execution to be nondeterministic and dependent on the underlying system's memory model, often leading to bugs and portability issues.

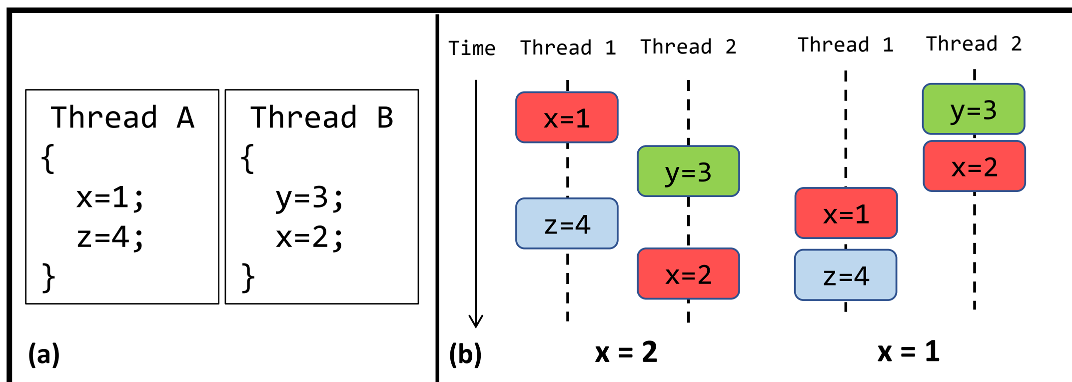


Figure 4.1: (a) Code snippet illustrating multithreaded program with a data race on shared variable x. (b) Two possible execution traces on a system with two threads.

A parallel application may have many thread interleavings (*i.e.*, possible schedules), which grow exponentially with the number of threads and instructions per thread. A data race may manifest in only a small subset of them. Thus, identifying data races manually or using standard software testing techniques is often ineffective. Due to a large number of possible thread interleavings, these approaches fail to detect many possible data races in the application.

Next, we formally define data races, describe different types of data race detectors, and provide a brief classification of different types of data races.

Definition 4.1.1. (Data Race) During program execution, a data race occurs when two memory accesses:

- Access the same memory location.
- At least one access is a write.
- Both accesses are in parallel with no ordering.

Two memory accesses may be ordered if one access always executes before the other access. For OpenMP applications, this ordering may be satisfied by one of the following criteria: (1) both memory accesses belong to the same serial fragment of execution, (2) there exists an OpenMP synchronization construct (*i.e.*, barrier, taskwait, *etc.*) that ensures one access always finishes before the other access, and (3) a mutual exclusion primitive such as a lock or OpenMP critical construct ensures both memory accesses are never executed in parallel. For example,

Figure 4.1(a) shows a simple multithreaded program with a data race on the shared variable x. Figure 4.1(b) shows two possible execution traces of the program on a system with two threads, illustrating that the shared write accesses to variable x are unordered. Hence, constituting a data race.

Data races in an application are most often undesired, and identifying them with standard software testing techniques is mostly inadequate. Hence, researchers have proposed data race detection tools to assist developers in identifying data races. In practice, proposed solutions make different trade-offs to design practical data race detectors to identify data races with acceptable overheads. Next, we provide a characterization of data races and provide a brief background on prior data race detection tools.

4.1.1 Characterizing Data Races

Data races can be characterized into two categories: feasible and apparent [146]. Feasible data races follow the intuitive notion of a data race that manifests on an actual program trace for some input. To detect a feasible data race requires assessing both the computation and the behavior of synchronization primitives in the application. In contrast, apparent data races are based solely on the semantics of the synchronization primitives of the program. However, locating all feasible is data races in an application is computationally hard [146]. Intuitively, multithreaded applications have exponential number of thread interleavings that grows in proportion to the number of threads and instructions per thread [1]. Since locating all feasible data races in an application requires analyzing each interleaving, it makes solving this problem computationally intractable. In contrast, detecting apparent data races is simpler. Detecting apparent data races requires analyzing the semantics of the synchronization primitives of the application into account. Hence it requires capturing the logical-series parallel relations in an application. These relations are a property of the program for a given and are independent of thread interleavings in the application.

Apparent data races are an approximation of a program's feasible data races. That is, every feasible data race is an apparent data race. However, the opposite is not true. Figure 4.2(a)

[1] Approximately $t^{t \times n}$ interleavings for a program using t threads with n instructions per thread

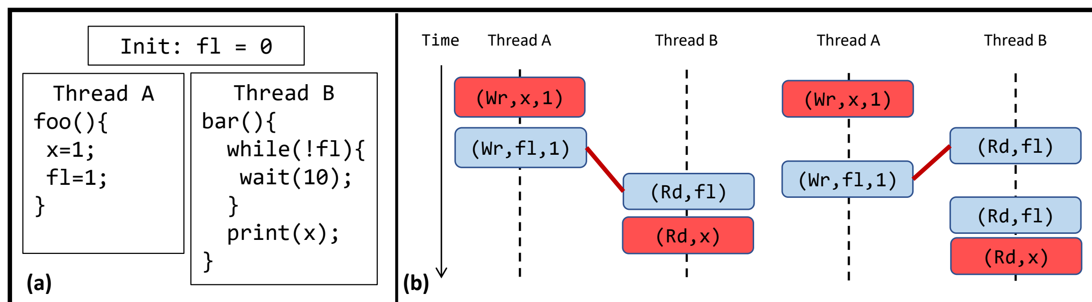


Figure 4.2: (a) Code snippet illustrating multithreaded program with two apparent data races on shared variables `x` and `fl`. (b) Two possible execution traces of the code snippet in (a), illustrating that the data race on variable `fl` is feasible.

provides an example. In this code snippet, we observe two apparent data races on shared variables `x` and `fl`. If we explore all possible execution traces for this application, it becomes clear that the read access to variable `x` always follows the write access to it. Hence, the apparent data race on variable `x` is not a feasible data race. This ordering between these two accesses to the variable `x` is enforced by the while loop checking the value of the variable `fl` since the execution in the function `bar` will not proceed until the value of `fl` is set by the last instruction in function `foo`. In contrast, the data race on variable `fl` is feasible as shown by two execution traces in Figure 4.2(b).

As shown in the example code snippet, feasible data races are actual races in the sense that they occur for some possible execution of the application. In contrast, an apparent data race may not occur in any actual program execution trace. However, it has been proven that if a program has apparent data races, it must have at least one feasible data race [144]. Further, for a class of parallel applications with commutative critical sections, known as Abelian programs [44], every apparent race is also a feasible race. Hence, many data race detectors, including our work in this chapter, focus on identifying simpler to locate apparent data races.

Ideally, we would want to design data race detection algorithms that are both sound and complete. A soundness guarantee implies that the algorithm does not report false negatives. That is, if the program has a data race, the algorithm will detect and report it. In contrast, a completeness guarantee implies that the data race detection algorithm will not report false positives. However, achieving both guarantees is not pragmatic as it is equivalent to detecting all possible feasible data races in the program, which is shown to be NP-hard [146].

Further, a data race detection algorithm may detect data races at three different granularities: (1) per-schedule, (2) per-input, or (3) per-program where each granularity provides stronger guarantees than the previous granularity. First, a per-schedule data race detection algorithm aims to detect data races in a given program schedule (*i.e.*, thread interleaving). per-schedule data race detectors tend to focus on finding feasible data races. Second, a per-input data race detection algorithm aims to identify data races for a given program input across all possible schedules. From this definition, it becomes clearer why a feasible per-schedule data race detector is not practical to identify data races per-input. Instead, most per-input data race detectors focus on detecting apparent data races. Lastly, a per-program data race detection algorithm aims to detect all data races in the program across all inputs and schedules.

4.1.2 Static Data Race Detectors

These tools detect data races in an application by using static analysis techniques. A static data race detector can identify data races across all possible program schedules and inputs. However, this comes at the price of increased false positives. Static data race detection is a well-studied problem. Prior work has used static analysis to identify races in multithreaded programs [9, 66, 140, 158]. More recently, several lines of work have utilized static analysis to identify data races in OpenMP programs. Since these tools exclusively target OpenMP applications, they can identify data races in OpenMP with higher accuracy when compared to their more general-purpose counterparts [25, 42, 191]. Static data race detection algorithms provide per-program guarantees, and some provide soundness guarantees. However, their main drawback is that they may report a large number of false positives. Requiring the developer to inspect the validity of a reported data race manually.

4.1.3 Dynamic Data Race Detectors

A dynamic data race detector uses dynamic analysis techniques to identify data races. Generally, a dynamic data race detector runs the program and analyzes its execution trace to report data races. Unlike a static data race detector, a dynamic data race detector tends to have lower false positive rates since the tool has access to the program input and the application's execution trace. A large body of work explores dynamic data race detection for multithreaded applications [69, 168, 175].

There are two types of dynamic data race detection algorithms: on-the-fly and post-mortem. On-the-fly data race detectors perform data race detection analysis during program execution. In contrast, some data race detectors perform post-mortem analysis on a captured program trace after the program's execution to discover data races [21, 145]. The main trade-off between the two approaches is that the former does not require generating log files to capture the execution trace for later analysis. The latter approach tends to have lower runtime and memory overheads since it performs data race detection analysis after its execution.

Several data race detection algorithms focus on identifying feasible data races for a given program execution trace. This results in per-schedule data race detection algorithms. The data race detectors in this category are based on analyzing the happens-before [112] relation in the captured trace. FastTrack [69] improved upon the vector clock based method of capturing happens-before relations to reduce data race detection overheads. ThreadSanitizer [175], uses a hybrid approach inspired by FastTrack with the aim of targeting large-scale applications at the cost of introducing some false positives.

In contrast, per-input dynamic data race detection algorithms analyze the logical series-parallel relations in the application trace, resulting in detecting apparent data races in the application. This line of work mainly focuses on structured parallel programming languages and the use of series-parallel graphs to model the logical series-parallel relations[68, 161]. Subsequent works expand upon these techniques to support a larger class of parallel programs and perform data race detection in parallel [162, 194]. offset-span labeling [132] uses a labeling scheme in fork-join programs to summarize the program's series-parallel graph and uses it to detect apparent races in fork-join parallel applications.

4.1.4 Prior Dynamic Data Race Detectors for OpenMP Applications.

While a data race detector written for multithreaded programs [98] may be used to identify data race in OpenMP applications, since it does not capture the semantics of the OpenMP API, it will not be very effective at detecting data races. Hence, in recent years there has been a growing interest in developing data race detectors for OpenMP applications [20, 21, 25, 42, 81, 191].

Several dynamic data race detection tools have been proposed for OpenMP programs. Archer [20] is the state-of-art dynamic data race detector for OpenMP programs. Archer builds

upon ThreadSanitizer [175] by extending it to support OpenMP semantics. Since Archer is a per-schedule data race detector, it may require multiple executions of the same program with different number of threads and the same input to detect the program's data races.

SWORD [21] is a post-mortem apparent data race for OpenMP applications. It uses the offset-span labeling technique [132] to detect if two memory accesses may execute in parallel. As a post-mortem data race detector, SWORD uses a constant amount of memory overhead per worker thread during program analysis. However, the main drawback of SWORD is the size of log files and the increasing amount of time needed to perform post-mortem analysis. Further, SWORD does not detect data races in OpenMP programs that use tasking directives. ROMP [81], expands upon the offset-span labeling technique by adapting it to encode OpenMP task-based constructs and design an on-the-fly OpenMP data race detector. While ROMP aims to address some of SWORD's limitations, its public prototype [80] only works for microbenchmarks and fails to analyze larger applications.

4.2 Overview of OMP-RACER

We describe OMP-RACER, a novel per-input apparent data race detector for OpenMP programs. We discussed several prior OpenMP data race detectors. Our goal is to design an on-the-fly apparent data race detector where the program is run once for a given input and identify data races that can occur in other thread interleavings of the application. We also want to detect data races in a large class of OpenMP applications with structured and unstructured constructs, including task-based constructs.

To detect apparent race, OMP-RACER must consists of the following components:

- **A mechanism to check if two shared memory accesses may execute in parallel**. A program's OSPG captures the logical series-parallel relations between all program fragments, which are represented by W-nodes in the OSPG. By constructing a program's OSPG on-the-fly and attributing each memory access to its corresponding W-node in the OSPG, we can identify if two memory accesses may execute in parallel by performing a least common ancestor (LCA) query.

- **Managing access histories**. A dynamic data race detector must keep track of previous memory accesses in the program trace to check if the current memory access is involved in a data race with previous accesses. OMP-RACER provides two modes for managing access histories with different trade-offs. The fast mode uses a constant amount of memory overhead for each memory location accessed by the program. However, it is limited to a subset of OpenMP programs that do not employ locks and use taskwaits in a fully nested manner. In contrast, precise mode's memory overhead grows in proportion to the program's nesting level and the size of its lockset for a given memory location. However, it is able to detect data races in a larger class of OpenMP programs.

4.2.1 Illustrative Example

Consider the example OpenMP code snippet in Figure 4.3(a). This code snippet modifies the shared variables `x` and `y`. This code snippet creates two sibling tasks at lines 5 and 9 in Figure 4.3(a). Based on OpenMP semantics, in the absence of task dependency clauses, two sibling tasks may execute in parallel. Hence, the program snippet has a write-read data race on shared variable `x` because of write access at line 7 and the read access at line 11 in Figure 4.3(a). The two accesses may execute in parallel if there a sufficient number of worker threads to schedule the tasks on different worker threads.

To detect data races on-the-fly, a data race detector implements the following two mechanisms. First, it must track how different serial fragments of the application's execution trace execute relative to each other. For example, the code fragment at lines 3-4 in Figure 4.3(a) executes serially before the code fragment at lines 6-8 since the worker thread executing the single block must execute lines 3-4 before creating the task at line 5. As described in Section 4.1.3, different data race detectors employ different strategies to capture these series-parallel relations (*i.e.*, happens-before relations, offset-span labeling, series-parallel graphs, *etc.*). For OMP-RACER, we use the OSPG to capture the logical series-parallel relations between different program fragments. Second, an on-the-fly data race detector must track previous memory accesses for each memory location in the program execution. For example, as shown in Figure 4.3(b), an execution trace with two worker thread of the program snippet in Figure 4.3(a), when the current memory access is the read operation on variable `x` (time 4 in Figure 4.3(b)), to identify the data

race on x, an on-the-fly data race detector must keep track of the previous write access to it (time 3 in Figure 4.3(b)).

Capturing Logical Series-Parallel Relations. To detect apparent data races on-the-fly in OpenMP application, we need a data structure that encodes the logical series-parallel relations in the program execution trace. Since logical series-parallel relations are independent of a particular thread interleaving in the application, using them enables a data race detector to detect data races per-input. For example, in Figure 4.3(a), if the code snippet is executed on a system with one worker thread, the two sibling tasks will execute serially by the same worker thread. However, based on the semantics of OpenMP, the two tasks logically execute in parallel. Thus, by encoding the logical series-parallel relations between program fragments, a data race detector can detect data races per-input, which is a stronger guarantee than per-schedule data race detectors.

As described in Chapter 2, the OSPG is a data structure that captures the dynamic execution of an OpenMP program as a set of program fragments and encodes the logical series-parallel between all pairs of fragments. OMP-RACER uses the OSPG to capture the logical series-parallel relations in an OpenMP application. OMP-RACER executes with the program under test and creates its OSPG on-the-fly. A different W-node represents each program fragment in the OSPG. As described in Section 2.5, to check for the logical series-parallel relation between two program fragments, the pair corresponding W-nodes an OSPG can be queried by performing an LCA query. For example, Figure 4.3(c) shows the OSPG that OMP-RACER constructs during the execution of program snippet in Figure 4.3(c). As illustrated, the three program fragments are represented by W-nodes $W1 - W3$, where the OSPG captures the serial relations between $W1$ and $W2 - W3$ and the parallel relation between the two sibling task fragments $W2$ and $W3$.

Access History Management. To identify apparent data races, OMP-RACER must keep track of previous memory accesses in the program's execution trace. A naive but straightforward approach is to maintain all previous memory accesses in the OpenMP program for each memory location. While simple, this approach is impractical as the amount of access history per memory location grows with the number of dynamic load and store instructions in the program. OMP-RACER uses two different access history management modes to limit the amount of metadata

stored for each memory location: fast and precise mode. The main advantage of fast mode is that amount of metadata per memory location is constant. In fast mode, for each memory location OMP-RACER maintains up to two previous parallel read accesses and the latest write access. Fast mode is applicable to programs with structured parallelism. That is, the program uses fully nested OpenMP taskwaits and has no OpenMP critical constructs. In contrast, when using precise mode, OMP-RACER no longer maintains a constant amount of metadata per memory location; however, it can detect data races in OpenMP programs that use taskwaits and critical constructs with no restrictions needed. In precise mode, the metadata used per memory location grows in proportion to the program's task nesting level and its lockset size.

The program snippet in Figure 4.3(a) does not use OpenMP taskwait or critical construct. Hence, it uses fast mode for access history management. As shown in the third column of the table in Figure 4.3(b), OMP-RACER keeps track of memory accesses by maintaining an access history (*i.e.*, variables `x` and `y` in Figure 4.3(a)). In fast mode, the access history metadata for each memory location is comprised of up to three entries, the W-node corresponding to the latest write operations, and up to two W-nodes, which correspond to previous parallel read operations (represented as a 4-tuple in Figure 4.3(b)). The important updates to the access history metadata happen at time steps 3 and 4, shown in the program trace in Figure 4.3(b). When the program execution reaches line 11 in Figure 4.3(a), it performs a read operation on variable `x` (time 4 in Figure 4.3(b)). At this point, OMP-RACER retrieves the access history corresponding to the location of `x` and updates it with the current read operation using the W-node $W3$. Next, OMP-RACER looks for an apparent data race by checking if the current read access is involved with the previous write access' corresponding W-node in the retrieved metadata, W-node $W2$. Based on the constructed OSPG in Figure 4.3(c), W-nodes $W2$ and $W3$ may execute in parallel. Hence, OMP-RACER reports a write-read data race on the memory access to variable `x`.

While both access history management strategies employ significantly less memory overhead than the naive approach, When comparing the two modes, the main trade-off between them is that the fast mode uses less memory when compared to the precise mode. In contrast, the precise mode is applicable to a larger class of OpenMP applications. To choose between the modes, OMP-RACER uses an analysis pass that involves a quick execution of the program under test to check if the program has fully nested taskwaits and does not use critical constructs. If so,

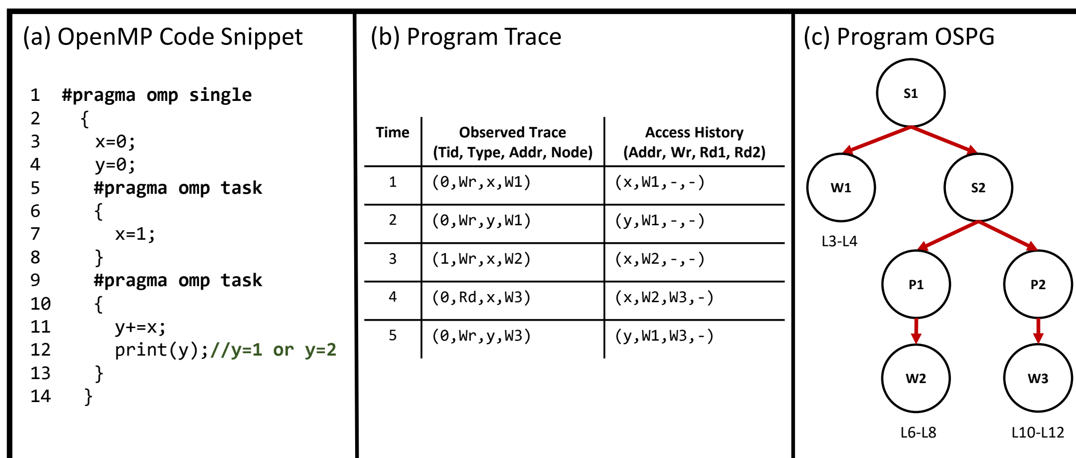


Figure 4.3: (a) OpenMP program snippet with a write-read data race on shared variables x (b) Possible execution trace of the code snippet in (a), when executing on a 2 threaded system. (c) The code snippet's OSPG. The code fragment each W-node represents is shown below it.

it uses fast mode; otherwise, it switches to precise mode to ensure that it correctly identifies data races in the program. We describe the two modes in more detail in Sections 4.3.1 and 4.3.2 respectively.

4.3 Data Race Detection Algorithm

The goal of OMP-RACER is to detect apparent data races in OpenMP applications. OMP-RACER runs with the program under test and constructs its OSPG in parallel. To update its access history metadata, OMP-RACER instruments memory the program's read and write operations. With a single execution of the program, OMP-RACER detects apparent races that manifest not just in the program's current execution trace but also in other possible thread interleavings for a given input. In this section, we first describe OMP-RACER data race detection algorithm in fast mode. Afterward, we describe how we extend OMP-RACER to detect data races in programs with unrestricted OpenMP taskwaits and critical constructs.

4.3.1 Data Race Detection in Fast Mode

In fast mode, OMP-RACER's data race detection algorithm operates on OpenMP programs that use perfectly nested taskwaits and do not use the critical constructs. When limiting the use of taskwaits to being fully nested, its uses do not result in unstructured parallelism (See

Algorithm 17: Analysis algorithm checking if using fast mode is safe for an OpenMP program. This analysis requires OMP-RACER to run the program once before running its data race detection algorithm using fast or precise mode.

```
1  procedure OnProgramStart ()
2  |  use_fast_mode ← false
3  |  RegisterCallback(on_node_pop, OSPGNodePop)
4  |  RegisterCallback(critical_begin, CriticalBegin)
5  |  ConstructOSPG()
6  procedure OSPGNodePop (node)
7  |  if NodeType(node) = ST-node ∧ node.st_val = 0 then
8  |  |  use_fast_mode ← False
9  |  end
10 procedure CriticalBegin (lock)
11 |  use_fast_mode ← False
```

Section 2.4.6), Thus resulting in a simpler data race detection algorithm that uses a constant amount of metadata per memory location. When possible, OMP-RACER uses fast mode over precise mode.

OSPG Construction in Fast Mode

OMP-RACER constructs the program's OSPG during program execution. A program's OSPG is constructed incrementally and in parallel as the program executes. The OSPG construction algorithm in fast mode is similar to the OSPG construction described in Section 2.4 and its construction details in Section 3.3.1. Except it does not create ST-nodes. Instead, all ST-nodes are treated as S-nodes.

Choosing Between Fast Mode and Precise Mode. A question that arises when using OMP-RACER's data race detection algorithm is when it is safe to use fast mode over precise mode? For example, using fast mode instead of precise mode in programs that do not have perfectly nested taskwaits will incorrectly capture the relation of a pair of logically parallel program fragments as serial. If these two program fragments have a conflicting memory access, OMP-RACER will miss the data race, reporting a false negative. Further, in fast mode, OMP-RACER is not aware of mutual exclusion locks. Thus, when two logically parallel fragments access a shared location protected by a lock, OMP-RACER in fast mode will incorrectly report a false positive.

Algorithm 17 illustrates the check performed by OMP-RACER to deem if using fast mode is safe. The analysis algorithm requires running the program under test once to determine which data race detection mode is suitable for a program and a given input. The analysis algorithm sets a flag variable (use_fast_mode) to indicate if OMP-RACER can use fast mode for data race detection. The analysis works as follows. When the program begins execution, the algorithm sets the value of the flag variable to true, registers two callback functions, before starting to construct the program's OSPG (lines 1-5 in Algorithm 17). The first callback, OSPGNODEPOP, is called whenever during the OSPG construction algorithm, a node goes out of scope, and it gets popped from the per-thread pop an OSPG node (See Section 3.3.1 for details on the OSPG construction algorithm). The second callback, CRITICALBEGIN, is called whenever the program encounters an OpenMP critical construct.

The callback OSPGNODEPOP is used to check if the taskwaits in the program are perfectly nested. Recall that the OSPG construction algorithm creates an ST-node whenever a parent task creates its first child task in the program or after an OpenMP synchronization construct (See Section 2.4.6) with a starting `st_val` of zero. During the program execution, whenever the current task encounters a taskwait, it sets the corresponding ST-node's `st_val` to -1. Thus, it follows that a program's OSPG ST-nodes are all set to -1 if and only if all taskwaits are fully nested in the program, which implies structured parallelism semantic where each parent task waits for the completion of all descendant tasks. The OSPGNODEPOP callback checks to see if this property is maintained during the construction algorithm whenever an OSPG node goes out of scope and is popped from the OSPG. As soon as the algorithm detects an ST-node with an `st_val` of zero, the flag variable use_fast_mode is set to false (lines 6-9 in Algorithm 17). Alternatively, we can construct the entire program's OSPG in memory and then traverse all its nodes and check if all ST-nodes have a value of -1 before the program ends. However, the latter approach requires has higher memory overheads since it requires maintaining the entire OSPG in memory until the end of the program.

The callback CRITICALBEGIN is used to check if the program has any OpenMP critical constructs for a given input. As soon as the program encounters a critical constructs, the algorithm identifies that the use of fast mode is not safe and it sets the flag variable use_fast_mode to false (lines 10-11 in Algorithm 17).

Algorithm 18: The ISPARALLEL relation when using OMP-RACER's fast mode for a pair of W-nodes where W_l is left of W_r.

```
1  function ISPARALLEL(W_l, W_r)
2      L ← LCA(W_l, W_r)
3      L_lc ← L.childTo(W_l)
4      if L_lc.type = S-node ∨ L_lc.type = W-node then
5          return False
6      end
7      L_rc ← L.childTo(W_r)
8      if L_lc.type = P-node then
9          if DIRECTEDPATH(L_lc, L_rc) = True then
10             return False ▷ series due to task dependencies
11         end
12         return True
13     end
```

Computing Series-Parallel Relations in Fast Mode

Using the OSPG, we can check if two W-nodes logically execute in parallel. Since when using fast mode, the program's OSPG does not have any ST-nodes, performing this check is more straightforward than the ISPARALLEL relation described in Section 2.5 and does not require storing `st_val` values in each nodes. Note that for each memory access, OMP-RACER keeps track of which W-node it corresponds to; Thus to compute the series-parallel relations between a pair of memory accesses is equivalent to finding the series-parallel relation of the pair's corresponding W-nodes.

Algorithm 18 shows the steps involved for performing this check given a pair of W-nodes, W_l and W_r where W_l is to the left of W_r. This procedure is as follows. (1) Compute the least common ancestor (LCA) of the two nodes W_l and W_r (line 2). (2) Identify the left child of the LCA node on the path to W_l and (line 3). If the left child of the LCA, node L_{lc}, is an S-node or a W-node, then the two W-nodes logically execute in series (lines 4-6). This follows directly from the properties of OSPG nodes (See Section 2.3). (3) If the left child of the LCA on the path to W_l is a P-node, the pair of W-nodes may execute in parallel unless they are serialized by task dependencies (lines 8-12 in Algorithm 18). To perform this check, the algorithm calls the DIRECTEDPATH function to perform a reachability query comprised of only dependency edges between the two nodes. If a directed path exists, the pair of W-nodes execute in series. Otherwise, they logically execute in parallel.

Algorithm 18 is simpler than Algorithm 1, as it does not need to consider the presence of ST-nodes. Further, the ISPARALLEL relation in fast mode does not require checking the nesting levels of the left W-node and the `st_val` values of nodes on the path to the LCA node. This also results in a simpler check for task-dependency serializations between the two algorithms. This simplification occurs because, in fast mode, the algorithm can safely assume that a parent task's continuation starts execution only after all the parent task's descendant tasks complete execution. Thus when a dependency clause serializes two sibling tasks, it also serializes all their descendants because of the fully nested taskwaits in programs analyzed in fast mode.

Access History Management in Fast Mode

OMP-RACER maintains access history metadata with each shared memory address. In fast mode, OMP-RACER maintains up to three W-nodes per memory location. One W-node, W_1, corresponds to the latest write operation. Two W-nodes, R_1 and R_2, correspond to two previous parallel read accesses, resulting in a constant amount of metadata per memory location used by the program under test. The access history management strategy used by OMP-RACER in fast mode is similar to prior work [132, 162, 194], where the authors have explored on-the-fly data race detection for task-parallel and fork-join applications.

For every shared memory location x referenced by the program, OMP-RACER maintains a 3-tuple $\langle W_1, R_1, R_2 \rangle$ for its access history. Initially, all three entries are set to empty, $\langle -, -, - \rangle$. Until the first data race is detected, OMP-RACER keeps the following invariants. For every write operation to x, the W_1 entry is updated to the W-node that corresponds to the latest write operation. The intuition behind keeping track of only the latest write access W-node is that all previous write accesses, including the most recent one, must have executed in series relative to each other. Otherwise, we would have two parallel reads, breaking the assumption that no data race has been detected so far.

In contrast, multiple parallel read accesses to location x may exist without breaking the assumption of the absence of data races so far in the execution[2]. In such cases where n parallel read W-nodes, $R_{1..n}$, to location x exist, the access history entries R_1 and R_2 are chosen

[2] two parallel reads accesses to the same location does not constitute a data race

from $R_{1..n}$ such that $L = LCA(R_1, R_2)$, is closer to the root node than $L' = LCA(R_1, R_K)$ or $L'' = LCA(R_2, R_K)$ for any $R_K \in R_{1..n}$. This pruning criteria is similar to the SPD3 algorithm proposed by Raman *et al.* [161, 162]. Further, updates to the metadata entries must be performed atomically.

The intuition behind such a choice is that if any future memory access is involved in a data race on location `x` with prior any prior n parallel reads, $R_{1..n}$, then it will also have a data race with either R_1 or R_2. Thus, it is sufficient to maintain two previous parallel reads for a given memory location without missing a data race on location `x`. in fast mode, the `isParallel` relation has the transitive property. Consider the read access R_k that is a member of $R_{1..n}$. By definition, `isParallel(`R_1, R_k`) =` $True$ and `isParallel(`R_2, R_k`) =` $True$. If a future write access, W_j, is involved in a data race with R_k, implying that `isParallel(`R_k, W_j`) =` $True$, based on the transitive property, `isParallel(`R_1, W_j`) =` $True$ and `isParallel(`R_2, W_j`) =` $True$. Indicating that under this access history management strategy, OMP-RACER will still detect the data race despite not including R_k in the access history metadata.

Since two memory accesses before and after a global synchronization construct can not be involved in a data race, all metadata entries can be safely set to empty after the program exits an OpenMP parallel region.

Data Race Detection Algorithm in Fast Mode

OMP-RACER runs with the OpenMP program under test and constructs the program's OSPG on-the-fly. The OSPG is used to compute the ISPARALLEL function shown in Algorithm 18. OMP-RACER instruments memory accesses in the program to update the access history metadata and to identify if the current access is involved in a data race. The function ONMEMORYOPERATION in Algorithm 19 is invoked whenever the OpenMP program encounters a new memory access event. Algorithm 19 performs two main tasks. First, it checks if the current memory access is involved in a data race with the stored W-node entries stored in the access history metadata (lines 3-17). Second, after performing the check for data races, the algorithm updates the access history metadata corresponding to the current memory access (lines 18-41)

Algorithm 19: OMP-RACER data race detection algorithm in fast mode. The function is invoked whenever the program accesses a memory location.

```
1  function OnMemoryOperation(tid, addr, type)
2      ⟨W1, R1, R2⟩ ← GetAccessHistory(addr)
3      Cur ← GetCurrentWNode(tid)
       ▷ Check if current access is involved in a data race
4      if type = Read ∧ IsParallel(Cur, W1) = True then
5          ReportDataRace(addr, Read, Write)
6      end
7      else
           ▷ Current memory operation is a Write
8          if IsParallel(Cur, W1) = True then
9              ReportDataRace(addr, Write, Write)
10         end
11         if IsParallel(Cur, R1) = True then
12             ReportDataRace(addr, Write, Read)
13         end
14         if IsParallel(Cur, R2) = True then
15             ReportDataRace(addr, Write, Read)
16         end
17     end
       ▷ Update Access History Metadata
18     if type = Read then
19         if R1 = ⟨−⟩ then
20             R1 ← Cur
21         end
22         else if R2 = ⟨−⟩ then
23             if IsParallel(Cur, R1) = True then
24                 R2 ← Cur
25             end
26             else
27                 R1 ← Cur
28             end
29         else if IsParallel(Cur, R1) = False ∧ IsParallel(Cur, R2) = False
            then
30             R1 ← Cur
31             R2 ← ⟨−⟩
32         else if IsParallel(Cur, R1) = True ∧ IsParallel(Cur, R2) = True
            then
33             LCA12 ← LCA(R1, R2)
34             LCAc1 ← LCA(Cur, R1)
35             LCAc2 ← LCA(Cur, R2)
36             if LCAc1 > LCA12 ∧ LCAc2 > LCA12 then
37                 R1 ← Cur
38             end
39     end
40     else
           ▷ Current memory operation is a Write
41         W1 ← Cur
42     end
```

The algorithm starts by first retrieving the access history metadata corresponding to the current memory access (line 2). In fast mode, this entry is comprised of up to three W-nodes, W_1, which corresponds to the previous write operation, and R_1 and R_2, which corresponds to two previous read operations. Next, the algorithm retrieves W-node, Cur, the program fragment the current memory operation belongs to. Depending on the type of the current memory access, lines 4-17 check for apparent data races. If the current access is a read operation, the algorithm checks if the current access is involved in a data race with the latest write operation, W_1. When the current access is a write operation, checking for a data race with entries R_1 and R_2 is not needed since multiple read operations will not constitute a data race. If the current access is a write operation, in addition to checking for data race with the latest write access, the algorithm checks if the current access is involved in a $Read/Write$ data race with the two previously store read entries, R_1 and R_2.

Afterward, OMP-ADVISER updates the metadata data entries to include the current memory access, if required. If the current memory access is a write, we update the W_1 entry with the current access to maintain the invariant of W_1 storing the latest write operation (line 41). If the current memory access is a read operation, the update criteria are slightly more involved. When the current read access may execute in parallel with both reads stored in R_1 and R_2, we must keep two of the read accesses in the access history metadata. OMP-ADVISER maintains the invariant of storing a pair of read accesses such that their LCA node is closest to the root node of the OSPG (*i.e.*, the pair's LCA node subsumes the other two pairs LCA node). The data race detector performs this check at lines 32-38. Replacing R_1 with the current access' W-node, Cur, if the LCA criteria is met. In contrast, if the current access executes is series with both R_1 and R_2 the algorithm removes clears both entries, replacing R_1 with Cur

Time and Space Overhead. The running time of OMP-RACER's data race detection algorithm can be broken down into two components. First, the cost of OSPG construction. Second, the cost of calling the ONMEMORYOPERATION on each memory operation. Creating the OSPG requires time proportional to the number of program fragments. Since program fragments start/end when the program trace encounters an OpenMP construct, the number of program fragments is proportional to the number of dynamic OpenMP constructs encountered during the program execution. The height of the OSPG, h, is proportional to the task nesting

level of the program (including both implicit and explicit tasks) since creating a child task by a parent task results in adding a constant number of nodes to the OSPG. Hence, the overall cost of creating the OSPG is proportional to the running of the program under test.

The number of times the function ONMEMORYOPERATION is called is equal to the number of shared memory access operations performed within OpenMP parallelism constructs in the program, which is proportional to the dynamic trace of the program. The running time of each call to ONMEMORYOPERATION is $O(h)$, since all operations except the calls to ISPARALLEL and LCA perform a constant amount of work. The LCA operation takes a pair of W-nodes, which are leaf nodes in the OSPG. Performing an LCA query between a pair of leaf nodes can be accomplished by traversing each node's parent on the path to the root node and reporting the first common node between the two paths as their LCA. The height of the OSPG bounds the cost of this operation. Hence, the LCA running time is $O(h)$.

In the absence of task dependencies, the running time of ISPARALLEL is $O(h)$ since the algorithm does not call the DIRECTEDPATH function. The DIRECTEDPATH function performs a reachability query on the subgraph of sibling tasks and the dependency edges connecting them. For a set of sibling tasks with the size of k, up to k^2 unique dependency edges can exist. In a graph with k^2 edges, performing a depth-first traversal answers a reachability query in $O(k^2)$ time and $O(k)$ space. We can improve the running time of a reachability query to $O(k)$ by maintaining the task dependency subgraph's transitive closure using $O(k^2)$ space. By default, OMP-RACER uses the second approach. Hence the running time of ISPARALLEL is $O(h + k)$. In practice, task dependencies are often used to model stencil computations, where k is constant. In such cases, the running time of ISPARALLEL is $O(h)$.

Since the cost of an LCA query is one of the dominating factors in our algorithm's running time, it naturally raises the question of whether more efficient algorithms exist to perform this query. Finding faster algorithms for LCA queries on trees and DAGs has been explored in prior works. However, these solutions explore the problem in a serial setting [26, 46] or require preprocessing the entire DAG first [170]. Either limitation makes the algorithm not suitable to our parallel and on-the-fly data race detection setting. Nonetheless, a faster LCA algorithm can improve our data race detection algorithm.

The space overhead of the algorithm is bound by the access history size. Each access history entry has the size $O(1)$ since it uses the identifiers to up to three W-nodes. Since metadata entries can be safely discarded after a global synchronization construct at the end of an OpenMP parallel region, the number of access history entries is bound by the number of memory locations accessed within the parallel region, W_{set} (*i.e.*, its working set within the parallel region). Thus, OMP-RACER's data race detection algorithm in fast mode requires additional memory proportional to the working set size of the program under test.

4.3.2 Data Race Detection in Precise Mode

In precise mode, OMP-RACER data race detection algorithm applies to programs with unrestricted use of taskwaits and programs that use OpenMP locks.

Computing Series-Parallel Relations in Precise Mode

In precise mode, the program's OSPG may include ST-nodes. To capture the logical-series parallel relations in the program in precise mode, we need to use the ISPARALLEL relation as described in Section 2.5. There are two main differences between the ISPARALLEL relation used in fast and precise mode. First, the inclusion of ST-nodes is needed to correctly capture the logical series-parallel relations of program fragments in the presence of unrestricted use of taskwaits, which results in additional checks at lines 16-20 in Algorithm 1 compared to Algorithm 18. Second, the presence of unrestricted taskwaits impacts the check for serialization implied from a task dependency since we can no longer assume that all descendant tasks of a parent task complete when the parent task completes. This change is reflected in the additional check performed at lines 10-11 in Algorithm 1.

Access History Management in Precise Mode

In precise mode, it is no longer possible to maintain a constant amount of access history metadata per memory location. Two factors contribute to this change: First, the unrestricted use of taskwaits results in unstructured parallelism in the program. Thus unlike fast mode, we can no longer assume that the ISPARALLEL relation is transitive. Second, the use of locks requires

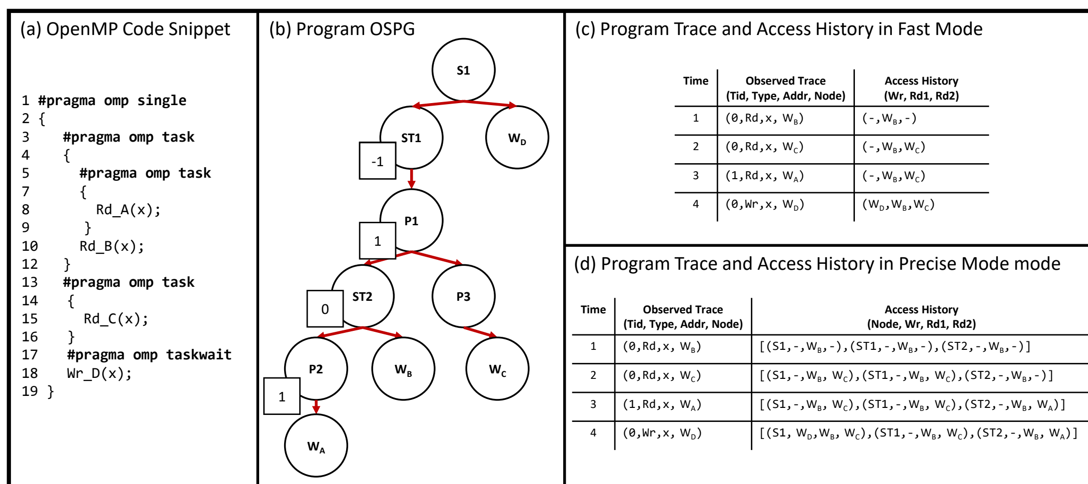


Figure 4.4: Illustration of an example OpenMP program snippet where the use of constant amount of access history metadata is not sufficient to detect the data race on variable x. (a) OpenMP program snippet. (b) The code snippet's OSPG. (c) Program trace with fast mode's access history metadata management. (d) Program trace with precise mode's access history metadata management.

the need to store additional access history metadata that grows in proportion to the lockset size[3]. This change is required since if two conflicting memory operations hold at least one common lock (*i.e.*, the intersection of the locks held between the two access is not the empty set), then the two memory operations do not constitute a data race. Hence, requiring OMP-RACER's data race detection algorithm to keep track of the locks held by the program's threads and updating the access history metadata entries accordingly. To ease explanation, we first start with an example lock-free program that does not have fully nested taskwaits and describes the changes required to the access history metadata. Next, we discuss the changes required to detect data races in the presence of locks correctly. Finally, we describe the update rules to the access history metadata in precise mode.

Consider the example OpenMP program snippet and its OSPG shown in Figure 4.4(a) and (b). This program is comprised of four serial functions where based on the prefix of their name, `Rd_` or `Wr_`, perform a read or write operation to the shared variable `x`, respectively. Since the program is missing a taskwait after line 9, no serial ordering between functions `Rd_A` and `Wr_D` exists. Hence, the program has a read-write data race on variable `x`.

[3]For a program that uses n unique locks; the lockset size is the size of its power set, *i.e.*, 2^n.

In this example program, using fast mode to manage access history metadata entries is not sufficient to detect some data races. We can no longer maintain three W-nodes (latest write and up to two reads) per memory location. Figure 4.4(c) shows a possible program trace and the access history management using fast mode as described in Section 13. At time 3, when the read operation at line 8 occurs, the W-node corresponding to read access W_A is not added to the metadata entry. Following lines 33-38 of Algorithm 19, since W-node W_A may execute in parallel with both previously stored metadata entries (W_B, W_C), and $LCA(W_A, W_C)$ and $LCA(W_A, W_B)$ is not closer to the root node of the OSPG the metadata entry is left unchanged after time 3 in the program trace in Figure 4.4(c).

In precise mode, for lock-free programs, instead of storing a single 3-tuple of W-nodes, (Wr, R_1, R_2), the number of 3-tuples grow proportionally to the number of active ST-nodes in the program. As shown in Figure 4.4(c) for programs with ST-nodes in their OSPG, maintaining two read access W-node entries is not sufficient to find the first data race. The example from the program snippet in Figure 4.4(a) can be generalized as follows. Consider two parallel reads, (R_1, R_2) that occur in tasks that are at an outer nesting level of the program and a parallel read that occurs in a task at an inner nesting level, R_3. Maintaining the earlier invariant results in keeping (R_1, R_2) in the access history. Leading to potentially missing data races that involve R_3. Thus, in precise mode, OMP-RACER maintains two additional reads and one write for each active ST-node.

Figure 4.4(d) shows the program trace and the access history metadata in precise mode. The access history is shown as a list of 4-tuples, where the first entry in the tuple references the root S-node or an ST-node in the program's OSPG. For each memory location reference by the program, OMP-RACER maintains one write, and two read accesses for the entire program, corresponding to the program's OSPG root S-node. An additional write and two read accesses for every active ST-node in the program. For example, with the extra number of access history entries per memory location in precise mode, OMP-RACER does not miss the data race in Figure 4.4(a), since at time 3, OMP-RACER stores W-node W_A in the access history list as shown in Figure 4.4(d). Subsequently, at time 4, when the program performs the write access that corresponds to W-node W_D, OMP-RACER detects the program's data race since it finds that W-nodes W_D and W_A are involved in a data race.

In precise mode, the metadata is updated as follows. On a memory access operation that corresponds to W-node W_{cur}, traverse the OSPG towards the root S-node and identify all ST-nodes on the path to the root node. For every ST-node encountered and the root node, retrieve the current entry from the access history if present; otherwise, create a new entry. This entry contains three W-nodes, corresponding to the latest write and up to two parallel reads. The update rules for this 3-tuple is similar to OMP-RACER's fast mode. For example, if the retrieved entry already includes two read entries, if the LCA node of W_cur and one of the existing read entry W-nodes is closer to the root than the LCA of the existing pair, then W_cur replaces one of the previous read accesses. Otherwise, information about W_cur is already subsumed by the existing two read entries in the access history tuple, and the entry is not updated.

Detecting Data Race in the Presence of Locks and Critical Construct. We now describe how OMP-RACER's access history management in precise mode is adjusted to handle mutual exclusion locks[4] and the OpenMP critical construct. OpenMP critical construct (See Section 2.4.4 for more detail), acts similar to a lock in that it provides mutual exclusion to the block of code surrounded by a critical construct where only one worker thread at a time can execute. Locks and the OpenMP critical construct provide ordering between two memory accesses. Hence, two memory accesses to the same location by different threads in parallel does not constitute a data race if both accesses hold the same lock or execute within critical constructs with the same name. Locks and OpenMP critical constructs are identified by a unique identifier[5], which makes their handling by OMP-RACER' precise mode similar. For the rest of this section, when we make a statement about OMP-RACER's handling of locks, it similarly applies the OpenMP critical construct.

To detect data races in the presence of locks, inspired by prior data race detectors such as ALL-SETS [44], SPD3 [161], and PTRACER [194], OMP-RACER's data race detection algorithm keeps track of the set of locks held before a memory access operation and only reports data races when two memory accesses do not hold a common lock. Thus, OMP-RACER maintains an access history metadata entry for each distinct set of locks used by the program. The lockset size grows exponentially with the number of distinct locks in the program. The

[4] Also referred to as a mutex.

[5] In most implementations, this identifier is the address of the lock

exponential growth of the lockset size impacts the usability of OMP-RACER's precise mode's access history management strategy in programs that use many distinct locks. In practice, most OpenMP programs use locks sparingly. For example, in our evaluated benchmarks from Sections 4.4 and 3.9, the tested programs were either lock-free or use a single lock.

Further, in the presence of locks, we need to adjust the number of write accesses maintained per access history metadata data entry. Unlike lock-free memory accesses, where two parallel write accesses on a shared variable constitute a data race, two parallel shared memory location write operations holding a common lock do not constitute a data race because of the ordering provided by the common lock. Hence, instead of storing the latest write operation, each metadata entry maintains up to two parallel write accesses in precise mode. Thus, for a given lockset and memory location, OMP-RACER maintains up to four W-nodes, two parallel writes, (Wr_1, Wr_2), and two parallel reads, (R_1, R_2). The metadata entry update rule for parallel writes is similar to how parallel reads are handled in fast mode. If the W-node corresponding to the current write operation, W_cur is in parallel with the two previously stored write entries in the access history metadata, (Wr_1, Wr_2), it replaces one of them if LCA node of W_{cur} and $(Wr_1$ or $Wr_2)$ is closer to the root node than the LCA of the existing pair.

Overall, in the presence of locks and unrestricted use of taskwait, the metadata entry per memory location is comprised of 6 entries. The first two entries, `Lockset` and `Node` uniquely identify the lockset and the corresponding node (*i.e.*, root node or an ST-node) in the program's OSPG. The remaining four entries correspond to two writes and two read operations. On a memory access with lockset (l_c) with a corresponding W-node W_{cur}, the metadata is checked as follows. OMP-RACER iterates over the access history metadata entries to find the corresponding list of 6-tuples, $(Lockset, Node, Wr_1, Wr_2, R_1, R_2)$.

Data Race Detection Algorithm in Precise Mode

Algorithm 20 illustrates OMP-RACER's data race detection algorithm in precise mode. The algorithm comprises two parts. First, OMP-RACER checks for data races by checking if the current memory access is involved in a data race with previous memory accesses stored in the access history metadata (lines 4-14). Next, it updates corresponding access history metadata entries to include the current memory access operation (lines 15-45). The data race detection

algorithm in precise mode is similar to the algorithm used in fast mode, except each part is slightly more involved mainly due to the increased number of access history metadata entries.

The data race detection algorithm retrieves the access history list of metadata entries at line 2. Unlike fast mode, this operation returns a list of 6-tuple entries. Each entry is uniquely identified by its first two entries, corresponding to a lockset and the corresponding root node or ST-node in the program's OSPG. The remaining four entries correspond to two reads and two writes. To report data races, the OMP-RACER iterates over each access history list entry (lines 4-14). For each entry, if the intersection of the entry's lockset and current access' lockset is non-empty (line 5), the data race detection algorithm checks if the W-node corresponding to the current memory access, Cur, and the entries' 2 reads/writes are conflicting and may logically execute in parallel. If so, it reports an apparent race (lines 5-13).

Subsequently, OMP-RACER updates the access history metadata entry list as follows. Unlike fast mode, we modify the entries corresponding to the current lockset and all the ST-nodes on the path to the root node. Hence, starting at line 16, the data race detection algorithm traverses from the current W-node towards the OSPG's root node. For every encountered ST-node and the OSPG's root node, $Node$, OMP-RACER creates or retrieves a new entry from the list of access history entries that corresponds to the current lockset (lines 18-20). Next, the OMP-RACER retrieves the corresponding pair of W-nodes matching the currently encountered node, lockset, and corresponding type of access from the entry list. Each retrieved entry contains two W-nodes, a pair of read/write W-nodes, (N_1, N_2), matching the type of the current memory access (line 21). Finally, similar to fast mode, the OMP-RACER updates the pair of read W-nodes by replacing one of the existing entries with the current W-node, if needed[6] (lines 22-39 in Algorithm 20).

Time and Space Overhead. Similar to fast mode, the running time of OMP-RACER's data race detection is divided into three parts: First, the cost of constructing the program's OSPG. The asymptotic cost of OSPG construction is the same between fast and precise modes since updating st_val values add a constant number of operations per internal OSPG node. Second, tracking

---

[6] See Section 4.3.2 for more detail on the update rule.

Algorithm 20: OMP-RACER data race detection algorithm in precise mode.

```
1  function OnMemoryOperation (tid, addr, type, lockset)
2      EntryList ← GetAccessHistoryList(addr)
3      Cur ← GetCurrentWNode(tid)
4      foreach ⟨Lockset, Node, Wr1, Wr2, R1, R2⟩ ∈ EntryList do
5          if Lockset ∩ lockset = ∅ then
6              foreach W_hist ∈ ⟨Wr1, Wr2, R1, R2⟩ do
7                  if IsParallel(Cur, W_hist) = True then
8                      if type = Write ∨ Type(W_hist) = Write then
9                          ReportDataRace(Addr, type, Type(W_hist))
10                     end
11                 end
12             end
13         end
14     end
15     Node ← Parent(Cur)
16     while Node ≠ ∅ do
17         if Node.type = ST-node ∨ Node = GetRootNode() then
18             if ⟨⟨−, −, −, −⟩, Node, lockset⟩ ∉ EntryList then
19                 EntryList ← EntryList ∪ ⟨⟨−, −, −, −⟩, Node, Lockset⟩
20             end
21             ⟨N1, N2⟩ ← RetrieveEntry(EntryList, Node, lockset, type)
22             if N1 = ⟨−⟩ then
23                 N1 ← Cur
24             else if N2 = ⟨−⟩ then
25                 if IsParallel(Cur, N1) = True then
26                     N2 ← Cur
27                 else
28                     N1 ← Cur
29                 end
30             else if
                isParallel(Cur, N1) = False ∧ isParallel(Cur, N2) = False
                then
31                 N1 ← Cur
32                 N2 ← ⟨−⟩
33             else if isParallel(Cur, N1) = True ∧ isParallel(Cur, N2) = True
                then
34                 LCA12 ← LCA(N1, N2)
35                 LCAc1 ← LCA(Cur, N1)
36                 LCAc2 ← LCA(Cur, N2)
37                 if LCAc1 > LCA12 ∧ LCAc2 > LCA12 then
38                     N1 ← Cur
39                 end
40         end
41         Node ← Parent(Cur)
42     end
43     return
```

the set of locks held during program execution. Third, the cost of onMemoryOperation on each memory operation.

To track the locks held by the program execution trace, we need to maintain a per-thread set of locks held by the OpenMP program's worker threads. For a program with p worker threads and l unique locks, this results in an additional $O(pl)$ space overhead. The additional running time overhead is bounded by the number of lock operations in the program trace. For each lock operation, updating the corresponding per-thread lock set requires $O(1)$ operations. Thus for a program with n lock operations, OMP-RACER incurs an additional $O(n)$ running time overhead.

Similar to fast mode, the number of times the function onMemoryOperation is called is equal to the number of memory accesses in the program trace. However, the running time of onMemoryOperation is higher than fast mode. The increased running time can be attributed to the increased number of metadata entries per memory location. The loop that checks for data races in Algorithm 20 (lines 4-14), calls the isParallel function in proportion to the size of the metadata entry list. The metadata entry list size grows in proportion to the lockset size and the nesting level of the program's OSPG for a given memory location. Hence for a program with l unique locks and a height of h. The number of entries in the metadata list is bound by $O(lh)$. The loop that updates the metadata entry list (lines 16-42 in Algorithm 20), performs a traversal from a leaf node in the OSPG towards its root node, which is bound by the height of the OSPG. In the worst case, $O(h)$ ST-nodes may exist on the path towards the root node, which results in the branch at line 17 to be taken up to $O(h)$ times. The body of the branch includes a constant number of calls to the function isParallel. Overall, the function onMemoryOperation makes $O(lh)$ calls to isParallel.

For an OpenMP program with an OSPG of height h and up to k sibling tasks, the running time of isParallel is $O(h)$ in the absence of task dependencies, and $O(h + k)$ in the presence of task dependency. In precise mode, the asymptotic running time of isParallel is similar to fast mode since the Algorithm refalg.dmhp, which is used in precise mode, only performs a constant number of operations more than Algorithm 18, used in fast mode. Hence, in the absence of task dependencies, the running time of the function onMemoryOperation in precise mode is $O(lh^2)$, which is higher than its fast mode counterpart, which has a running time

of $O(h)$. Intuitively, in precise mode, OMP-RACER's data race detection algorithm running time increases by the same factor as the increase per metadata size per memory location.

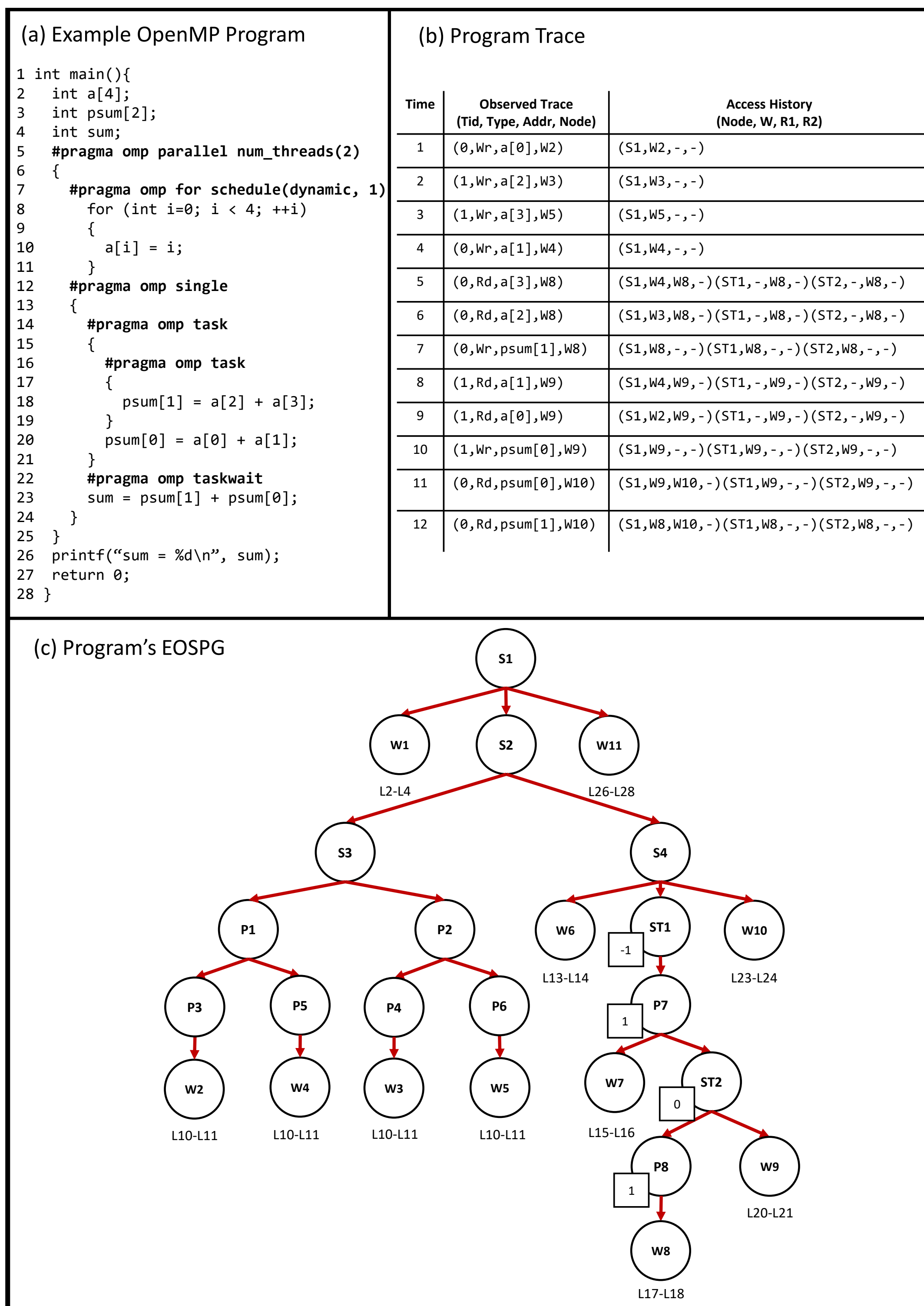
The space overhead of the data race detection algorithm is bound by the access history metadata's size. In precise mode, each access history list grows in proportion to the lockset's size and the number of active ST-nodes in the program bounded by $O(lh)$. Hence, unlike fast mode where the algorithm's space overhead is proportional to the program's working set size, in the worst case, OMP-RACER's precise mode requires a significantly larger amount of memory.

Discussion For a given program input, OMP-RACER data race detection algorithm is Complete; that is, for programs that use OpenMP constructs supported by OMP-RACER, it does not report false positives. OMP-RACER is also sound for programs with supported constructs. For unsupported constructs, OMP-RACER may miss data races. For example, as a dynamic data race detector, OMP-RACER cannot detect data races caused by instruction-level parallelism, namely OpenMP's SIMD construct, without auxiliary information passed by the compiler. Additionally, OMP-RACER does not support OpenMP's offloading construct used for offload computations to GPUs and other accelerators. To support offloading, we need the ability to track memory accesses in the offloaded device, which we plan to explore in future work.

It should be noted that this soundness and completeness guarantee applies only until the first data race in the program is observed. After the first data race is observed, the assumptions made by the data race detection algorithm no longer hold. In practice, if the developer's goal is to eliminate all data races in an application with n data races, they can identify all n data races by running the program n times and each time eliminating the first reported data race.

4.3.3 Illustrative Example

We will now show OMP-RACER's data race detection algorithm when used to detect the data race in the example program in Figure 4.5(a) that uses different OpenMP constructs to compute the sum of an array in parallel. It uses worksharing and tasking constructs to add parallelism to the program. This example has a data race due to insufficient synchronization between child tasks and the implicit task executing the single construct (lines 12-24 in Figure 4.5(a)). To detect this data race, we use OMP-RACER's precise mode.



Time	Observed Trace (Tid, Type, Addr, Node)	Access History (Node, W, R1, R2)
1	(0,Wr,a[0],W2)	(S1,W2,-,-)
2	(1,Wr,a[2],W3)	(S1,W3,-,-)
3	(1,Wr,a[3],W5)	(S1,W5,-,-)
4	(0,Wr,a[1],W4)	(S1,W4,-,-)
5	(0,Rd,a[3],W8)	(S1,W4,W8,-)(ST1,-,W8,-)(ST2,-,W8,-)
6	(0,Rd,a[2],W8)	(S1,W3,W8,-)(ST1,-,W8,-)(ST2,-,W8,-)
7	(0,Wr,psum[1],W8)	(S1,W8,-,-)(ST1,W8,-,-)(ST2,W8,-,-)
8	(1,Rd,a[1],W9)	(S1,W4,W9,-)(ST1,-,W9,-)(ST2,-,W9,-)
9	(1,Rd,a[0],W9)	(S1,W2,W9,-)(ST1,-,W9,-)(ST2,-,W9,-)
10	(1,Wr,psum[0],W9)	(S1,W9,-,-)(ST1,W9,-,-)(ST2,W9,-,-)
11	(0,Rd,psum[0],W10)	(S1,W9,W10,-)(ST1,W9,-,-)(ST2,W9,-,-)
12	(0,Rd,psum[1],W10)	(S1,W8,W10,-)(ST1,W8,-,-)(ST2,W8,-,-)

Figure 4.5: (a) Example OpenMP program with a write-read apparent data race on variable psum[1] at lines 18 and 23. (b) The execution trace of the example program when executed with two threads. The first column specifies the ordering of the observed trace. The second column specifies the memory access as a 4-tuple comprised of thread id, memory access type, memory access location, and the W-node performing the access. The third column illustrates the access history maintained by OMP-RACER for the corresponding memory access as a 4-tuple with the node identifier, W-node corresponding to the latest write, and two W-nodes corresponding to two parallel read accesses (- denotes the empty set). (c) The program's OSPG. The code fragment each W-node represents is shown below it. The square boxes next to some OSPG nodes represent the value indicating the nesting depth in the presence of taskwaits.

Figure 4.5(b) provides the dynamic execution trace and the updates to the access history metadata for each memory location using the precise mode for the example program in Figure 4.5(a). The write operation to `psum[1]` at line 18 of the example program corresponds to the W-node, $W8$. Upon this write operation, the metadata associated with `psum[1]` is updated to include $W8$. The path from the root of the OSPG to $W8$ has two ST-nodes: $ST1$ and $ST2$. In addition, the metadata corresponding to `psum[1]` is also updated to include $W8$ for each ST-node on the path to the root node, resulting in three entries for `psum[1]` in the access history (as illustrated at time 7 in Figure 4.5(b)). Eventually, the program execution reaches the taskwait construct at line 22 in Figure 4.5(a). OMP-RACER updates the st_val of node $ST1$ from 0 to -1 to record the presence of a taskwait. Later, when the implicit task executing the single construct performs a read operation on `psum[1]` at line 23 in Figure 4.5(a), the W-node representing the current read is $W10$. OMP-RACER retrieves the metadata for `psum[1]`, which includes access histories corresponding to the root node of the OSPG and ST-nodes $ST1$ and $ST2$. Since the path from $W10$ to the root node of the OSPG does not contain any ST-nodes, only the metadata entry corresponding to the root node of the OSPG, $S1$, is updated to include the current read access (time 12 in Figure 4.5(b)). Next, OMP-RACER looks for possible data races by checking if the current read operation happens in parallel with any previous write accesses recorded in the access history metadata. In our example, OMP-RACER checks if the current read access may happen in parallel with the previously recorded write operation corresponding to W-node $W8$. In this case, the LCA of $W8$ and $W10$ is $S4$. The left child of $S4$ on the path to $W8$ is an ST-node $ST1$. Next, OMP-RACER computes the sum of the st_val values from $W8$ to $ST1$, which evaluates to 1. Thus, $W8$ and $W10$ may execute in parallel, and one of the operations is a write operation, which results in OMP-RACER reporting an apparent race on the memory access to `psum[1]`.

4.4 Evaluation

In this section, we discuss the evaluation with our OMP-RACER prototype. We start by describing our prototype's implementation details and some correctness and performance concerns that OMP-RACER prototype must address to build an effective and usable tool. Next, we list the

benchmarks and the experimental environment used to evaluate OMP-RACER. Subsequently, we present our evaluation results. Our evaluation results primarily answer two research questions: (1) Can OMP-RACER correctly identify apparent data races in different OpenMP applications? (2) What are the performance overheads of OMP-RACER, and how does it compare to other state-of-the-art OpenMP data race detectors?

4.4.1 Prototype Implementation

Prototype. Our OMP-RACER prototype supports C/C++ OpenMP programs. It uses LLVM-10's [8] OpenMP runtime and its implementation of the OpenMP tooling interface, OMPT. The OMPT interface supports registering callbacks whenever the runtime encounters different execution events. OMP-RACER's prototype uses OMPT to receive callbacks whenever the runtime encounters different OpenMP constructs and uses them to construct the program's OSPG on-the-fly during program execution. It also includes an LLVM pass to instrument memory accesses. OMP-RACER has two modes: a precise and a fast mode. The precise mode detects data races even when the program uses locks and imposes no restriction on how taskwaits are used in the program, which can have significant overheads compared to fast mode. To safely use fast mode, OMP-RACER first checks if the program has fully nested taskwaits. If so, it uses a constant amount of metadata per memory location in the subsequent execution for race detection. It is significantly faster. OMP-RACER's prototype is publicly available [32].

OMPT callbacks. The OpenMP tooling interface is a portable solution to build performance tools for OpenMP program that has been adopted and improved by the recent OpenMP specifications [152, 153]. OMPT supports registering different callbacks that notify the tool whenever different events such as entering/exiting an OpenMP construct. Our OMP-RACER prototype uses the callbacks listed in Table 4.1. The table lists the callback's name and provides a short description of the callback on when it gets dispatched to the tool.

While all these callbacks are specified in the latest OpenMP 5.0 specification [153], implementing some callbacks is optional as per the standard. OMP-RACER uses LLVM-10's OpenMP runtime implementation. The OMPT runtime implementation does not implement the `ompt_callback_dispatch` callback. OMP-RACER needs this callback to correctly model the parallel relation between dynamically scheduled loop chunks and the section regions

Table 4.1: OMPT callbacks used by OMP-RACER prototype

Callback name	Description
ompt_callback_thread_begin	Called when a new low-level thread is created
ompt_callback_thread_end	Called when a low-level thread ends
ompt_callback_parallel_begin	Called when a parallel region starts execution
ompt_callback_parallel_end	Called when a parallel region ends
ompt_callback_task_create	Called when a task is created
ompt_callback_task_schedule	Called when task scheduling decision are made
ompt_callback_implicit_task	Called when initial tasks and implicit tasks are created
ompt_callback_dependences	Called to announce the dependencies of a task
ompt_callback_work	Called when a worksharing region begins-ends
ompt_callback_master	Called when a master region begins-ends
ompt_callback_sync_region	Called when barriers, takswaits, and taskgroups are encountered
ompt_callback_mutex_acquire	Called when locks are acquired or critical regions are begun
ompt_callback_mutex_released	Called when locks are released or ciritcal regions are exited
ompt_callback_dispatch*	Called when a section or loop iteration begin execution

of a section construct. Hence, our prototype uses a slightly modified version of the OpenMP runtime with support for this callback.

OSPG Data Structure Organization. OMP-RACER constructs the program's OSPG during program execution. Hence we need to maintain the OSPG data structure in memory. Since the OSPG is a DAG, using graph representation such as an adjacency list is an option. However, for OMP-RACER's prototype, we store the OSPG by laying it on an array. Compared to adjacency lists, this approach requires less pointer chasing, and we can limit calls to the dynamic memory allocator when new OSPG nodes are created. For example, since OMP-RACER works on Linux, we use mmap to reserve the space for storing OSPG array. Each element in the array stores an OSPG node. The node's index in the array is used as the OSPG node's identifier. In the default setting, OMP-RACER allocates an array with 2^{24} elements. Hence, to store an OSPG node identifier requires 24-bits.

The OSPG node identifier is used to store OSPG edges. We maintain the parent-child edges in the OSPG data structure by storing the parent node's index in the child node element. Dependency edges are maintained by storing a node's dependent nodes indices in each array element. Since the maximum number of dependency edges for a node is highly program-dependent and most programs only use a few dependencies per node, by default, we store up to four dependency edges in each OSPG node array element. Additional dependency edges

are stored outside the array element by using a linked list. This approach provides better performance for programs with a low number of dependencies per task while enabling our data structure to support storing programs with an arbitrarily large number of dependencies per task. Recall that the OSPG construction algorithm uses a per-thread stack that stores OSPG nodes (see Section 3.3.1). Since the actual nodes are stored in the OSPG array, OMP-RACER stores the node identifier on the per-thread stack.

When the program starts, the OSPG array is empty, and all elements are available. As the OSPG is incrementally constructed, we start populating the OSPG one element at a time by using a tail variable that contains the index of the first available OSPG element. OMP-RACER maintains the invariant that the tail index only grows. Since OSPG construction is in parallel, this invariant makes it straightforward to allocate new OSPG nodes from the array. We use an atomic type for the tail variable. On every new OSPG node request, we atomically increment the tail variable's value to point to the next available OSPG node. Maintaining the tail invariant raises the question of how does OMP-RACER scale to analyze long-running programs that have an OSPG with more than 2^{24} nodes?

OMP-RACER addresses this problem by leveraging the following simple idea: Once an OpenMP program exits a parallel and the program's execution is returned to the serial threads, we deallocate all the OSPG array elements, except the first entry that represents the root node, by moving the tail index back to the beginning of the OSPG array. Removing all these nodes from the OSPG does not impact OMP-RACER data race detection ability since because of the implicit barrier at the end of the parallel region, no data race can occur between a memory operation within the parallel region and its continuation. Hence, it is safe to discard all the OSPG nodes of the previous parallel region. By deallocating OSPG nodes after a parallel region, OMP-RACER can be used with long-running applications. With this approach, the OSPG array's length limits the maximum number of OSPG nodes within a program's parallel region. If within a parallel region, a program's OSPG contains more nodes than the number of elements in the OSPG array, then OMP-RACER will run out of memory to store nodes and exits the program. However, in the benchmarks we tested, 2^{24} was sufficient to store the OSPG subtree of each parallel region. Hence, in practice, this restriction does not impact the usability of the

prototype for real-world applications. If required, OMP-RACER provides an option to use larger OSPG arrays.

Access History Metadata Organization. OMP-RACER's access history metadata tracks memory locations' accesses during the program execution. Recall that for a memory access operation, OMP-RACER stores the W-node corresponding to the memory access. In our prototype, OMP-RACER stores the OSPG node identifier corresponding to this W-node in the metadata entry. OMP-RACER's access history metadata is stored in a shadow memory. The shadow memory is designed as a map data structure that maps each memory referenced memory location to the corresponding access history metadata entry. We implement this map with a two-level trie [139]. Our prototype supports x86-64 systems. Hence, we map the 48-bit virtual address space with a two-level trie. The trie acts like a software implementation of a two-level page table. The first level of the trie is indexed by the first 10-bits of the memory address and is allocated using mmap during OMP-RACER's initialization. The second level is addressed by the next 24-bits of the memory address and is allocated based on demand during program execution.

In fast mode, each metadata entry is comprised of three W-nodes, one write, and two reads. Hence, the metadata entry size in fast mode is three times the size of W-node identifiers. By default, in the OMP-RACER prototype, W-node identifiers are 24-bit values, resulting in 9 bytes of required storage per metadata entry in fast mode. Updates to each metadata entry must be performed atomically. One approach is to use a lock, which depending on the lock used, adds at least 4 bytes to the size of each metadata entry[7]. Thus, instead of using locks fast mode, we use the 128-bit atomic compare-and-swap (CAS) instruction to update metadata entries. This change requires using a 128-bit alignment for the metadata entries. In practice, the CAS instruction provides lower overheads than using common lock objects.

In contrast, precise mode's metadata entry size is no longer constant and comprises a list of entries, where each entry comprises one OSPG internal node (the root S-node or an ST-node), a lockset, and four W-nodes (two reads and two writes). This requires storing five OSPG node identifiers and a lockset for each entry in the access history list. By default, OMP-RACER

[7]For example, `pthread_spinlock_t` and `pthread_mutex_t` respectively use 4 and 40 bytes on x86-64 machines.

uses 128-bits to store an element of the entry list when using precise mode. We use 120 bits to store the OSPG node identifiers and use the remaining 8 bits to encode locksets of up to seven different locks[8]. To represent the metadata entry list, OMP-RACER stores the first seven elements (112 bytes) of the list directly in the metadata object while using an indirect pointer (8 bytes) to store additional elements. We also need to use a lock object to ensure that metadata entry updates are performed atomically. Overall, in precise mode, OMP-RACER uses 128 bytes for an access history metadata entry.

By default, OMP-RACER keeps an access history metadata entry at a 4-byte memory address granularity. Using a 4-byte granularity implies that OMP-RACER may report false data races when the program performs memory accesses at a 1-byte granularity, *e.g.*, Initializing a character array in a parallel loop. The default option of 4-byte granularity is chosen since it offers a good trade-off between usability and lower memory overheads. Further, to prevent running out of memory for long-running applications, similar to the program's OSPG, OMP-RACER has an option to clear the access history metadata at the end of a parallel region.

Caching ISPARALLEL Queries. Computing the ISPARALLEL relation requires performing an LCA query in the program's OSPG (See Section 4.3). Performing an LCA query on a OSPG with a large height can be costly, *e.g.*, task-parallel programs with a large nesting level. OMP-RACER provides an option to cache the recently used OSPG LCA queries. By default, this option is disabled in our prototype since, as our evaluation results indicate, the cache did not have a high enough hit-rate to amortize the cost of managing it.

Avoiding False Data Races Caused by Task-local Memory. An OpenMP program may create many logically parallel sibling tasks, *e.g.*, in a divide-and-conquer algorithm with a large nesting level. The OpenMP runtime executes these tasks by scheduling them for execution on a smaller number of worker threads. Once a task completes execution, the OpenMP runtime schedules the next task. Most OpenMP runtime implementations optimize this step by recycling and reusing the completed tasks object, including task-local heap and stack regions. Since these memory regions are task-local, they do not constitute a data race between two logically

[8]We use 1-bit to encode when the memory operation holds no locks.

Table 4.2: List of applications used for evaluating OMP-RACER's performance overheads

Benchmark	Application	Description
Coral	amgmk	Algebraic multigrid solver microkernel
PBBS	Convex Hull (CH)	Compute convex hull of a set of points in 2D plane
PBBS	Delaunay Triangulation (DT)	Compute Delaunay triangulation of a set of points in 2D plane
PBBS	Minimum Spanning Forest (MSF)	Compute minimum spanning forest of for input weighted undirected graph
PBBS	Nbody Forces (NBODY)	Solve n-body force calculation problem for a set of points in 3D space
Coral2	Quicksilver	Monte Carlo transport benchmark
BOTS	Alignment	Align a sequences of proteins
BOTS	FFT	Compute a Fast Fourier Transformation
BOTS	Floorplan	Compute the optimal placement of cells in a floorplan
BOTS	Health	Simulates a country health system
BOTS	Sort	Use a mixture of sorting algorithms to sort a vector
BOTS	NQueens	Finds solutions of the N Queens problem
BOTS	Sparselu	Compute the LU factorization of a sparse matrix
BOTS	Strassen	Compute matrix multiplication with Strassen's method
PBBS	BFS	Run a breadth first search on a directed graph
PBBS	CompSort	Sort a sequence given a comparison function
PBBS	Delaunay Refinement (DR)	Refine a Delaunanay triangulation given in input angle
PBBS	Dictionary	Perform dictionary operations
PBBS	Integer Sort (ISORT)	Sort fixed-length unsigned integer keys
PBBS	Maximal Independent Set (MIS)	Compute maximal independent set of the input graph
PBBS	Maximal Matching (MM)	Compute maximal matching of the input graph
PBBS	K-Nearest Neighbors (KNN)	Find k nearest neighbors of a set of points
PBBS	Ray Cast	Find the intersection of input rays and triangles
PBBS	Remove Duplicates	Remove duplicate items from a sequence of input objects
PBBS	Spanning Forest (SF)	Compute a valid spanning forest for input undirected graph
PBBS	Suffix Arrays (SA)	Generate suffix array of input string

parallel tasks. However, since both tasks are using the same memory ranges, a dynamic data race detector, unaware of this issue, may falsely report data races on these reused memory locations.

To address this issue, when a task is created, OMP-RACER tracks the address ranges used by a task's stack and heap. Further, it updates the stack address range as the stack grows during the task's lifetime. once the task finishes execution, OMP-RACER clears the access history metadata corresponding to the stack and heap address ranges to avoid reporting false races on subsequent accesses to the same task-local memory ranges. Handling this issue is not unique to OMP-RACER. Recent versions of other state-of-the-art dynamic data race detectors use different techniques to avoid reporting such false data races [159].

4.4.2 Evaluation Setup

Benchmarks. We evaluate OMP-RACER with two sets of benchmarks. To check the effectiveness of OMP-RACER at detecting data races, we use DataRaceBench 1.2.0 [116]. DataRaceBench is a collection of 116 OpenMP microbenchmarks designed to evaluate OpenMP data race detection tools. Each microbenchmark is related to a specific use of an OpenMP

Table 4.3: Data race detection report on DataRaceBench 1.2.0

Tool	Accuracy	Recall	Precision
OMP-RACER	1.00	1.00	1.00
Archer	0.95	0.91	1.00
Coderrect	0.91	0.93	0.89

construct and contains zero or one data races. Since we know the ground truth for each DataRaceBench microbenchmark, it enables us to check the accuracy, precision, and recall of OMP-RACER and compare its results to other OpenMP tools.

To measure OMP-RACER's performance overheads, we use a suite of 26 OpenMP applications from Coral, BOTS, and PBBS benchmarks suites are shown in Table 4.2. These applications are taken from OpenMP benchmarks. We use these benchmarks to show the usability of OMP-RACER in real-world OpenMP applications. Unlike the microbenchmarks in DataRaceBench, these application's data races are not labeled with their data races' location, if any. To the best of our knowledge and using manual inspection of the applications based on the data races reported by other OpenMP data race detectors indicate that these applications do not have a data race. However, these are large programs, and understanding their intricacy requires domain-knowledge. Hence, unlike DataRaceBench where we trust the ground truth, we preface that the absence of data races in the applications in Table 4.2 is a product of a best-effort manual inspection and does not prove the absence of data races.

Test Setup. We performed all experiments in an Ubuntu 16.04 machine with a 16-core Xeon 6130 processor running at 2.1GHz and with 32GB of memory. All tests were performed with the LLVM-10 release branch with the included OpenMP runtime implementation. When testing OMP-RACER, we used a modified version of LLVM-10's OpenMP runtime to add the missing OMPT callbacks listed 4.4.1 to the OpenMP runtime. When comparing to Archer, we used the tool that was included in the LLVM-10's release[9]. When using Coderrect static data race detector [7, 181], we used version 0.7.0.

4.4.3 OMP-RACER Evaluation Results

Data Race Detection Effectiveness. DataRaceBench 1.2.0 is comprised of 116 microbenchmarks, where each microbenchmark is designed to check a tool's data race detection ability in a specific use of an OpenMP construct. OMP-RACER prototype does not support detecting instruction-level SIMD data races, and it does not support instrumenting memory accesses on accelerators. Hence it does not detect data races on programs that use the offload construct to run code on accelerators. Overall, out of the 116 microbenchmarks, OMP-RACER supports 106 of them that do not contain these constructs. Each microbenchmark in DataRaceBench is labeled with yes or no depending on if the microbenchmark has a data race. A data race detection tool that correctly identifies the existence or lack of a data race in a given microbenchmark is considered a true positive (TP) or true negative (TN), respectively. In contrast, a tool may misclassify a microbenchmark with a false positive (FP) or a false negative (FN). Hence, we can measure the accuracy[10], precision[11], and recall[12] of a data race detector based on the results it reports over the set of DataRaceBench's microbenchmarks.

The results in Table 4.3 show the effectiveness of OMP-RACER and other state-of-the-art data race detectors when run on these set of microbenchmarks. We ran Coderrect up to five times. However, as a static data race detector, its results do not change across different runs. For OMP-RACER, we run each application five times with 16 worker threads using precise mode. Since OMP-RACER is a per-input data race detector, we do not need to run it with a different number of threads. For Archer, we follow DataRaceBench test harness defaults and run each application five times with a different number of threads ranging from 3 to 256[13].

As shown in Table 4.3, since OMP-RACER correctly identifies data races across all tested microbenchmarks, it outperforms Archer and Coderrect over all three metrics. This version of Coderrect does not support several task-based constructs such as taskwait and task dependencies. Hence it fails to identify data races in microbenchmarks that use these OpenMP constructs.

---

[9] Available at https://github.com/llvm/llvm-project/tree/release/10.x/openmp/tools/archer

[10] Accuracy is defined as TP + TN / (TP + FP + TN + FN)

[11] Precision is defined as TP / (TP + FP)

[12] Recall is defined as TP / (TP + FN)

[13] DataRaceBench test harness uses $3, 35, 45, 72, 90, 180, 256$ threads to evaluate a data race detector.

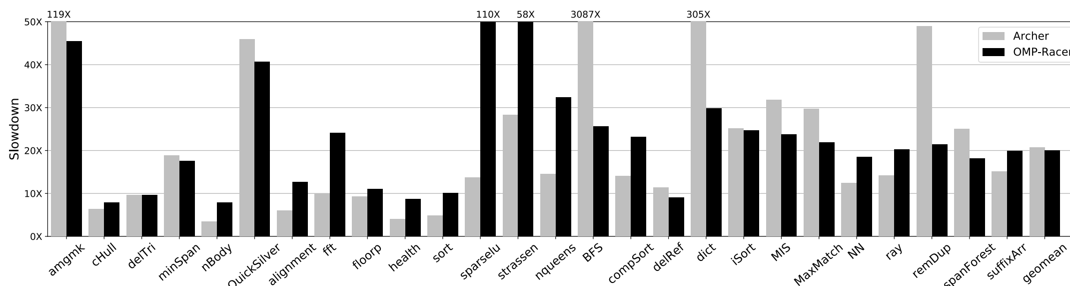


Figure 4.6: Performance slowdown over parallel execution of OMP-RACER and Archer with our performance overhead benchmarks.

When comparing OMP-RACER to Archer, the state-of-the-art dynamic data race detector for OpenMP, both tools do not detect any false positives and have a precision of 1.00. However, Archer reports false negatives on five microbenchmarks. The reason for these false positives is caused by Archer being a per-schedule data race detector. For example, microbenchmark DRB008-indirectaccess4-orig-yes.c has a data race that only manifests if the program is run with more than 180 threads on a per-schedule data race detector. Hence, Archer misses this data race on approximately 80% of our runs. In contrast, since OMP-RACER is a per-input data race detector and captures the program's logical series-parallel relations, it can detect the data races in these microbenchmarks with only 16 threads and in a single execution.

Overall, our results indicate that OMP-RACER effectively detects data races in a set of OpenMP applications using that use a variety of OpenMP constructs. Further, in DataRaceBench 1.2.0, OMP-RACER slightly outperforms both state-of-the-art dynamic and static OpenMP data race detectors.

Performance overheads in Fast Mode. Figure 4.6 reports the performance overhead of OMP-RACER in fast mode and Archer with our performance benchmarks. Each bar indicates the slowdown incurred by the tool over the parallel execution of the application. The runtime overhead of OMP-RACER in its fast mode, on average[14], is 20.04×. The overhead of Archer is 21.67×. This result indicates that the OMP-RACER's fast mode performance overhead is comparable to Archer, a state-of-the-art OpenMP data race detector.

Compared to Archer, OMP-RACER has higher runtime overhead in BOTS Strassen, Sparselu, FFT, and NQueens. All four programs use OpenMP tasking to solve a problem

[14] Using geometric mean

using recursive decomposition. The nesting level of these programs is higher than other benchmarks in this set. Since these programs perform a significant amount of recursive decomposition, OMP-RACER has higher runtime overhead compared to Archer and other applications tested with OMP-RACER. The height of the OSPG is proportional to the nesting level of the program. An increase in the height of the OSPG increases the cost of performing LCA queries used to compute ISPARALLLEL queries, resulting in higher overheads compared to other benchmarks we tested.

We considered a caching solution for LCA queries to remedy this issue. When tested, Caching LCA results lead to slight overhead reductions in Strassen and Sparselu. However, overall, it increased OMP-RACER's average runtime overhead across our performance benchmarks. Further investigations indicate the ineffectiveness of using a cache is caused by a lack of a sufficient per-thread locality, which results in a low cache hit ratio, which is not sufficient to amortize the cost of managing a cache. Thus by default, OMP-RACER does not cache LCA queries. However, the option is available in our prototype, and it is useful in task-based applications with a large nesting level. Overall, while comparable to other tools, we would still like to investigate solutions that can reduce the performance overheads of OMP-RACER in fast mode to improve its usability as a debugging tool.

Performance overheads in Precise Mode. Table 4.4 compares the performance overhead of using OMP-RACER in fast mode and precise mode over the set of our OpenMP benchmarks. All values reported in the table denote the performance slowdown compared to the program's parallel execution on our test setup's 16-core machine (See Section 4.4.2).

The data in Table 4.4 show that compared to fast mode, OMP-RACER's precise mode has a significantly higher slowdown. By comparison, the cost of performing the check to determine if it is safe to use fast mode over precise mode, shown in column **Check**, is negligible. Thus, using this check before running OMP-RACER is justified since it can offer significant time-savings if an application is determined to be safe to use with OMP-RACER's fast mode.

The main reason for this increased slowdown in precise mode is attributed to the large increase in the amount of metadata maintained per memory location. Since the amount of metadata entries per memory location grows with the number of active ST-nodes in the program, task-based OpenMP programs in the BOTS benchmark suite incur a larger increase in slowdown

Table 4.4: Performance overhead comparison between OMP-RACER's fast mode and precise mode

Application	**Check**	**Fast**	**Precise**
amgmk	1.15×	45.42×	156.58×
Convex Hull (CH)	1.01×	7.86×	48.80×
Delaunay Triangulation (DT)	1.03×	9.62×	26.54×
Minimum Spanning Forest (MSF)	1.02×	17.52×	100.52×
Nbody Forces (NBODY)	1.00×	7.82×	10.42×
Quicksilver	1.07×	40.69×	196.32×
Alignment	1.10×	12.67×	194.43×
FFT	5.13×	24.04×	OOM
Floorplan	1.07×	11.07×	OOM
Health	1.32×	8.75×	OOM
Sort	1.47×	10.12×	537.00×
NQueens	1.02×	110.93×	205.17×
Sparselu	1.06×	58.22×	823.00×
Strassen	1.03×	32.38×	984.10×
BFS	1.02×	25.56×	180.67×
CompSort	1.01×	23.18×	84.22×
Delaunay Refinement (DR)	1.01×	9.01×	34.71×
Dictionary	1.07×	29.78×	59.22×
Integer Sort (ISORT)	1.00×	24.64×	245.61×
Maximal Independent Set (MIS)	1.04×	23.74×	196.00×
Maximal Matching (MM)	1.01×	21.93×	207.32×
K-Nearest Neighbors (KNN)	1.02×	18.51×	58.75×
Ray Cast	1.01×	20.28×	OOM
Remove Duplicates	1.02×	21.42×	172.29×
Spanning Forest (SF)	1.01×	18.12×	138.55×
Suffix Arrays (SA)	1.01×	19.91×	40.06×
Geo Mean	1.12×	20.04×	124.62×

compared to other tested applications (*e.g.*, Strassen and Sparselu). Precise mode's increased size in the number of metadata entries per memory location results in more time spent reading and updating the access history metadata per memory access, contributing directly to increased overhead. Further, the increased size of the access history metadata necessitates the use of locks instead of atomic CAS operations when updating the access history metadata. This change also contributes to the increased runtime overheads compared to fast mode. The increased size of the access history metadata in precise mode also causes OMP-RACER to run into memory issues when testing four applications from our benchmarks suite[15].

Hence, due to its significant runtime and memory overheads, OMP-RACER's precise mode, in its current state, is not usable with large or long-running OpenMP applications. However, we found OMP-RACER's precise mode useful for testing subsets of OpenMP programs with parallel regions that use task and task synchronization constructs in an intricate manner. Currently, the user can manually annotate such regions for data race detection. We intend to explore ideas that use static analysis to identify such regions instead of manual inspection.

4.5 Summary

Detecting data races in OpenMP applications with high accuracy and low overheads is challenging. In this chapter, we proposed OMP-RACER, our data race detector that uses the OSPG to encode the logical series-parallel relations of the OpenMP program trace. We showed that OMP-RACER effectively identifies data races, and when using its fast mode, it has comparable overhead with state-of-the-art OpenMP dynamic data race detectors. Further, a key novelty of OMP-RACER is in designing an apparent data race detector with support for both worksharing and tasking construct. By detecting apparent data races for a given input, OMP-RACER can identify data races that manifest in other thread interleavings. Thus alleviating the need for repeated execution of the program under test and exploring different thread interleavings. OMP-RACER relies on the OSPG to precisely encode the series-parallel relations in an OpenMP application, highlighting the usefulness of the OSPG data structure in designing debugging tools for OpenMP applications. While OMP-RACER in fast mode has comparable runtime

[15] indicated with OOM (out of memory) in Table 4.4

overheads to other data race detectors, its runtime overheads in precise mode leave room for improvement. We plan to explore ideas to reduce the overhead of precise mode and investigate hybrid data race detection with static analysis.

Chapter 5

Related Work

Correctness and performance are the main concerns of application developers. Throughout the years, many techniques to analyze the correctness and performance of serial and parallel applications have been proposed. The work presented in this dissertation aims to tackle performance issues for OpenMP applications by proposing a parallelism centric performance analysis tool OMP-ADVISER and take on the correctness issues by exploring apparent data race detection in OpenMP application, proposing OMP-RACER. The key enabler in these two solutions is the OSPG, a series-parallel graph for OpenMP applications that precisely encodes the logical-series parallel relations between fragments of an OpenMP program's dynamic trace.

5.1 Series-Parallel Graphs

Capturing the execution of a parallel application is required to analyze its correctness and performance. One approach is to use graphs to capture the execution of the application. The graph is designed based on the type of analysis, information captured, and the underlying parallelism abstractions.

Some works capture time and space requirements of a multithreaded applications as a computation graph [28, 190]. Such a computation graph aims to capture a given execution trace as a DAG of instructions. In a computation graph, a node represents an instruction in the program trace. The computation graph is comprised of three types of edges: (1) continue, (2) spawn, and (3) dependency. Continue edges represent the sequential ordering of instructions within a thread. A spawn edge is used to denote the creation of a new thread by its parent thread. Lastly, dependency edges are used to model the data and control dependency between different threads, such as synchronization primitives, including locks. Similar to an OSPG, a computation graph can be used to capture the work and the critical work of an application. In a

computation graph, the application's work can be computed as the total cost of the instructions in the instruction nodes in the graph. The critical work is defined as the total cost of instruction on an execution schedule on a system with a hypothetically infinite number of processors. Inspired by computation graphs, the program activity graph (PAG) has been used to capture the execution trace and critical path of shared memory and message passing parallel programs [91, 172]. Compared to the OSPG, the PAG does not provide a way to measure parallelism at finer granularities.

With the increasing popularity of high-level parallel programming languages such as OpenMP [153], Cilk [47, 70], TBB [163], X10 [41], and OpenACC [151], several lines of work propose data structures aimed at capturing the execution of programs written in these parallel programming languages. Compared to thread-based programming, these languages introduce higher-level abstractions and parallelism primitives that either forbid or discourage ad hoc use of threads and synchronization primitives with no structure. Hence, there has been different lines of research work aimed at encoding the series-parallel relations in more structured parallel programming models [27, 58, 68, 132, 148, 160, 162].

For the fork-join programming model, where each fork operation has a corresponding join, the program's computation graph [28, 190] can be encoded by the series-parallel (SP) parse tree [68]. Similar to the OSPG, each fragment of execution has a corresponding leaf node in the (SP) parse tree. The (SP) parse tree captures the logical series-parallel relations between program fragments using two types of internal nodes, S-nodes and P-nodes. The (SP) parse tree constructions algorithm requires a serial, depth-first execution of the Cilk program. In contrast, the OSPG can be constructed in parallel and encodes the logical series-parallel relations in OpenMP programs that are not limited to the fork-join programming model.

Instead of constructing a series-parallel graph, the English-Hebrew labeling [148] proposes a labeling technique to encode the logical series-parallel relations in fork-join programs. When using the English-Hebrew labeling, each program fragment is assigned two labels: English and Hebrew. Each type of label is assigned to a program fragment by using a different traversal of the program's (SP) tree. By comparing the labels of a pair of program fragments, it is possible to determine their logical series-parallel relation. Offset-span labeling [132] is another labeling technique for encoding the logical series-parallel relations in fork-join programs using

numeric labels. Offset-span label sizes grow in proportion to the program's nesting level. For English-Hebrew labeling, each label size grows in proportion to the number of logical threads in the program. When using offset-span labeling, determining the series-parallel relations for a pair of program fragments is proportional to the program's nesting level, which improves upon English-Hebrew labeling.

ROMP expands the offset-span labeling technique to support OpenMP constructs [81]. Similar to the OSPG, ROMP's offset-span labeling captures the logical series-parallel relations between different fragments of an OpenMP application. Asymptotically, for basic worksharing and tasking constructs, the cost of operations in the OSPG are comparable to ROMP's offset-span labeling since the length of labels and the depth of the OSPG grow proportional to the nesting level of the program. However, it is not clearly described how ROMP models some OpenMP constructs such as taskwait.

The OSPG is inspired by the Dynamic Program Structure Tree (DPST) [160, 162]. The DPST encodes the logical series-parallel relations in task-parallel programs. It was initially proposed for data race detection in explicit async-finish programs written in Habanero Java (HJ) [38] and X10 [41]. Later the DPST was adapted to support task-parallel programs in TBB for data race detection [194] and performance analysis [192, 193]. The OSPG and DPST have similar properties. Except, since the OSPG is designed to model OpenMP applications, it advances the DPST design to support a larger class of parallel programs. The following is a list of parallelism constructs that the OSPG supports over the DPST. First, the OSPG supports encoding OpenMP worksharing constructs. Second, while the DPST support task-parallelism, it only supports structured task-parallelism where each task synchronization implies a wait for all descendant tasks. In contrast, the OSPG supports taskwaits and task dependencies, which is not categorized as structured parallelism. While the OSPG can encode a larger class of parallel applications than the DPST, it maintains the DPST's nice properties. Similar to DPST, the OSPG is incrementally constructed in parallel. Further, the time and space overheads during construction and different operations are similar for both data structures.

5.2 Performance Analysis Tools

The design of different performance analysis techniques has been an active area of research for many decades. Traditional profiling tools monitor the execution trace of the program and measure metrics of interest. Such profilers aggregate their measurements at different granularities and identify program hotspots, which are parts of the program where it spends most of its time. Gprof [76, 77] pioneered this type of tools by introducing a call graph profiler, where the execution time and the number of function calls in the program are measured and presented to the developer as a ranked list of program hotspots. Path profiling [23, 24] is a variation of call graph profiling, where the calling context of a function is used to distinguish between different invocations of the function.

Many of the current profiling tools are inspired by gprof. Common between these profilers is that their measurements are primarily used to identify program hotspots [4, 5, 14, 50, 53, 59, 71, 105, 189]. These tools primarily differ from each other based on the programming model they support, how they perform measurements, and how they summarize and present their measurements to the user.

There has been a large body of work focused on profiling parallel applications. Some of these works support profiling OpenMP programs [4, 5, 14, 49, 53, 71, 105, 176, 189]. There are two common approaches used by profiling tools to measure the computation in the program. First, some tools use instrumentation to precisely record the events of interest, such as entering and exiting a function, accessing a lock, or changing wait states [5, 57, 59, 105, 174, 177, 189]. Second, to reduce runtime measurement overheads and the probe effect, some profiling tools such as Intel VTune [50] and HPCToolkit [14] use hardware event-based sampling to compute performance metrics and associate the metrics to the source code calling context [49, 53, 79, 174]. Lowering the sampling rate can result in lower overheads. However, it may result in missing some events of interest [19].

Traditional profiling tools provide insight into the program's performance issues. While all these tools measure the amount of work performed by the program, they vary in their approach to presenting developer feedback. The common views include a thread-based view of execution or function call graphs [14, 49, 52, 189]. While effective at identifying bottlenecks,

these tools are less capable of identifying which regions of the program must be optimized to improve performance. Further, while a thread-based view of the program's execution is useful in identifying what type of bottlenecks the program is experiencing and are effective in identifying load imbalance and lock contention [43, 55, 195], the insight is limited to a specific execution trace of the program. The performance issues identified in a specific execution may shift as the program is executed with a larger amount of threads. In contrast, OMP-ADVISER uses logical parallelism at the core of its profiling analysis techniques. Logical parallelism is execution independent and can be used to assess the performance of the program when run at scale. Further, OMP-ADVISER's what-if analysis technique assists developers in identifying regions that matter for improving the parallelism of the program. Hence, OMP-ADVISER analysis techniques complement traditional profiling techniques.

Some performance analysis techniques focus on identifying certain performance and scalability bottlenecks. There exists a line of work on identifying load imbalance [30, 56, 72, 95, 149, 167, 182] and threading inefficiencies [17, 60, 61, 67, 184] in parallel programs. Tallent *et al.* [183, 184] propose a blame shifting technique that is used to identify the cause of thread inefficiencies caused by lock contention in the application. To determine which regions in the program are suspects for causing thread idleness, blame shifting attributes a thread's idleness caused by lock contention to threads holding the lock. The key insight provided by blame shifting is that an idle thread is a symptom of a performance issue rather than its cause. Initially, blame shifting was designed for Cilk programs. Later, it was extended for use in OpenMP applications [122] and heterogeneous systems [39]. While these approaches help identify performance bottlenecks, their main drawback is that their analysis results are execution specific and may not be applicable when changing the runtime scheduler or the number of processors.

Parallel applications may be subject to different performance bugs. Studying what performance bugs are common in parallel applications [85, 157] can guide performance tool designers to create tools that can effectively identify or address such bugs. These studies can help improve OMP-ADVISER's design to identify other performance bottlenecks in OpenMP applications and provide more detailed feedback about the cause of bottlenecks in the program.

Many performance analysis tools listed so far, including our work with OMP-ADVISER, focus on identifying bottlenecks, and leave the task of addressing such bottlenecks to the

developers. Addressing some performance bottlenecks may require domain expertise and changes to the program's data structures and algorithms. Complementary to our work, some research proposals aim to assist developers by providing parallel libraries that provide efficient and scalable implementations of parallel data structures and algorithms [64, 163].

Parallelism and Speedup Estimates. The role of parallelism in determining the scaling of a parallel application has long been known by researchers [18]. Some early works focused on discovering instruction-level parallelism in a serial application [22, 106] using dependency analysis. Other lines of work aim to determine parallelism opportunities at other granularities such as basic blocks [101] and functions [84].

Some approaches focus on identifying and reducing the program's critical path to improve the program's parallelism [16, 36, 93, 135, 155, 171, 172, 190]. Some proposals use post-mortem analysis of the program trace to identify the program's critical path [36, 87, 93, 135]. Yang and Miller [134, 190] generate a critical path profile consisting of each function's contribution to the program's critical path. They identify the program's critical path by constructing its program activity graph (PAG) using collected program traces in the IPS [134] performance analysis tool. Slack [93] and Logical Zeroing [92] are two performance metrics that are derived from critical path in the program's activity graph that aim to estimate the impact of optimizing a function on the critical which were added to IPS-2 [135]. Bohme *et al.* [36] develop a post-mortem analysis technique to identify the critical path in MPI programs. They provide a critical path profile that identifies which program activities (*i.e.*, the call path and the process id) constitute the critical path. They propose other metrics targeting SPMD and MPMD programs that use the critical path to quantify the imbalance among MPI processes. Some research proposals compute the program's critical path on-the-fly [90, 91, 155]. Hollingsworth proposes an online algorithm that computes the program's critical path by keeping track of different variables at each process during program execution without the need to construct the program's activity graph [90, 91]. Oyama *et al.* [155] propose an online algorithm to compute the program's critical path in parallel. Compared to OMP-ADVISER, most of these tools identify a trace-specific critical path that may change by varying the number of processes or thread interleavings in the program. Further, these techniques provide little information on changes to the program's critical path after optimizing functions on the critical path.

There exists significant prior research on performance tools that use parallelism as a metric of interest to identify performance issues. ParaGraph [87] is an early performance visualization tool that illustrates the program's critical path when providing a process-based (*i.e.*, space-time diagram) view of the execution trace of the program. Grain graphs [137] visualizes the execution of an OpenMP program. It uses "instantaneous parallelism" as one of its performance metrics to identify if processor cores are being underutilized.

Kremlin [73] and Kismet [100] use parallelism measurement of program regions to parallelize and estimate the speedup of a serial program when parallelized, respectively. Similarly, DiscoPoP [115] discovers parallelism opportunities in serial programs. Unlike Kremlin, it uses a bottom-up approach to find parallelism at different granularities. Additionally, some works attempt to parallelize discovered parallelism opportunities automatically [185]. In contrast, OMP-ADVISER does not focus on the feasibility of parallelizing a code region. Instead, OMP-ADVISER aims to provide developer insight into the impact of parallelizing different regions on the program's performance.

Parallel Prophet [104] projects potential parallel speedup in serial programs. Similarly, Intel Advisor's [48] suitability analysis provides speedup estimates using different threading models, including OpenMP. Unlike Kremlin, both tools require annotations to specify which regions should be hypothetically parallelized. This type of analysis is similar to OMP-ADVISER's what-if analysis. In practice, identifying which regions in the program to annotate to assess speedup can be difficult. In contrast, OMP-ADVISER also provides an automatic what-if analysis technique that does not require user-defined annotations and can be useful to developers in finding which regions must be parallelized to reach a given threshold.

The idea of using logical parallelism measurements to identify scalability bottlenecks in parallel applications has been explored by prior research [86, 169, 192, 193]. Cilkview [86] uses Pin binary instrumentation framework [126] to construct the program's computation graph [28, 190] and computes a Cilk program's logical parallelism. Cilkprof [169] builds on the approach taken by Cilkview, and in addition to the program's logical parallelism, it computes the logical parallelism at each call site, assisting users to identify which call sites are the program's scalability bottlenecks. Both of these solutions require a serial execution of the program under test. In contrast, OMP-ADVISER can generate a program's parallelism profile in parallel and

on-the-fly. Further, OMP-ADVISER's performance model enables its what-if analysis technique to estimate the impact of parallelizing a region of code on the program's parallelism. Neither of these tools provides such a feature.

Taskprof [192] and its successor Taskprof2 [193] are closely related to our work with OMP-ADVISER. Taskprof measures logical parallelism and supports what-if analysis for task-parallel TBB programs. Additionally, Taskprof2 enables automatic what-if analysis and differential analysis. Taskprof uses the DPST [160, 162] to encode the logical series-parallel relations in a task-parallel TBB application. OMP-ADVISER's design is inspired by Taskprof. Compared to Taskprof, OMP-ADVISER uses a novel series-parallel graph data structures that enable it to support OpenMP programs that use both worksharing and tasking constructs. Further, OMP-ADVISER captures the semantics of task-parallel synchronization constructs such as OpenMP's taskwait and task dependencies that are not supported by Taskprof.

The causal profiler Coz [54] measures optimization potential in programs. By using a novel virtual speedup technique, Coz measures the effects of optimizing user-defined program on the program's performance. The key idea used by Coz's virtual speedup technique is that a selected program region, specified by progress points, can be virtually sped up by slowing down all other threads running concurrently with it. To reduce overheads, Coz uses sampling to perform each performance experiments without the requirement of adding delays to all threads in the program. To gather enough samples, Coz may need to be run multiple times. Unlike Coz, OMP-ADVISER can predict the effect of parallelizing different user-defined program regions by running the program once and constructing its performance model. Further, Coz's predictions are specific to the program execution with a given number of threads. This enables Coz to accurately estimate performance improvements for the current system and help developers find which program regions impact the program's speedup the most. In contrast, OMP-ADVISER's measurement of logical parallelism enables it to identify if a program will scale to systems with more threads.

Some works use probabilistic and statistical approaches to identify and fix scalability bottlenecks [37, 110, 111, 164, 197, 198]. Analytical performance modeling techniques aim to predict the program's scalability by providing a model that expresses execution time as a function of the number of processors in the system [37]. Calotoiu *et al.* [37, 164] propose a technique that uses statistical methods to automatically generate performance models that are

used to identify which parts of the program impact the program's scalability when run on a larger number of processors. AutomaDeD [109] uses a distributed probabilistic algorithm to find MPI tasks with the least amount of progress with low overheads. Other works, build statistical models and train them with small-scale executions of applications to diagnose scalability issues in large-scale deployment of the application [110, 111, 197, 198]. These solutions use different statistical methods to estimate performance on a larger number of cores. The quality of their predictions is dependent on the number of data points observed. In contrast, OMP-ADVISER's use of logical parallelism provides an upper on the program's performance. It can be computed without running the program with a different number of cores to generate.

Parallelism Granularity and Runtime Design. A common problem in parallel programming is to take a high-level parallel program and run it efficiently on actual hardware [11]. Prior research in this domain takes two general approaches to address this problem. First, to reduce average runtime overheads, there have been different lines of work centered around identifying the optimal parallelism granularity [82, 97, 143, 165, 166]. Second, the design of parallel runtimes with lower overheads or the ability to alleviate specific performance problems during program execution [10–12, 147].

A line of work focuses on reducing runtime overheads by automatically optimizing the program under test. For recursive task-parallel programs, Nandivada *et al.* [143] propose a range of program transformations for optimizing task parallelism by reducing the number of task creation and synchronization operations in the program. The DECAF framework [82] extends these transformations to find further opportunities to reduce runtime overheads. Iwasaki *et al.* [97] propose a compiler-based technique to find tasking cut-offs that identify the optimal task granularity. Compiler based optimizations to reduce runtime overheads [125] are complementary to our work with OMP-ADVISER. Applying some transformation in the program may excessively reduce its parallelism, which could result in lower performance. OMP-ADVISER's parallelism profile, which includes the average tasking overhead for each static directive in the program, helps identify such issues.

Another line of research aims to identify tasking granularities using profiling [165, 166]. Tgp [166] is a task granularity profiler for Java applications. It uses metrics collected in the Java Virtual Machine (JVM) to profile every task spawned and provides an aggregate

report highlighting the program's task granularities. Rosa *et al.* [165] expand on this profiling technique by providing performance estimates at different spawn sites in the program. Similar to these approaches, OMP-ADVISER provides feedback on the program's tasking overheads by estimating the cost of every OpenMP task created. However, unlike these solutions, OMP-ADVISER's logical parallelism measurement helps developers identify the right balance between lowering runtime overheads and achieving sufficient parallelism for scalable speedups.

One research direction focuses on designing runtimes that can mitigate the runtime parallelization costs [10–12, 147]. Noll *et al.* [147] propose a customized JVM that can lower the cost of parallel execution by merging concurrent tasks during program execution. Acar *et al.* [12] propose an oracle-guided scheduling technique to control the granularity of parallel programs. Using a combination of static and dynamic information, the runtime approximates the work performed by a given thread and decides if the current task should execute the current thread serially or parallel by scheduling it to another thread. A follow-up work [10] extends oracle-guided scheduling to support nested-parallel programs. Oracle-guided scheduling requires the programmer to annotate every parallel call with an expression for computing its asymptotic cost for the runtime to choose if a task should execute serially or in parallel. In contrast, Heartbeat scheduling [11] delivers provably efficient results without the need for programmer annotations. Broadly runtime techniques to control task granularity rely on task-coarsening [62, 96] or lazy splitting [75, 89, 136, 186]. Using task-coarsening, in an attempt to reduce runtime overheads, the runtime reduces parallelism by serializing logically parallel tasks. Lazy splitting adjusts task granularities by postponing the creation of new parallel tasks until worker threads become available. These runtimes alleviate the need for the programmer to tune the tasking granularities of their programs. However, heuristics to estimate runtime costs and the additional runtime overhead to make granularity control decisions can lead to suboptimal results compared to hand-tuned parallel applications. Hence, profiling techniques that provide feedback on runtime overheads remain in demand by developers.

Identifying Unintended Resource Contention. A parallel application's scalability can be limited by unintended resource contention caused by secondary effects of execution. Different profiling techniques have been proposed in identifying different types of resource contention. These works can be broadly categorized into the following: (1) identifying cache contention

in the program [35, 40, 65, 99, 118, 119, 127, 142, 188, 196], (2) identifying performance bottlenecks caused by Non-Uniform Memory Access (NUMA) systems [108, 121, 129], (3) detecting lock contention [43, 55, 184, 195], and (4) identifying and pinpointing data locality issues [120, 123] . These techniques are effective at finding specific performance bottlenecks. In contrast, OMP-ADVISER's differential analysis technique has the potential to detect and pinpoint program regions that are experiencing different types of resource contention. For OMP-ADVISER to identify a program region with a specific type of contention, the oracle execution must produce different results than the parallel execution in some metric of interest correlated to the source of contention.

Prior works have explored the idea of using differential analysis to identify performance bottlenecks [45, 94, 124, 130, 138, 173, 193]. In an early work, Mckenney [130] proposes differential profiling for parallel programs as a technique where profiling data from different program executions is compared to pinpoint performance bottlenecks in the program. In this work, differential profiling is used to pinpoint program regions with lock contention and cache-thrashing. EGprof [173] is an extension to the seminal profiling tool gprof [76, 77] that adds support for differential profiling. Through its differential analysis, eGprof provides support for comparing two gprof profiles observed during two executions. Coarfa *et al.* [45] propose a differential profiling technique to identify scalability bottlenecks in SPMD programs. This differential analysis technique uses call path profiles collected from different program executions. It compares them with the program's expected scalability and reports the function call paths with scalability bottlenecks.

OMP-ADVISER's differential analysis technique is inspired by Taskprof2 [193]. OMP-ADVISER builds on Taskprof2's differential analysis technique by supporting OpenMP parallelism constructs that do not exist in task-parallel programs analyzed by Taskprof2.

5.3 Data Race Detection Tools

There exists a large body of work studying the problem of data race detection in parallel applications. In this section, we describe closely related work.

Static data race detection tools. These tools aim to detect data races in a program using static analysis techniques. The use of static analysis enables these tools to identify data races for all inputs with low overheads. However, their main drawback is reporting many false positives caused by the conservativeness of static analysis. Some tools identify races in multithreaded programs [9, 66, 140, 158]. There have been several lines of work that utilize static analysis to identify data races in OpenMP programs [25, 42, 181, 191]. Since these tools exclusively target OpenMP, generally, they can identify more data races in OpenMP compared to their more general-purpose counterparts. OmpVerify [25], PolyOMP [42], Draco [191] use polyhedral analysis to identify data races or verify that certain OpenMP loops are data race free. OMPRacer [181], not to be confused with our work OMP-RACER, is a recent static data race detector that uses a novel value-flow analysis and happens-before tracking to identify a broader category of data races in OpenMP applications. In contrast to static data race detectors, OMP-RACER is able to identify data races in a larger class of OpenMP programs that use tasking constructs. Moreover, compiler techniques that verify loops to be data race free complement dynamic data race detectors, including our work, OMP-RACER. Since there is no need to instrument memory shared memory accesses within a data race free loop.

Dynamic data race detection tools. Dynamic data race detection uses analysis of the program's execution to detect data races. A body of work aims to identify data races in multithreaded programs [69, 168, 175]. Eraser [168] introduced the lockset algorithm to identify data races in programs that only use locks for synchronization. Some dynamic data race detectors capture the happens-before relation [112] to identify data races. FastTrack [69] improves upon the vector clock based method of capturing happens-before relations to reduce data race detection overheads. ThreadSanitizer [175], uses a hybrid approach with the aim of targeting large-scale applications. For example, by maintaining a fixed set of access histories, ThreadSanitizer achieves constant memory overhead, a trade-off between performance and accuracy.

Our work is inspired by works that capture the happens-before relation using series-parallel graphs [68, 161]. Subsequent works expand upon these techniques to support a larger class of parallel programs and perform data race detection in parallel [162, 194]. Offset-Span labeling [132] uses a labeling scheme in fork-join programs to summarize the series-parallel graph. In

contrast, OMP-RACER uses a novel series-parallel graph, OSPG, that captures the execution of OpenMP programs with a larger variety of parallelism constructs.

Our work is closely related to prior dynamic data race detection tools that target OpenMP programs. Archer[20], builds upon ThreadSanitizer by extending it to support OpenMP semantics. Compared to Archer, OMP-RACER is able to identify data races for all possible program schedules for a given input. ROMP [81], expands upon offset-span labeling by adapting it to support OpenMP constructs, including tasking and task synchronization. Asymptotically, the operations in the OSPG are comparable to ROMP's offset-span labeling since the length of labels and the depth of the OSPG grow proportional to the nesting level of the program. In practice, certain checks may be handled more efficiently in either approach.

Chapter 6
Conclusion

In this dissertation, we present the design for the OpenMP series-parallel graph (OSPG), a novel data structure that precisely captures the logical-series parallel for a large class of OpenMP applications. We show that the OSPG is useful for building OpenMP performance analysis tools and debuggers. Our parallelism profiler, OMP-ADVISER, uses the OSPG in its design for a parallelism centric performance model. This novel performance model enables our what-if analysis technique that mimics the effects of parallelization to estimate performance improvement in OpenMP before attempting to optimize the program. Further, OMP-ADVISER uses our performance model to perform fine-grained identification of regions with parallel work inflation using its differential analysis technique. Overall, making a case for measuring logical parallelism to identify performance and scalability bottlenecks in OpenMP applications. OMP-RACER, our proposed apparent data race detector, uses the OSPG to check if two memory access operations logically execute in parallel. For a given input, OMP-RACER can detect possible data races for different thread interleavings. OMP-ADVISER and OMP-RACER have proven useful in identifying performance bottlenecks and data races in OpenMP benchmarks representing realistic OpenMP applications.

6.1 Summary of Contributions

The OpenMP series-parallel graph (OSPG), described in Chapter 2 encodes the logical series-parallel relations between each serial fragment of execution in the dynamic execution trace of an OpenMP application. OSPG is inspired by prior series-parallel graphs. However, the OSPG is the first series-parallel data structure that supports modeling the execution of OpenMP applications that use both worksharing and tasking constructs. Further, the OSPG precisely

encodes the logical series-parallel semantics of certain task-based OpenMP constructs, such as taskwait and task-dependency, that are not categorized as structured parallelism.

Our design goal with the OSPG was to make it suitable for use with on-the-fly OpenMP analysis tools that can leverage the program's parallelism. Hence, an OpenMP program's OSPG can be constructed incrementally on-the-fly and in parallel.

Our performance analysis tool, OMP-ADVISER, is described in Chapter 3. OMP-ADVISER uses the OSPG alongside fine-grained measurements to construct a novel performance model for the application under analysis. This performance model enables its what-if and differential analysis techniques. OMP-ADVISER parallelism profiling algorithm aggregates measurements in the performance model to generate a parallelism-centric performance profile of the application. The parallelism profile can assist developers in pinpointing serialization bottlenecks in the application. OMP-ADVISER's what-if analysis technique uses its performance model to estimate the performance improvements in an OpenMP application when selected program regions are parallelized. This technique enables developers to assess the impact of parallelizing different program regions and prioritizing which regions to optimize before concrete optimization strategies are devised. Our automatic region identification technique expands on what-if analysis by providing a list of regions to optimize to reach a user-specified target parallelism. OMP-ADVISER also introduces a differential analysis technique that uses its performance model with differential profiling to identify performance bottlenecks caused by unintended resource contention.

Chapter 4 presents OMP-RACER, an apparent on-the-fly data race detector for OpenMP applications. To detect apparent data races, OMP-RACER uses the OSPG to identify if two memory operations may execute in parallel. Further, by devising two different strategies, fast and precise modes, to manage previous memory accesses in the program, OMP-RACER can identify apparent data races for a given input, alleviating the need for exploring other interleavings in the program. Since OMP-RACER's use of the OSPG enables it to precisely encode the logical series-parallel relations in a large class of OpenMP programs, it can identify data races with higher accuracy and precision than prior state-of-the-art OpenMP data race detection tools.

6.2 Future Work

In our tests, the OMP-ADVISER has successfully helped us identify performance bottlenecks and regions that matter to address these performance bottlenecks. However, there are different directions that we can explore to improve its efficacy and usefulness.

By design, the regions reported by OMP-ADVISER's automatic region identification lie on the program's critical path. Most often, optimizing these regions results in performance improvements. However, our automatic what-if analysis technique does not consider the feasibility of parallelization. Parallelizing and optimizing some program regions are significantly more challenging than other program regions. A program region may have some inherent dependencies that limit its parallelization or require algorithmic changes. Instead of using a greedy algorithm for automatic what-if analysis, we would like to explore an algorithm that takes the feasibility of parallelizing a program region into account, taking inspiration from prior works in this domain [133].

When implementing large-scale applications, the parallelism from the application can be realized at different levels. These include multi-node, single-node host, and single-node offloaded parallelism. However, for an application to utilize the parallelism provided by all layers, it first needs to have high enough parallelism to saturate the capacity provided by each level, which OMP-ADVISER's inherent parallelism measured by OMP-ADVISER can be useful in ensuring that the application has sufficient parallelism. However, the current OMP-ADVISER design focuses on single-node parallelism with no offloading. We want to explore ideas that can expand OMP-ADVISER to support multi-node parallelism or offloading computation to heterogeneous accelerators such as GPUs and FPGAs.

To expand OMP-ADVISER for use in offloading or multi-node systems, the costs of data movement are communication is no longer negligible compared to computation costs. Hence, we need to augment our performance model to take communication costs into account. In the context of OpenMP offloading, measuring a close upper bound on data transfer costs between host and the offloading device is challenging. However, this challenge is mitigated when analyzing high-level languages such as OpenMP. When using offloading in OpenMP, the developer must explicitly specify the data mapping between the host and device. Otherwise, the

host data gets moved via implicit rules defined in the OpenMP specification. We want to explore extending OMP-ADVISER to track all the data mappings between host and target devices and asymptotically estimate the data movement cost. With an updated performance model, we can expand OMP-ADVISER what-if analysis to answer the following question. First, does offloading a program region to an accelerator improves performance? Second, among suitable candidates for offloading, which candidates should be prioritized to run on the accelerator first? Answering these questions requires work on the current OMP-ADVISER design.

Our experience using OMP-RACER shows that in its precise mode, it can identify data races that use task dependencies and different task synchronization constructs that currently, no other OpenMP data race detector can identify. However, OMP-RACER's precise mode overhead is significant and limits its usability to small applications. One possible approach to reduce this overhead, inspired by prior tools such as RaceMob [103], is to combine static data race detection with OMP-RACER's dynamic data race detection. When using the approach, we run OMP-RACER's data race detection algorithm only in program regions that the static analyzer cannot prove the absence of data races, effectively combining the completeness and soundness of both approaches to create a more effective data race detector.

6.3 Concluding Thoughts

Initially, we started designing the OSPG data structure to create a parallelism profiler for OpenMP applications. Our first performance analysis tool, OMP-WHIP, made a case for measuring inherent parallelism and what-if analysis for identifying serialization bottlenecks in OpenMP applications. During this time, we learned about the intricacies of the OpenMP parallelism constructs and their series-parallel semantics. When designing OMP-RACER, we improved upon the OSPG design to precisely model the series-parallel relations in the presence of different task-based OpenMP synchronization constructs. We also had to devise different access history management strategies to correctly detect data races in a large class of OpenMP programs. Later, we revisited our performance analysis tool design with task-creation costs measurement, automatic what-if analysis, and differential analysis. The results of these enhancements, OMP-ADVISER, proved to be better equipped with identifying regions that

matter for performance in OpenMP applications. Realizing OSPG, OMP-ADVISER, and OMP-RACER were only made possible by building upon decades of prior research in the parallel programming domain. We hope that our work can also help form the basis for the design of additional performance analysis and debugging tools.

Bibliography

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

[2] The kastros benchmark suite. https://gitlab.inria.fr/openmp/kastors.

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

[4] Arm map, 2018. https://developer.arm.com/products/software-development-tools/hpc/arm-forge/arm-map.

[5] Paraver, 2018. https://tools.bsc.es/.

[6] C plus plus memory model reference, 2020. https://en.cppreference.com/w/cpp/language/memory_model.

[7] Coderrect openmp static data race detector, 2020. https://coderrect.com/.

[8] The llvm compiler infrastructure, 2020. https://llvm.org/.

[9] M. Abadi, C. Flanagan, and S. N. Freund. Types for safe locking: Static race detection for java. *ACM Trans. Program. Lang. Syst.*, 28(2):207–255, 2006.

[10] U. A. Acar, V. Aksenov, A. Charguéraud, and M. Rainey. Provably and practically efficient granularity control. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, PPoPP '19, page 214–228, New York, NY, USA, 2019. Association for Computing Machinery.

[11] U. A. Acar, A. Charguéraud, A. Guatto, M. Rainey, and F. Sieczkowski. Heartbeat scheduling: Provable efficiency for nested parallelism. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, page 769–782, New York, NY, USA, 2018. Association for Computing Machinery.

[12] U. A. Acar, A. Charguéraud, and M. Rainey. Oracle-Guided Scheduling for Controlling Granularity in Implicitly Parallel Languages. *Journal of Functional Programming*, 26, Nov. 2016.

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

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

[15] K. Agrawal, J. Devietti, J. T. Fineman, I.-T. A. Lee, R. Utterback, and C. Xu. Race detection and reachability in nearly series-parallel dags. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '18, page 156–171, 2018.

[16] 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*, MASCOTS '94, pages 308–317, 1994.

[17] E. Altman, M. Arnold, S. Fink, and N. Mitchell. Performance analysis of idle programs. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, pages 739–753, 2010.

[18] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), page 483–485, New York, NY, USA, 1967. Association for Computing Machinery.

[19] J. M. Anderson, L. M. Berc, J. Dean, S. Ghemawat, M. R. Henzinger, S.-T. A. Leung, R. L. Sites, M. T. Vandevoorde, C. A. Waldspurger, and W. E. Weihl. Continuous profiling: Where have all the cycles gone? *ACM Trans. Comput. Syst.*, 15(4):357–390, nov 1997.

[20] S. Atzeni, G. Gopalakrishnan, Z. Rakamaric, D. H. Ahn, I. Laguna, M. Schulz, G. L. Lee, J. Protze, and M. S. Müller. Archer: Effectively spotting data races in large openmp applications. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, IPDPS '16, pages 53–62, 2016.

[21] S. Atzeni, G. Gopalakrishnan, Z. Rakamaric, I. Laguna, G. L. Lee, and D. H. Ahn. Sword: A bounded memory-overhead detector of openmp data races in production runs. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 845–854, 2018.

[22] T. M. Austin and G. S. Sohi. Dynamic dependency analysis of ordinary programs. In *Proceedings the 19th Annual International Symposium on Computer Architecture*, pages 342–351, 1992.

[23] T. Ball. Efficiently counting program events with support for on-line queries. *ACM Trans. Program. Lang. Syst.*, 16(5):1399–1410, Sept. 1994.

[24] T. Ball and J. R. Larus. Efficient path profiling. In *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 29, page 46–57, USA, 1996. IEEE Computer Society.

[25] V. Basupalli, T. Yuki, S. Rajopadhye, A. Morvan, S. Derrien, P. Quinton, and D. Wonnacott. ompverify: Polyhedral analysis for the openmp programmer. In *OpenMP in the Petascale Era*, pages 37–53. Springer Berlin Heidelberg, 06 2011.

[26] M. A. Bender and M. Farach-Colton. The lca problem revisited. In *LATIN 2000: Theoretical Informatics*, pages 88–94, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

[27] M. A. Bender, J. T. Fineman, S. Gilbert, and C. E. Leiserson. On-the-fly maintenance of series-parallel relationships in fork-join multithreaded programs. In *Proceedings of the 16th Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA, pages 133–144, 2004.

[28] R. D. Blumofe and C. E. Leiserson. Space-efficient scheduling of multithreaded computations. In *Proceedings of the Twenty-Fifth Annual ACM Symposium on Theory of*

Computing, STOC '93, page 362–371, New York, NY, USA, 1993. Association for Computing Machinery.

[29] H.-J. Boehm and S. V. Adve. Foundations of the c++ concurrency memory model. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, page 68–78, New York, NY, USA, 2008. Association for Computing Machinery.

[30] D. Böhme, M. Geimer, L. Arnold, F. Voigtlaender, and F. Wolf. Identifying the root causes of wait states in large-scale parallel applications. *ACM Trans. Parallel Comput.*, 3(2), July 2016.

[31] N. Boushehrinejadmoradi, A. Yoga, and S. Nagarakatte. A parallelism profiler with what-if analyses for openmp programs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, SC '18, pages 16:1–16:14, 2018.

[32] N. Boushehrinejadmoradi, A. Yoga, and S. Nagarakatte. Omp-racer data race detector, 2020. https://github.com/rutgers-apl/omprace.

[33] N. Boushehrinejadmoradi, A. Yoga, and S. Nagarakatte. On-the-fly data race detection with the enhanced openmp series-parallel graph. In *OpenMP: Portable Multi-Level Parallelism on Modern Systems*, pages 149–164. Springer International Publishing, 2020.

[34] N. Boushehrinejadmoradi, A. Yoga, and S. Nagarakatte. Omp-adviser performance analysis tool, 2021. https://github.com/rutgers-apl/omp-adviser.

[35] B. Brett, P. Kumar, M. Kim, and H. Kim. Chip: A profiler to measure the effect of cache contention on scalability. In *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*, IPDPSW '13, pages 1565–1574, 2013.

[36] 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*, pages 1330–1340, 2012.

[37] 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*, SC '13, pages 45:1–45:12, 2013.

[38] V. Cavé, J. Zhao, J. Shirako, and V. Sarkar. Habanero-java: The new adventures of old x10. In *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java*, PPPJ, pages 51–61, 2011.

[39] M. Chabbi, K. Murthy, M. Fagan, and J. Mellor-Crummey. Effective sampling-driven performance tools for gpu-accelerated supercomputers. In *SC '13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–12, 2013.

[40] M. Chabbi, S. Wen, and X. Liu. Featherlight on-the-fly false-sharing detection. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '18, pages 152–167, 2018.

[41] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. 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, pages 519–538, 2005.

[42] P. Chatarasi, J. Shirako, M. Kong, and V. Sarkar. An extended polyhedral model for spmd programs and its use in static data race detection. In *Languages and Compilers for Parallel Computing: 29th International Workshop, LCPC 2016, Rochester, NY, USA, September 28-30, 2016, Revised Papers*, pages 106–120, 2017.

[43] G. Chen and P. Stenstrom. 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, pages 71:1–71:11, 2012.

[44] G.-I. Cheng, M. Feng, C. E. Leiserson, K. H. Randall, and A. F. Stark. Detecting data races in cilk programs that use locks. In *Proceedings of the 10th ACM Symposium on Parallel Algorithms and Architectures*, SPAA, pages 298–309, 1998.

[45] 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*, ICS '07, pages 13–22, 2007.

[46] R. Cole and R. Hariharan. Dynamic lca queries on trees. *SIAM Journal on Computing*, 34, 03 2002.

[47] I. Corporation. Intel cilk plus, 2017. https://www.cilkplus.org.

[48] I. Corporation. Intel advisor, 2020. https://software.intel.com/content/www/us/en/develop/tools/advisor.html.

[49] I. Corporation. Intel parallel studio xe, 2020. https://software.intel.com/content/www/us/en/develop/tools/parallel-studio-xe.html.

[50] I. Corporation. Vtune profiler, 2020. https://software.intel.com/content/www/us/en/develop/tools/vtune-profiler.html.

[51] N. Corporation. Nvidia cuda, 2020. https://developer.nvidia.com/cuda-zone.

[52] N. Corporation. Nvidia visual profiler, 2020. https://developer.nvidia.com/nvidia-visual-profiler.

[53] Cray Inc. Using Cray Performance Measurement and Analysis Tools, 2015. http://docs.cray.com/PDF/Cray_Performance_Measurement_and_Analysis_Tools_User_Guide_640.pdf.

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

[55] F. David, G. Thomas, J. Lawall, and G. Muller. Continuously measuring critical section pressure with the free-lunch profiler. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA, pages 291–307, 2014.

[56] L. DeRose, B. Homer, and D. Johnson. Detecting application load imbalance on high end massively parallel systems. In *Proceedings of the 13th International Euro-Par Conference on Parallel Processing*, Euro-Par, pages 150–159, 2007.

[57] Y. Ding, K. Hu, K. Wu, and Z. Zhao. Performance monitoring and analysis of task-based openmp. *PLOS ONE*, pages 1–12, 2013.

[58] A. Dinning and E. Schonberg. An empirical comparison of monitoring algorithms for access anomaly detection. In *Proceedings of the Second ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, PPOPP '90, page 1–10, New York, NY, USA, 1990. Association for Computing Machinery.

[59] 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.

[60] K. Du Bois, S. Eyerman, J. B. Sartor, and L. Eeckhout. Criticality stacks: Identifying critical threads in parallel programs using synchronization behavior. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA, pages 511–522, 2013.

[61] K. Du Bois, J. B. Sartor, S. Eyerman, and L. Eeckhout. Bottle graphs: Visualizing scalability bottlenecks in multi-threaded applications. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA, pages 355–372, 2013.

[62] A. Duran, J. Corbalan, and E. Ayguade. An adaptive cut-off for task parallelism. In *SC '08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, pages 1–11, 2008.

[63] 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*, pages 124–131, 2009.

[64] H. C. Edwards and C. R. Trott. Kokkos: Enabling performance portability across manycore architectures. In *2013 Extreme Scaling Workshop (xsw 2013)*, pages 18–24, 2013.

[65] A. Eizenberg, S. Hu, G. Pokam, and J. Devietti. Remix: Online detection and repair of cache contention for the jvm. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, pages 251–265, 2016.

[66] D. Engler and K. Ashcraft. Racerx: Effective, static detection of race conditions and deadlocks. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP, pages 237–252, 2003.

[67] S. Eyerman, K. Du Bois, and L. Eeckhout. Speedup stacks: Identifying scaling bottlenecks in multi-threaded applications. In *Proceedings of the 2012 IEEE International Symposium on Performance Analysis of Systems & Software*, ISPASS, pages 145–155, 2012.

[68] 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*, SPAA, pages 1–11, 1997.

[69] C. Flanagan and S. N. Freund. Fasttrack: Efficient and precise dynamic race detection. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 121–133, 2009.

[70] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, PLDI, pages 212–223, 1998.

[71] 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*, IWOMP'05/IWOMP'06, pages 15–23, 2008.

[72] T. Gamblin, B. R. de Supinski, M. Schulz, R. Fowler, and D. A. Reed. Scalable load-balance measurement for spmd codes. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, SC '08. IEEE Press, 2008.

[73] S. Garcia, D. Jeon, C. M. Louie, and M. B. Taylor. Kremlin: Rethinking and rebooting gprof for the multicore age. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI, pages 458–469, 2011.

[74] T. Gautier, C. Perez, and J. Richard. On the impact of openmp task granularity. In B. R. de Supinski, P. Valero-Lara, X. Martorell, S. Mateo Bellido, and J. Labarta, editors, *Evolving OpenMP for Evolving Architectures*, pages 205–221. Springer International Publishing, 2018.

[75] S. C. Goldstein, K. E. Schauser, and D. E. Culler. Lazy threads: Implementing a fast parallel call. *Journal of Parallel and Distributed Computing*, 37(1):5 – 20, 1996.

[76] S. L. Graham, P. B. Kessler, and M. K. Mckusick. Gprof: A call graph execution profiler. In *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction*, SIGPLAN, pages 120–126, 1982.

[77] S. L. Graham, P. B. Kessler, and M. K. McKusick. An execution profiler for modular programs. *Software: Practice and Experience*, 13(8):671–685, 1983.

[78] B. Gregg. Linux perf examples, July 2020. http://www.brendangregg.com/perf.html.

[79] T. P. Group. Pgi profiler user's guide - pgi compilers, 2017. https://www.pgroup.com/resources/pgprof-quickstart.htm.

[80] Y. Gu. Romp data race detector, 2018. https://github.com/zygyz/romp/tree/09fddd023cf4df6b11e38190e1834fc29d2c6998.

[81] Y. Gu and J. Mellor-Crummey. Dynamic data race detection for openmp programs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, SC '18, pages 61:1–61:12, 2018.

[82] S. Gupta, R. Shrivastava, and V. K. Nandivada. Optimizing recursive task parallel programs. In *Proceedings of the International Conference on Supercomputing*, ICS '17, pages 11:1–11:11, 2017.

[83] R. J. Hall. Call path profiling. In *Proceedings of the 14th International Conference on Software Engineering*, ICSE '92, pages 296–306, 1992.

[84] C. Hammacher, K. Streit, S. Hack, and A. Zeller. Profiling java programs for parallelism. In *2009 ICSE Workshop on Multicore Software Engineering*, pages 49–55, 2009.

[85] X. Han and T. Yu. An empirical study on performance bugs for highly configurable software systems. In *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM, pages 23:1–23:10, 2016.

[86] 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*, SPAA, pages 145–156, 2010.

[87] M. T. Heath, A. D. Malony, and D. T. Rover. The visual display of parallel performance data. *Computer*, 28(11):21–28, 1995.

[88] T. Hilbrich, J. Protze, M. Schulz, B. R. de Supinski, and M. S. Müller. Mpi runtime error detection with must: Advances in deadlock detection. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 30:1–30:11, 2012.

[89] T. Hiraishi, M. Yasugi, S. Umatani, and T. Yuasa. Backtracking-based load balancing. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '09, page 55–64, New York, NY, USA, 2009. Association for Computing Machinery.

[90] J. K. Hollingsworth. An online computation of critical path profiling. In *Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools*, SPDT '96, page 11–20, New York, NY, USA, 1996. Association for Computing Machinery.

[91] J. K. Hollingsworth. Critical path profiling of message passing and shared-memory programs. *IEEE Transactions on Parallel and Distributed Systems*, 9(10):1029–1040, 1998.

[92] J. K. Hollingsworth and B. P. Miller. Parallel program performance metrics: A comparison and validation. In *Proceedings of the 1992 ACM/IEEE Conference on Supercomputing*, Supercomputing '92, page 4–13, Washington, DC, USA, 1992. IEEE Computer Society Press.

[93] J. K. Hollingsworth and B. P. Miller. Slack: A new performance metric for parallel programs. Technical report, University of Wisconsin-Madison, 1994.

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

[95] K. A. Huck and J. Labarta. Detailed load balance analysis of large scale parallel applications. In *2010 39th International Conference on Parallel Processing*, pages 535–544, 2010.

[96] L. Huelsbergen, J. Larus, and A. Aiken. Using the run-time sizes of data structures to guide parallel-thread creation. In *LFP '94*, 1994.

[97] S. Iwasaki and K. Taura. A static cut-off for task parallel programs. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*, PACT '16, pages 139–150, 2016.

[98] A. Jannesari, K. Bao, V. Pankratius, and W. Tichy. Helgrind+: An efficient dynamic race detector. In *2009 IEEE International Symposium on Parallel Distributed Processing*, pages 1–13, 05 2009.

[99] S. Jayasena, S. Amarasinghe, A. Abeyweera, G. Amarasinghe, H. De Silva, S. Rathnayake, X. Meng, and Y. Liu. Detection of false sharing using machine learning. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 30:1–30:9, 2013.

[100] 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, OOPSLA, pages 519–536, 2011.

[101] M. Kambadur, K. Tang, and M. A. Kim. Parashares: Finding the important basic blocks in multithreaded programs. In *Proceedings of Euro-Par 2014 Parallel Processing: 20th International Conference*, Euro-Par, pages 75–86, 2014.

[102] I. Karlin, J. Keasler, and R. Neely. Lulesh 2.0 updates and changes. Technical Report LLNL-TR-641973, August 2013.

[103] B. Kasikci, C. Zamfir, and G. Candea. Racemob: Crowdsourced data race detection. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP, pages 406–422, 2013.

[104] 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*, pages 1318–1329, 2012.

[105] 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*, pages 139–155. 2008.

[106] M. Kumar. Measuring parallelism in computation-intensive scientific/engineering applications. *IEEE Transactions on Computers*, 37(9):1088–1098, 1988.

[107] L. L. N. Laboratory. Llnl sequoia benchmarks, 2017. https://asc.llnl.gov/sequoia/benchmarks.

[108] R. Lachaize, B. Lepers, and V. Quema. Memprof: A memory profiler for NUMA multicore systems. In *Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12)*, USENIX ATC '12, pages 53–64, 2012.

[109] I. Laguna, D. H. Ahn, B. R. de Supinski, S. Bagchi, and T. Gamblin. Probabilistic diagnosis of performance faults in large-scale parallel applications. In *Proceedings of*

the 21st International Conference on Parallel Architectures and Compilation Techniques, PACT '12, pages 213–222, 2012.

[110] I. Laguna, D. H. Ahn, B. R. de Supinski, T. Gamblin, G. L. Lee, M. Schulz, S. Bagchi, M. Kulkarni, B. Zhou, Z. Chen, and F. Qin. Debugging high-performance computing applications at massive scales. *Commun. ACM*, 58(9):72–81, Aug. 2015.

[111] I. Laguna and M. Schulz. Pinpointing scale-dependent integer overflow bugs in large-scale parallel applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '16, pages 19:1–19:12, 2016.

[112] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, pages 558–565, 1978.

[113] E. A. Lee. The problem with threads. *Computer*, 39(5):33–42, May 2006.

[114] T. Li, A. R. Lebeck, and D. J. Sorin. Spin detection hardware for improved management of multithreaded systems. *IEEE Transactions on Parallel and Distributed Systems*, 17(6):508–521, 2006.

[115] Z. Li, A. Jannesari, and F. Wolf. Discovery of potential parallelism in sequential programs. In *2013 42nd International Conference on Parallel Processing*, pages 1004–1013, 2013.

[116] C. Liao, P.-H. Lin, J. Asplund, M. Schordan, and I. Karlin. Dataracebench: A benchmark suite for systematic evaluation of data race detection tools. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '17, pages 11:1–11:14, 2017.

[117] Y. Lin and S. S. Kulkarni. Automatic repair for multi-threaded programs with deadlock/livelock using maximum satisfiability. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, page 237–247, New York, NY, USA, 2014. Association for Computing Machinery.

[118] T. Liu and E. D. Berger. Sheriff: Precise detection and automatic mitigation of false sharing. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11, pages 3–18, 2011.

[119] T. Liu, C. Tian, Z. Hu, and E. D. Berger. Predator: Predictive false sharing detection. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14, pages 3–14, 2014.

[120] X. Liu and J. Mellor-Crummey. Pinpointing data locality bottlenecks with low overhead. In *2013 IEEE International Symposium on Performance Analysis of Systems and Software*, ISPASS, pages 183–193, 2013.

[121] X. Liu and J. Mellor-Crummey. A tool to analyze the performance of multithreaded programs on numa architectures. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP, pages 259–272, 2014.

[122] 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*, ICS, pages 69–80, 2013.

[123] X. Liu, K. Sharma, and J. Mellor-Crummey. Arraytool: A lightweight profiler to guide array regrouping. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT, pages 405–416, 2014.

[124] 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*, SC, pages 47:1–47:12, 2015.

[125] P. LOPEZ, M. HERMENEGILDO, and S. DEBRAY. A methodology for granularity-based control of parallelism in logic programs. *Journal of Symbolic Computation*, 21(4):715 – 734, 1996.

[126] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic

instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI, pages 190–200, 2005.

[127] L. Luo, A. Sriraman, B. Fugate, S. Hu, G. Pokam, C. J. Newburn, and J. Devietti. LASER: light, accurate sharing detection and repair. In *2016 IEEE International Symposium on High Performance Computer Architecture, HPCA 2016, Barcelona, Spain, March 12-16, 2016*, pages 261–273, 2016.

[128] M. McCool, A. Robison, and J. Reinders. *Structured Parallel Programming: Patterns for Efficient Computation*. Morgan Kaufmann, 2012.

[129] C. McCurdy and J. Vetter. Memphis: Finding and fixing numa-related performance problems on multi-core platforms. In *2010 IEEE International Symposium on Performance Analysis of Systems Software (ISPASS)*, pages 87–96, 2010.

[130] P. E. McKenney. Differential profiling. *Software - Practice & Experience*, pages 219–234, 1999.

[131] P. E. McKenney. Is parallel programming hard, and, if so, what can you do about it? (v2017.01.02a). *CoRR*, abs/1701.00854, 2017.

[132] J. Mellor-Crummey. On-the-fly detection of data races for programs with nested fork-join parallelism. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, Supercomputing, pages 24–33, 1991.

[133] C. Mendis, J. Bosboom, K. Wu, S. Kamil, J. Ragan-Kelley, S. Paris, Q. Zhao, and S. Amarasinghe. Helium: Lifting high-performance stencil kernels from stripped x86 binaries to halide dsl code. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pages 391–402, 2015.

[134] B. Miller and C.-Q. Yang. Ips: An interactive and automatic performance measurement tool for parallel and distributed programs. volume 7, pages 482–489, 01 1987.

[135] 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*, 1(2):206–217, Apr. 1990.

[136] E. Mohr, D. A. Kranz, and R. H. Halstead. Lazy task creation: a technique for increasing the granularity of parallel programs. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):264–280, 1991.

[137] 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*, PPoPP '16, pages 28:1–28:13, 2016.

[138] K. Nagaraj, C. Killian, and J. Neville. Structured comparative analysis of systems logs to diagnose performance problems. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 353–366, 2012.

[139] S. Nagarakatte, J. Zhao, M. M. K. Martin, and S. Zdancewic. Cets: Compiler enforced temporal safety for c. In *Proceedings of the 2010 International Symposium on Memory Management*, 2010.

[140] M. Naik, A. Aiken, and J. Whaley. Effective static race detection for java. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI, pages 308–319, 2006.

[141] M. Naik, C. Park, K. Sen, and D. Gay. Effective static deadlock detection. In *2009 IEEE 31st International Conference on Software Engineering*, pages 386–396, 2009.

[142] M. Nanavati, M. Spear, N. Taylor, S. Rajagopalan, D. T. Meyer, W. Aiello, and A. Warfield. Whose cache line is it anyway?: Operating system support for live detection and repair of false sharing. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 141–154, 2013.

[143] V. K. Nandivada, J. Shirako, J. Zhao, and V. Sarkar. A transformation framework for optimizing task-parallel programs. *ACM Trans. Program. Lang. Syst.*, 35(1), Apr. 2013.

[144] R. Netzer and B. P. Miller. Detecting data races in parallel program executions. In *In Advances in Languages and Compilers for Parallel Computing, 1990 Workshop*, pages 109–129. MIT Press, 1989.

[145] R. H. Netzer, S. Ghosh, R. H. B, and N. S. Ghosh. Efficient race condition detection for shared-memory programs with post/wait synchronization, 1992.

[146] R. H. B. Netzer and B. P. Miller. What are race conditions?: Some issues and formalizations. *ACM Lett. Program. Lang. Syst.*, pages 74–88, 1992.

[147] A. Noll and T. Gross. Online feedback-directed optimizations for parallel java code. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '13, page 713–728, New York, NY, USA, 2013. Association for Computing Machinery.

[148] I. Nudler and L. Rudolph. Tools for the efficient development of efficient parallel programs. In *Proceedings of the 1st Israeli conference on computer system engineering*, 1988.

[149] J. Oh, C. J. Hughes, G. Venkataramani, and M. Prvulovic. Lime: A framework for debugging load imbalance in multi-threaded execution. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE, pages 201–210, 2011.

[150] 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*, SC '12, pages 65:1–65:12, 2012.

[151] OpenACC organization members. Openacc 3.0 specification, Feb. 2020. https://www.openacc.org/sites/default/files/inline-images/Specification/OpenACC.3.0.pdf.

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

[153] OpenMP Architecture Review Board. Openmp 5.0 complete specification, Nov. 2018. https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf.

[154] OpenMP Architecture Review Board. Who's using openmp?, 2020. https://www.openmp.org/about/whos-using-openmp/.

[155] Y. Oyama, K. Taura, and A. Yonezawa. Online computation of critical paths for multi-threaded languages. In *Proceedings of the 15 IPDPS 2000 Workshops on Parallel and Distributed Processing*, IPDPS, pages 301–313, 2000.

[156] O. Pell and O. Mencer. Surviving the end of frequency scaling with reconfigurable dataflow computing. *SIGARCH Comput. Archit. News*, 39(4):60–65, Dec. 2011.

[157] G. Pinto, A. Canino, F. Castor, G. Xu, and Y. D. Liu. Understanding and overcoming parallelism bottlenecks in forkjoin applications. In *Proceedings of the 32st IEEE/ACM International Conference on Automated Software Engineering*, ASE '17, 2017.

[158] P. Pratikakis, J. S. Foster, and M. Hicks. Locksmith: Context-sensitive correlation analysis for race detection. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI, pages 320–331, 2006.

[159] J. Protze. Archer branch with support for avoiding report of false task-local data races, 2020. https://github.com/jprotze/llvm-project/blob/archer-task-fibers/openmp/tools/archer/ompt-tsan.cpp#L1200.

[160] R. Raman. *Dynamic Data Race Detection for Structured Parallelism*. PhD thesis, Rice University, 2012.

[161] R. Raman, J. Zhao, V. Sarkar, M. Vechev, and E. Yahav. Efficient data race detection for async-finish parallelism. In *Proceedings of the 1st International Conference on Runtime Verification*, RV, pages 368–383, 2010.

[162] 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*, PLDI, pages 531–542, 2012.

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

[164] 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.

[165] A. Rosà, E. Rosales, and W. Binder. Analyzing and optimizing task granularity on the jvm. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, CGO 2018, pages 27–37, 2018.

[166] E. Rosales, A. Rosà, and W. Binder. tgp: A task-granularity profiler for the java virtual machine. In *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*, pages 570–575, 2017.

[167] M. Roth, M. Best, C. Mustard, and A. Fedorova. Deconstructing the overhead in parallel applications. In *Proceedings - 2012 IEEE International Symposium on Workload Characterization, IISWC 2012*, IISWC '12, pages 59–68, 2012.

[168] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multi-threaded programs. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, SOSP, pages 27–37, 1997.

[169] 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*, SPAA, pages 89–100, 2015.

[170] B. Schieber and U. Vishkin. On finding lowest common ancestors: Simplification and parallelization. In J. H. Reif, editor, *VLSI Algorithms and Architectures*, pages 111–123, New York, NY, 1988. Springer New York.

[171] 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*, pages 186–195, 2014.

[172] M. Schulz. Extracting critical path graphs from mpi applications. In *2005 IEEE International Conference on Cluster Computing*, pages 1–10, 2005.

[173] 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*, pages 97–106, 2007.

[174] 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.*, pages 105–121, 2008.

[175] K. Serebryany and T. Iskhodzhanov. Threadsanitizer: Data race detection in practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications*, WBIA, pages 62–71, 2009.

[176] H. Servat, G. Llort, K. Huck, J. GiméNez, and J. Labarta. Framework for a productive performance optimization. *Parallel Comput.*, 39(8):336–353, 2013.

[177] S. S. Shende and A. D. Malony. The tau parallel performance system. *International Journal of High Performance Computing Applications*, pages 287–311, 2006.

[178] 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*, SPAA, pages 68–70, 2012.

[179] S. P. E. C. (SPEC). Spec omp 2012 benchmarks, 2012. https://www.spec.org/omp2012/.

[180] I. standardization working group for the programming language C. C programming language standard, 2020. http://www.open-std.org/jtc1/sc22/wg14/.

[181] B. Swain, Y. Li, P. Liu, I. Laguna, G. Georgakoudis, and J. Huang. Ompracer: A scalable and precise static race detector for openmp programs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '20. IEEE Press, 2020.

[182] N. R. Tallent, L. Adhianto, and J. M. Mellor-Crummey. 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, pages 1–11, 2010.

[183] N. R. Tallent and J. M. Mellor-Crummey. Effective performance measurement and analysis of multithreaded applications. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP, pages 229–240, 2009.

[184] N. R. Tallent, J. M. Mellor-Crummey, and A. Porterfield. Analyzing lock contention in multithreaded applications. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP, pages 269–280, 2010.

[185] G. Tournavitis, Z. Wang, B. Franke, and M. F. O'Boyle. Towards a holistic approach to auto-parallelization: Integrating profile-driven parallelism detection and machine-learning based mapping. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, page 177–187, New York, NY, USA, 2009.

[186] A. Tzannes, G. C. Caragea, U. Vishkin, and R. Barua. Lazy scheduling: A runtime adaptive scheduler for declarative parallelism. *ACM Trans. Program. Lang. Syst.*, 36(3), Sept. 2014.

[187] R. Utterback, K. Agrawal, J. Fineman, and I.-T. A. Lee. Efficient race detection with futures. *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, Feb 2019.

[188] S. Wen, X. Liu, J. Byrne, and M. Chabbi. Watching for software inefficiencies with witch. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, pages 332–347, 2018.

[189] 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*, pages 157–167. 2008.

[190] C. . 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*, pages 366–373, 1988.

[191] F. Ye, M. Schordan, C. Liao, P.-H. Lin, I. Karlin, and V. Sarkar. Using polyhedral analysis to verify openmp applications are data race free. In *2018 IEEE/ACM 2nd International Workshop on Software Correctness for HPC Applications (Correctness)*, pages 42–50, 11 2018.

[192] 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*, ESEC/FSE 2017, pages 15–26, 2017.

[193] A. Yoga and S. Nagarakatte. Parallelism-centric what-if and differential analyses. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, page 485–501, 2019.

[194] 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*, FSE, pages 833–845, 2016.

[195] T. Yu and M. Pradel. Syncprof: Detecting, localizing, and optimizing synchronization bottlenecks. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA, pages 389–400, 2016.

[196] Q. Zhao, D. Koh, S. Raza, D. Bruening, W.-F. Wong, and S. Amarasinghe. Dynamic cache contention detection in multi-threaded applications. In *Proceedings of the 7th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '11, pages 27–38, 2011.

[197] B. Zhou, M. Kulkarni, and S. Bagchi. Vrisha: Using scaling properties of parallel programs for bug detection and localization. In *Proceedings of the 20th International Symposium on High Performance Distributed Computing*, HPDC '11, pages 85–96, 2011.

[198] B. Zhou, J. Too, M. Kulkarni, and S. Bagchi. Wukong: Automatically detecting and localizing bugs that manifest at large system scales. In *Proceedings of the 22nd International Symposium on High-performance Parallel and Distributed Computing*, HPDC '13, pages 131–142, 2013.