PARALLELISM-DRIVEN PERFORMANCE ANALYSIS TECHNIQUES FOR TASK PARALLEL PROGRAMS

by

ADARSH YOGA

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

October, 2019

ABSTRACT OF THE DISSERTATION

Parallelism-Driven Performance Analysis Techniques for Task Parallel Programs

By ADARSH YOGA

Dissertation Director:

Santosh Nagarakatte

Performance analysis of parallel programs continues to be challenging for programmers. Programmers have to account for several factors to extract the best possible performance from parallel programs. First, programs must have adequate parallel computation that is evenly distributed to keep all processors busy during execution. Second, programs must reduce secondary effects caused by interactions in hardware, which can degrade performance. Third, performance problems due to inadequate parallel computation and secondary effects can get magnified when programs are executed at scale. Fourth, programs must ensure minimal overhead from other sources like runtime schedulers, lock contention, and heavyweight abstractions in the software stack. To diagnose performance issues in parallel programs, programmers rely on profilers to obtain performance insights. Although profiling is a well-researched area, existing profilers primarily provide information on where a program spends its time. They fail to highlight the parts of the program that matter in improving the performance and the scalability of a program.

This dissertation makes a case for using logical parallelism to identify parts of the program that matter in improving the performance of task parallel programs. It makes the following contributions. First, it describes a scalable and an efficient technique to compute the logical parallelism of a program. Logical parallelism defines the speedup of a program in the limit. Logical parallelism is an ideal metric to assess if a program has adequate parallel computation to attain scalable speedup on any number of processors. Second, it introduces a novel performance model to compute the logical parallelism of a program. To enable parallelism computation, the performance model encodes the series-parallel relations in the program and fine-grain work performed in an execution. Third, it presents a technique, called what-if analyses, that uses the parallelism computation and the performance model to identify parts of a program that matter in improving the parallelism. Finally, it describes a differential analysis technique that uses the performance model to identify parts of the program that matter in addressing secondary effects.

Using the techniques proposed in this dissertation we have developed TaskProf, a profiler and an adviser for task parallel programs. The performance insights gained from TaskProf have enabled us to design concrete optimizations and increase the performance of a range of applications. The techniques presented in this dissertation and our profiler TaskProf demonstrate that by designing rich abstractions that enable analyses to measure parallelism, expose secondary effects and evaluate how performance changes when regions are optimized, one can identify parts of the program that matter in improving the performance of task parallel programs.

Acknowledgements

While many people have helped shape my PhD journey, none have had a deeper impact on my career as a researcher than my adviser, Santosh Nagarakatte. During my early years as a PhD student, Santosh spent an inordinate amount of time discussing research ideas, jointly reading papers with me, and debugging my code on numerous occasions. He has always encouraged me to explore all aspects of a problem and helped convert my fledgling ideas to concrete solutions. I am fortunate to have started my research career with Santosh's guidance and mentorship.

I would like to thank the members of my dissertation committee, Ulrich Kremer, Sudarsun Kannan, and Madanlal Musuvathi for their insightful feedback about my research work that has helped shape this dissertation. I would also like to thank Milind Chabbi and Harish Patil for mentoring me during my internships at HP Labs and Intel respectively, and assisting me during my job search.

I feel grateful to have received the opportunity to learn from and interact with the faculty members at the Computer Science Department, particularly Vinod Ganapathy, Ulrich Kremer, Abhishek Bhattacharjee, Badri Nath, and Tomasz Imielinski. I am especially thankful to Vinod for providing valuable feedback about the initial directions of my research. I learned from Uli and Badri the importance of explaining complex ideas in simple language that could be followed by everybody. A few minutes of laughter-filled conversation with Tomasz would help brighten my day. I am forever grateful to Abhishek for his encouraging words during some of the tougher times in my PhD journey. I would also like to thank the administrative and technical staff at the Computer Science Department for their support.

I will always cherish the time I have spent at Rutgers - I made some great friends who made this journey enjoyable. I want to especially thank my lab-mates – David

Menendez, Jay Lim, Nader Boushehrinejadmoradi, Mohammadreza Soltaniyeh, and Sangeeta Chowdhary who were always willing to help refine the finer details of my solutions, polish my talks and proof-read my papers. I am grateful for their time and support. Beyond my lab, I was fortunate to pursue my PhD along with many bright PhD students – Jeff Ames, Guilherme, Zi Yan, Binh Pham, Jan Vesely, Ioannis, Liu, Hai Nyugen, and Cheng Li. The numerous interactions I have had with them was always refreshing.

I am deeply indebted to my family – my wife, my parents and parents-in-law for having shared every step of this journey with me. I am especially thankful to my parents for providing me with great education and inculcating sound values, which have helped me navigate numerous challenges in my research career. My wife, Maitreyi, has been a constant source of moral support and encouragement for me throughout this journey. I could not have imagined completing my PhD without her by my side. I would also like to thank my friends, Arvind and Ranga for all the fun conversations about sports, politics, and everyday life that provided refreshing breaks from work. Lastly, I would like to thank my brother, Abhishek Yoga for being my guiding light and always encouraging me to be my best self.

Dedication

In memory of my brother, Abhishek Yoga.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

The end of Moore's law [122, 123] has made parallel and heterogenous architectures ubiquitous in general purpose computing. With the proliferation of parallel hardware, parallel programming has become mainstream. Now every programmer has to write parallel programs to take advantage of the hardware features. However, designing parallel programs that have good performance is challenging. The programmer has to, first, come up with a parallel algorithm that produces correct results when executed concurrently. Typically, programmers start with a sequential algorithm and incrementally add parallel computation to it. Then, the programmer has to tune the program to ensure that the program has scalable performance. This is particularly challenging since the performance of a parallel program can be influenced by several factors, some due to inefficiencies in the program and others caused by adverse interactions during execution on hardware. The programmer has to consider all of these factors to obtain the best possible performance.

A parallel program should have sufficient computation that can be executed in parallel. A program that does not have sufficient parallel computation will have large sections that will be executed serially. The speedup of the program will be limited by the serial sections of the program (*i.e.*, from Amdahl's law [13]). Hence, programmers must strive to reduce the serial computation and add sufficient parallel computation to obtain optimal performance.

Even if a program has sufficient parallel computation, the computation should be evenly distributed among all processors. Uneven distribution of computation causes load imbalance where some processors remain idle while other processors execute their computation. This increases the overall execution time of a parallel program. Ensuring that the parallel computation is evenly distributed is a deceptively hard problem. At the level of the program source code, a parallel computation may appear to be evenly distributed. However, during execution the individual computations can execute for different amounts of time and thereby introduce load imbalance in the program.

A program with sufficient parallel computation that is evenly distributed can expect to have good performance. However, the parallel execution can experience secondary effects that adversely impact the performance of a program. Modern computer systems are highly complex with a large number of processor cores. These systems employ deep memory hierarchies comprising multiple levels of caches and memory [18]. Furthermore, many modern systems are multi-socket NUMA systems wherein, instead of having all cores integrated on a single socket, they are split on to multiple sockets with each socket having a directly attached memory to increase memory bandwidth. In addition, modern systems are further complicated by custom hardware accelerators that have their own memory hierarchy. Although modern systems are built to perform execution in parallel, contention for hardware resources or frequent data movement across the deep memory hierarchy, sockets or accelerators can cause secondary effects which degrade the performance of a program. Performance issues caused by secondary effects are particularly challenging to diagnose since a programmer has to somehow obtain insights into the execution of a program on hardware.

To highlight the challenges involved in parallel programming, let us consider designing a parallel program using a thread-based parallel programming model (for *e.g.*, using pthreads [129], Windows threads [118], or Java concurrency [97]). In such a model, the programmer explicitly creates each parallel thread and assigns it a computation to execute. The creation of threads and constant context switching between threads are expensive operations. Hence, a reasonable strategy is to create as many threads as the number of processors on the machine. The operating system scheduler then takes care of mapping these threads to the processors and executing their respective computations in parallel. The threads can communicate by updating shared data and the programmer can coordinate the updates using synchronization.

While designing the program, the programmer must account for all the factors that influence the performance of the program. Let us consider that the program being designed takes an array of integers as input. The program iterates through the input array, checks if each entry is a prime [141], and performs a computation on the prime integer. Since the primality test for each entry can be performed independent of the other entries in the input array, the program has sufficient computation that can be performed in parallel. To divide the computation evenly among the threads, a reasonable approach would be to split the input array of integers into as many chunks as the number of threads and assign each chunk to a separate thread. Figure 1.1 (a) shows the mapping of chunks of the input array to the execution threads. However, even though the input array has been split evenly, the amount of computation performed in each chunk may not be similar since the time taken by the primality test will not be the same for different integers. Hence, the computation can still have load imbalance.

Even if the computation is somehow load balanced, the performance of the program can be affected by the presence of secondary effects. For instance, parallel threads accessing entries of the input array that are allocated on the same cache line can cause secondary effects due to false sharing [32]. The programmer has to carefully design the program while ensuring potential secondary effects are minimized or totally eliminated.

All of these factors are further amplified when the program is executed at scale on machines with greater processor core count. A program may not have any performance issues when it is executed in a small scale setting on a machine with a small number of processors. Performance issues may manifest only when the program is executed on large scale machines with a greater number of processors. Ideally, the programmer would benefit from identifying the performance problems that may occur on large scale machines even while executing the program in the small scale setting.

1.1 Task Parallelism

Task parallelism [20, 29, 64, 96] addresses the problem of load imbalance in parallel computations by automatically readjusting the load during execution. In the task parallel framework, the programmer expresses parallel work in terms of light-weight structures known as *tasks*. Then, a runtime scheduler takes care of mapping the tasks

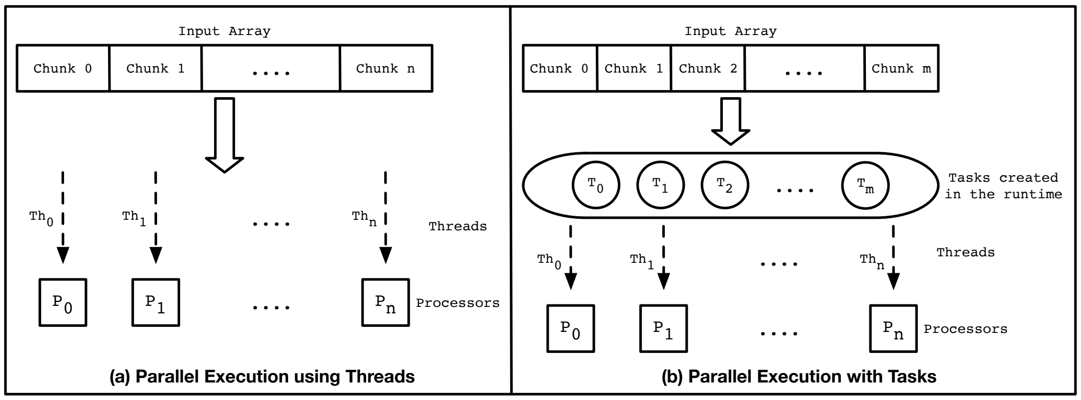


Figure 1.1: Illustration of the parallel executions of a program that takes an input array of integers and performs some computation on all the prime numbers in the array. Figure (a) shows an execution of the program using threads where the input array is divided into as many chunks as the number of threads and each chunk is processed by a single thread. Figure (b) shows the execution with tasks, where the array is divided into larger number chunks than the threads and each chunk is assigned to a task. The task runtime balances the load in the computation by dynamically assigning the tasks to the threads that become idle.

to the threads. Creating tasks is orders of magnitude less expensive than creating threads [139]. As a result, a task parallel program can afford to create more tasks than the number of processors on the execution machine. A common technique that is used in many task runtime schedulers to assign tasks to idle threads is the randomized work stealing technique [15, 22, 29, 30, 64]. The work stealing technique provides an efficient method to evenly balance the load among all the executing threads, as long as there is sufficient parallel computation expressed as tasks. Task parallel frameworks have grown in popularity and there are a number of task parallel frameworks, such as Cilk [29, 46, 52, 64, 99], Intel TBB [49, 139], OpenMP [17, 132], Habanero [20, 37], X10 [40], Java Fork/Join concurrency [96], and Task Parallel Library [98], that have become mainstream now. Section 2.1 in Chapter 2 provides a detailed overview of task parallelism and the mechanisms that enable task parallel programs to achieve load balancing.

Figure 1.1 (b) illustrates the execution of a task parallel program that takes an input array of integers and performs a computation on all the prime integers similar to thread-based program described above. In contrast to the thread-based program, the input array in the task parallel program can be split into a larger number of chunks than the number of processors on the machine. We then create as many tasks as the number of chunks with each task performing the computation on a single chunk of integers in the input array. We must take care that the chunks are large enough so that the computation in the tasks is sufficient to amortize the cost of creating the tasks. Since the program has more tasks than threads, the task runtime scheduler continuously assigns tasks to threads whenever threads become idle, thereby ensuring that the computation is sufficiently load balanced.

The above example highlights another important aspect of task parallel programs. The task runtime scheduler can balance the load as long as there are sufficient number of tasks that can be assigned to threads when they become idle. Therefore, like thread-based parallel programs task parallel programs must have sufficient parallel computation in order for the runtime to perform load balancing. Further, since tasks can execute in parallel, task parallel programs can also have performance issues due to the presence of secondary effects. Hence, despite the progress made in task parallel programs to address performance problems due to load imbalance, the programmers have to still deal with performance and scalability issues.

1.2 Inadequacy of Existing Performance Profilers

Profilers play a vital role in the performance debugging of parallel programs. When programmers observe that a program is not exhibiting the expected performance, they rely on profilers to identify performance bottlenecks in the program. An effective profiler must provide insights into the performance of the program and guide optimizations to improve the performance of the program.

A common approach to profiling programs is to measure various metrics that reflect the utilization of resources during execution and attribute the metrics back to the program source code. The intuition is that parts of the program that utilize the most amount of resources are the performance bottlenecks in the program. Hence, by measuring the resource utilization in various parts of the program, profilers can highlight the performance bottlenecks in the programs. For example, profilers can measure the time or the CPU clock cycles spent by each function during execution. The functions that execute for the most time or consume the most cycles are the bottlenecks in the program and the programmer has to focus on optimizing these functions. This technique is also known as “hot spot analysis”. There are numerous profilers that perform variants of hot spot analysis [10, 48, 50, 65, 72, 90, 150, 151, 158, 168].

Hot spot analysis is a useful technique to understand the performance of various parts of the program. However, the parts of the program that execute for the most amount of time may not be the parts that must be optimized to improve the performance of the program. For instance, consider a function that is highlighted by one of the profilers as taking the most time. If the function is not being executed on the critical path of the program, then optimizing the function to reduce the time it takes will not reduce the overall running time of the program. Hence, beyond techniques that identify parts of a program that take the most time, there is a need for techniques that can identify the parts of a program that must be optimized to increase the performance of the program.

To identify parts of a program that matter in increasing the performance, one approach is to identify the parts of the program that execute on the critical path. If the parts of the program which take the most time are also the parts that execute on the critical path, then optimizing the regions can improve the performance of the program. This idea forms the basis for numerous techniques that have used different ways to identify code that executes on the critical path [11, 35, 42, 82, 120, 134, 147, 148, 170]. Another approach to identify parts of a program that matter for increasing the performance, is to predict the improvement in performance on optimizing certain sections of code [47, 51, 144].

The techniques that either identify code executing on the critical path or predict performance improvements are based on a specific execution on a particular machine with a given number of threads. They do not account for the changes in performance and the critical path across different executions when parts of the program are optimized or when the program is executed at scale with a greater number of threads. Hence, the performance of the program may not improve even after optimizing regions highlighted by such techniques.

In summary, while prior profilers have made significant progress in identifying performance bottlenecks in parallel programs, they have the following drawbacks. Although existing profilers measure various metrics to identify parts of a program utilizing the most resources, they are not effective in identifying the parts of the program that matter in improving the performance. Further, most profilers highlight performance issues that manifest in a particular execution of a program. They do not indicate if a program can have scalable performance when executed on machines with greater number of processors.

1.3 Thesis Statement

This thesis develops a set of techniques that identify parts of the program that matter in improving the parallelism and the speedup of the program.

1.4 Our Contributions

This dissertation seeks to develop techniques that are necessary to understand the causes of low parallel computation and secondary effects that affect the performance of task parallel programs. In the process, we attempt to answer the following specific questions.

1. Does a program have sufficient parallel computation to enable scalable execution on machines with a larger core count?

2. Perhaps more importantly, what are the parts of a program that a programmer must focus on to increase the parallel computation in a program?

3. What parts of the program are experiencing secondary effects and why?

In the rest of this section, we describe our contributions to estimate the amount parallel computation, identify regions that matter for improving the parallel computation, and highlight regions experiencing performance degradation due to secondary effects. We discuss each technique in greater detail in subsequent chapters.

1.4.1 A Case for Measuring Logical Parallelism

We know from Amdahl's law [13] that the execution time and thereby, the speedup of a parallel program is limited by the maximum amount of computation that has to be executed serially in the program. Ideally, lesser amount of serial work [1] in a program will result in lower execution time of the program and greater possibility of the program exhibiting scalable speedup. Hence, if we can characterize the amount of serial work in the program, then we would be able to determine if a program has sufficient parallel work to achieve scalable speedup.

Logical parallelism is a metric that characterizes the amount of serial work in a program. *Logical parallelism* defines the speedup of the program in the limit, *i.e.*, the maximum speedup the program could ever attain given infinite processors, an ideal machine, and no runtime overhead. It is constrained by the critical path of the program, which is the longest (in terms of work) sequence of instructions that execute serially considering the series and parallel constructs expressed in the program. Logical parallelism is the ratio between the total work performed in the program and the work on the critical path of the program. Since logical parallelism defines the maximum speedup, it is an ideal metric to determine if a program will exhibit scalable speedup when executed on any machine with any number of processors. Specifically, for a program to have scalable speedup on a particular machine with a given number of processors, the logical parallelism must significantly exceed the number of processors on the machine [64, 146].

Computing the logical parallelism in a task parallel program requires a method to determine the total work in the program and the work on the critical path of the program. To determine the work on the critical path of the program, we need a way to measure work in the parts of the program that execute serially according to the series and parallel constructs expressed in the program. In any particular execution of a program, tasks that are supposed to execute in parallel according to the parallel constructs in the program can get mapped to the same thread and execute serially. Hence, we cannot measure the work on the critical path by merely tracking a program's execution and identifying the

[1]Work is the total time of a sequence of instructions. It can be also be expressed in terms of other metrics like, for *e.g.*, hardware execution cycles.

parts of the program that occur serially in the execution. Instead, we need a mechanism to determine the series and parallel relations expressed in the program.

The Dynamic Program Structure Tree (DPST) [138] is a structure that records the logical series and parallel relations expressed in a program from a given execution of the program. At a high level, the DPST represents the code fragments between task runtime constructs that create tasks (`spawn`) and synchronize tasks (`sync`). For example, a code fragment begins after the creation of a new task using the `spawn` statement and extends up to the subsequent `spawn` or `sync` statement. The DPST, then, captures the series or parallel relation between the code fragments. We can use the series and parallel relations encoded in the DPST to compute the logical parallelism of the program. Chapter 2 describes the DPST representation in detail.

In addition to determining the logical series-parallel relations, we need to record the work performed in the program to compute logical parallelism. We record the work by performing fine-grain measurement of the work in code fragments between task runtime constructs and attributing the measurements to the DPST, which represents the code fragments. The logical series-parallel relations in the DPST along with fine-grain work attributed to the DPST is a performance model of the program that enables us to compute the logical parallelism. We can compute the total work in the program by computing the aggregate of the fine-grain work of all the code fragments attributed to the performance model. At the same time, we can compute the work on the critical path by using the logical series-parallel relations and the fine-grain work to identify the code fragments that execute serially and perform the maximum work.

While the performance model enables us to compute the logical parallelism of a program, it cannot be efficiently stored in memory or on disk for long running applications. For such applications storing the entire performance model in memory and computing the logical parallelism over the entire performance model would not be feasible. A key contribution of this dissertation is an on-the-fly technique that computes the parallelism as the program executes by maintaining only parts of the performance model that represent the active tasks. Chapter 3 describes in extensive detail our technique to compute the parallelism using the entire performance model, as well as our technique to compute the parallelism on-the-fly.

As we were designing the parallelism computation technique, we realized that in addition to enabling parallelism computation, the performance model can be used to perform further analyses to identify performance issues in task parallel programs. We demonstrate the utility of the performance model by designing two novel analyses to identify parts a program that matter for increasing logical parallelism and addressing secondary effects. Next, we provide a brief overview of these analyses.

1.4.2 What-If Analyses to Identify Regions that Improve Parallelism

The parallelism computation is useful in identifying if the program has sufficient parallel work to achieve scalable speedup. Along with measuring the parallelism of the entire program, the parallelism computation technique measures the parallelism at various parts of the program. This enables the programmer to identify the parts of the program that have low parallelism. If a program has low speedup and inadequate parallelism, then the programmer can consider parallelizing the parts of the program that have low parallelism. However, parallelizing a part with low parallelism may not improve the parallelism or the speedup of the entire program, if that part of the program is not performing any significant work on the critical path of the program. Further, even if the part of the program is performing work on the critical path, the critical path itself may shift to a different path resulting in the parallelism and the speedup remaining almost the same. Hence, just measuring the parallelism of the program will not be sufficient to identify the parts of a program that matter for increasing the parallelism of the program.

Our performance model to compute parallelism represents the various code fragments in the program and has information about the work in each fragment and the series-parallel relations between fragments. Using the performance model, what if we can check if parallelizing a code fragment in the program increases the parallelism in the program? If the parallelism increases, it implies that actually implementing a parallelization strategy for the code fragment will increase the parallelism of the entire program. Using such a technique the programmer would be able to determine if a part of a program matters in increasing the parallelism of the program. We call this technique to estimate the improvement in parallelism as *what-if analyses*.

Our what-if analyses technique mimics the effect of parallelizing a code fragment by reducing the code fragment's contribution to the critical path work of the program. Our performance model stores enough information to enable what-if analyses. To perform what-if analyses we compute the parallelism of the program using the performance model by computing the total work and the critical path work. However, while computing the critical path work we reduce the work of the code fragment selected for what-if analyses and subsequently compute the critical path work. The parallelism computed from the reduced critical path work will reflect the effect of parallelizing the code fragment. Hence, the what-if analyses technique is useful in identifying if a region of code matters in improving the parallelism of the program.

Using what-if analyses, we design an iterative technique to automatically identify a set of regions that matter for increasing the parallelism of the program. In each step of the iterative process, we identify a code fragment that is performing the highest work on the critical path. Then, we perform what-if analyses to estimate the parallelism of the program on parallelizing the code fragment. We repeat this process until the estimated parallelism increases to a pre-defined level. The set of regions considered for what-if analyses in each iteration are the regions that can be parallelized to increase the parallelism of program. Chapter 4 describes our what-if analyses technique in greater detail and presents our algorithm to identify regions that matter in increasing the parallelism of a task parallel program.

1.4.3 Differential Analysis to Identify Secondary Effects

Even if a program has sufficient parallelism to achieve scalable speedup on a particular machine, it may not achieve the speedup due to the presence of secondary effects of parallel execution. It is hard to concretely detect such secondary effects without having incisive insights into the execution of the program on hardware. Our intuition is that a program that is experiencing performance degradation due to secondary effects shows an inflation in the work in the parallel execution compared to an oracle execution that is not experiencing secondary effects. If we observe the parallel execution showing an inflation in the work, then it is likely that the program is experiencing secondary effects.

To identify an inflation in the work, we need a mechanism to compare the work in the parallel execution with the work in an oracle execution. One approach is to measure the entire work in both the parallel and oracle executions and compare the work to identify the presence of secondary effects. While such an approach may indicate the presence of secondary effects in the program, it would not be sufficient to identify the parts of the program experiencing secondary effects. The performance model that we designed for computing parallelism measures fine-grain work performed in various code fragments in the program. If we perform a fine-grain comparison of the performance model for the parallel execution with the performance model of the oracle execution, the code fragments in the two models that are experiencing secondary effects will have higher work in the performance model of the parallel execution than in the performance model for the oracle execution. We call this analysis to compare the performance models to identify regions experiencing secondary effects as *differential analysis*.

Task parallel programs can experience secondary effects in any part of the program. However, not all secondary effects impact the speedup of the program. For instance, a region in a program may be experiencing significant secondary effects. But, if it is performing a small fraction of the total work or the work on the critical path, then it is likely that the secondary effects will not affect the speedup of the program. Hence, along with highlighting code fragments that are experiencing work inflation, our differential analysis technique also highlights code fragments that are having inflation in the critical path work and performing significant fraction of the total work and the critical path work. As a result, our differential analysis technique is able to highlight parts of the program that matter for addressing secondary effects. Chapter 5 describes our approach to perform differential analysis in detail.

1.4.4 Putting It Together in TaskProf

We have designed TaskProf, a profiler and an adviser for task parallel programs that implements all of the above techniques. As a profiler, TaskProf executes a task parallel program, constructs the performance model, and computes the logical parallelism using the performance model. From the parallelism information in the performance model, TaskProf generates a profile called the *parallelism profile*, that specifies the parallelism of the entire program and at various static code locations in the program. Along with the parallelism, TaskProf also measures the task runtime overhead in the program to identify sources of excessive parallelism (see Chapter 3), and reports it as a part of the parallelism profile. The programmer can use the parallelism profile to first, determine if a program has sufficient parallelism to achieve scalable speedup, and then identify parts of the program that have low parallelism or high task runtime overhead.

As an adviser, TaskProf reports to the programmer the program regions that matter for increasing the parallelism and addressing secondary effects. Using what-if analyses, TaskProf automatically identifies a set of static regions that matter in increasing parallelism of the program. The programmer can focus optimization efforts on these regions to improve the parallelism of a program. Alternatively, if the programmer has an intuition about parts of the program that can be parallelized, then TaskProf provides a method to check if parallelizing the identified parts matters in increasing the parallelism of the program. The programmer specifies a region that can be parallelized using annotations. TaskProf perform what-if analyses and generates a profile called the what-if profile, which specifies the parallelism after hypothetically parallelizing the annotated region.

TaskProf's adviser feature also includes the differential analysis technique to highlight regions experiencing secondary effects. To perform differential analysis, TaskProf first constructs the performance model from the parallel execution of a program. For the oracle execution, TaskProf uses the serial execution of the parallel program where tasks execute without any interference. Hence, it constructs the oracle performance model from the serial execution of the program. Then, TaskProf compares the two performance models and generates a profile called the *differential profile* that specifies the work inflation in the entire program and at various code regions in the program. The programmer can use the differential profile to identify regions experiencing secondary

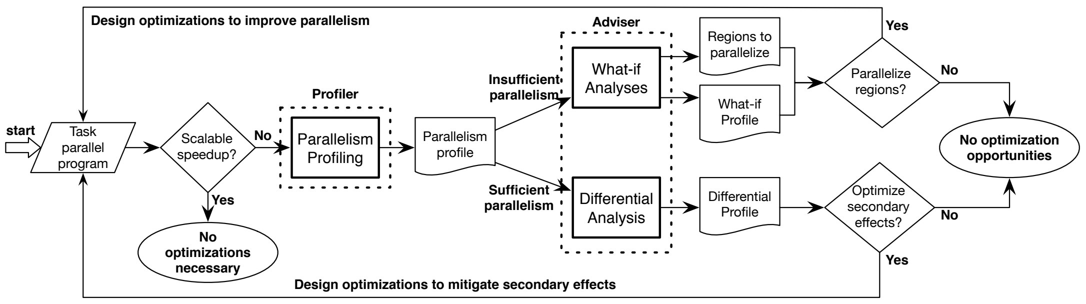


Figure 1.2: Illustration of a typical performance analysis workflow using TaskProf.

effects. Figure 1.2 shows a detailed workflow depicting the use of TaskProf's parallelism profiling, what-if analyses, and differential analysis in performance analysis and debugging of a task parallel program.

We have used our tool TaskProf to profile over twenty three task based applications. Using the performance insights gained from TaskProf, we have been able to design concrete optimizations and increase the performance of eleven applications. Our evaluation of TaskProf in Chapter 6 describes how TaskProf enabled us to identify performance bottlenecks in all these applications. Overall, the techniques we developed in TaskProf provide crucial insights into the performance of task parallel programs and guide optimizations to improve the performance.

1.5 Papers Related to this Dissertation

The ideas and techniques presented in this dissertation are drawn from the following published papers written in collaboration with my advisor Santosh Nagarakatte, Aarti Gupta, and Nader Boushehrinejadmoradi.

1. "A Fast Causal Profiler for Task Parallel Programs" [174], which introduces TaskProf and describes the technique to compute parallelism and perform what-if analyses.

2. "Parallelism-Centric What-If and Differential Analyses" [175], which extends the what-if analyses technique to automatically identify regions to parallelize and

presents the differential analysis technique to highlight program regions experiencing secondary effects.

3. "A Parallelism Profiler with What-If Analyses for OpenMP Programs" [33], which proposes an on-the-fly technique to compute the parallelism and perform what-if analyses.

4. "Atomicity Violation Checker for Task Parallel Programs" [173] and "Parallel Data Race Detection for Task Parallel Programs with Locks" [177], where we developed the initial technique to adapt the DPST for task parallel programs without serial semantics, albeit in the context of concurrency bug detection for task parallel programs.

1.6 Dissertation Organization

The rest of this dissertation is structured as follows. Chapter 2 provides a brief background on task parallelism and techniques to maintain series parallel relations in task parallel programs. Chapter 3 describes our technique to compute parallelism and assess task runtime overhead. Chapter 4 presents our what-if analyses technique and describes a method to automatically identify regions to parallelize using what-if analyses. Chapter 5 discusses our differential technique to highlight secondary effects. Chapter 6 presents the evaluation of TaskProf on a suite of benchmarks and provides insights on how TaskProf was used to identify optimization opportunities in the benchmarks. Chapter 7 discusses the prior related work in the context of this dissertation. Chapter 8 presents the conclusions and also discusses some potential topics to be explored in the future.

Chapter 2

Background

In this chapter, we provide a brief primer on task parallelism, with a particular emphasis on the logical series-parallel properties of task parallel programs. We then provide background on a previously proposed data structure for maintaining logical series-parallel properties, called the Dynamic Program Structure Tree [138], which forms the basis for all of the techniques that we propose in this dissertation.

2.1 Task Parallelism

Task parallelism is an effective abstraction for writing parallel programs. In a task parallel program, the programmer expresses the parallel work in the program in terms of light-weight structures known as *tasks*. As the task parallel program executes, a runtime system automatically creates threads and efficiently maps the tasks that are created in the program to the threads.

Task parallelism simplifies the development of parallel programs. In traditional thread-based parallel programming, the programmer creates threads that can execute in parallel and divides the parallel work in the program evenly among all the threads. To avoid the overhead from the creation and the context-switching of excessive threads, programmers typically create as many threads as the number of processors on the execution machine. Now, if the thread-based program has to be executed on a different machine with a different number of processors, then to obtain scalable performance the programmer will again have to divide the parallel work based on the number of processors available on that machine. Hence, developing thread-based parallel programs that are performance portable is challenging. In task parallel programs, the programmer does not manually divide the parallel work evenly among the threads. The programmer can express all the parallel work in the program in terms of tasks. The task runtime will take care of creating as many threads as the number of processors on the execution machine and mapping the tasks to the threads for execution. Hence, task parallelism provides the promise of performance portability in parallel programs.

In addition to performance portability, task parallelism simplifies parallel programming by automatically balancing the load in parallel computations. The primary challenge in developing parallel programs using threads is to divide the parallel work evenly among the threads to ensure that the computation is sufficiently load balanced. In contrast to threads, tasks are light-weight structures. Once a task is mapped to a thread, the task is executed to completion on the thread. Since tasks cannot be preempted, the task runtime does not need to maintain per-task resources like a stack and register state. As a result, creating tasks has significantly lesser overhead than creating threads. A task parallel program can create more tasks than the number of processors on the execution machine. Whenever a thread becomes idle, the task runtime assigns a task to the thread and thereby ensures that the computation is sufficiently load balanced. Task runtimes usually use an effective scheduling technique known as work stealing to efficiently map tasks to idle threads [15, 22, 29, 30, 64]. Hence, along with the promise of performance portability task parallelism provides the promise of automatic load balancing of the parallel computation.

Parallel programming frameworks provide task parallelism as programming language extensions (for *e.g.*, Cilk [64], Habanero Java [37]), compiler directives (for *e.g.*, OpenMP [17, 132]) or library functions (for *e.g.*, Intel TBB [49, 139], Microsoft Task Parallel Library [98]). In the task programming model, tasks are created and managed using a small, yet succinct set of constructs (functions in library based frameworks). To create a new task, the task programming model provides the *spawn* construct (or the *spawn* function in library based frameworks like Intel TBB). Semantically, spawning a new task specifies that the code immediately following a *spawn* construct and extending up to a *sync* construct can execute in parallel with the newly spawned task.

Figure 2.1 shows a simple example of a task parallel program that uses the *spawn* construct to create tasks. The program spawns two tasks - one task that executes the function B in line 3 and another that executes the function D in line 5. According to the semantics, the function B can execute in parallel with the code immediately following the creation of the task until the next *sync* which includes the functions C, D, and E. Similarly, the function D can execute in parallel with the function E which is part of the code that follows the spawning of function D.

```
1 void main() {
2    A();
3    spawn B();
4    C();
5    spawn D();
6    E();
7    sync;
8    F();
9 }
```

Figure 2.1: A simple example of a task parallel program that creates tasks and waits for tasks to complete using `spawn` and `sync` constructs, respectively.

The task programming model also provides the *sync* construct to synchronize tasks that may have dependencies. The semantics of the *sync* construct specifies that a task executing a *sync* construct will wait for all its child tasks that were spawned before the *sync* to complete execution. The semantics of the *sync* construct implies that the code before the *sync* construct has to execute in series with the code following the *sync* construct. In the simple task parallel program shown in Figure 2.1, the *sync* construct occurs in line 7. According to the semantics, the functions A-E, which occur before the *sync* construct, have to execute in series with the function F which executes after the *sync* construct.

Apart from the *spawn* and *sync* constructs, many task parallel frameworks provide high-level parallelization patterns like, for *e.g.*, parallel loops (`parallel_for` in Intel TBB and `cilk_for` in Cilk) and parallel reductions (`parallel_reduce` in Intel TBB). Such generic parallel algorithms can be implemented by parallel divide-and-conquer recursion over the loop iterations using *spawn* and *sync* constructs.

The *spawn* construct defines a parallel relation between the newly spawned task and the code following the *spawn* construct until the next *sync* construct. However, in any particular execution of the task parallel program, the newly spawned task and the code following the *spawn* can be mapped to the same thread by the runtime, and thereby execute in series. For instance, in an execution of the program in Figure 2.1 the functions B and C can get mapped to the same thread and execute in series even though the *spawn* construct that creates a task to execute the function B defines a parallel relation between the functions B and C. To compute the logical parallelism, which is the speedup of a program in the limit, we need to consider the series and parallel properties expressed in the program, rather than the series and parallel relations observed in any given execution. There are numerous representations that can capture the series and parallel properties of the program from a dynamic execution of the program for a given input, like SP-parse tree [63], SP-bags [63], ESP-bags [137], SP-Order [21], WSP-Order [161], Offset-Span Labeling [73, 117], OpenMP series-parallel graph [33], and Dynamic Program Structure Tree [138]. We use the Dynamic Program Structure Tree representation, which can be constructed in parallel, thereby enabling us to profile a program while executing it in parallel.

2.2 Dynamic Program Structure Tree

The Dynamic Program Structure Tree (DPST) [136, 138] encodes the logical series and parallel relations expressed in a task parallel program. We use it as a part of the performance model that we have designed to compute the logical parallelism (see Chapter 3) and perform what-if (see Chapter 4) and differential analyses (see Chapter 5). The DPST was originally proposed for explicit async-finish task parallel programs. We have adapted it for task parallel programs without explicit finish constructs.

The DPST is an ordered rooted tree that encodes the logical series and parallel relations in a dynamic execution of a task parallel program for a given input. The nodes of a DPST can be of three types: (1) *step*, (2) *async*, (3) *finish* nodes. Each of these nodes represent some part of the dynamic execution. The edges in the DPST represent the parent-child relations between the nodes. Figure 2.2 (d) shows the complete DPST constructed from a dynamic execution of the program in Figure 2.1.

Step Nodes

A step node in a DPST represents the maximal sequence of dynamic instructions that does not include a task runtime construct (like a *spawn* or a *sync* construct). This implies that step nodes represent fragments of the dynamic execution that begin immediately after a task runtime construct (for *e.g.*, *spawn* and *sync*) and end at the last instruction before another task runtime construct. Similarly, step nodes can also represent fragments that begin at the first instruction of a newly created task and end at the last instruction of the task if the task does not include task runtime constructs. The step nodes are always the leaf nodes of the DPST. In Figure 2.2 (d), the nodes S0-S5 are step nodes. These nodes represent the code fragment in each of the functions A-F, respectively (S0 represents the fragment in A, S1 represents the fragment in B, and so on) in the example program in Figure 2.1. We assume that functions A-F do not contain *spawn* and *sync* constructs.

Async Nodes

An async node represents the creation of a new task using the *spawn* construct. An async node captures the parallel relation between the newly spawned task and the code immediately following the *spawn* construct extending up to the subsequent *sync* construct. The execution within a newly spawned task is represented by nodes in the sub-tree under the corresponding async node. The execution of the code following the *spawn* construct extending up to the subsequent *sync* construct is represented by the sibling nodes that are to the right [1] of the async node and their descendants. Hence, async nodes have the property that nodes in the sub-tree under an async node will be in parallel with sibling nodes that are to the right of the async node and their descendants. Every dynamic instance of a *spawn* construct is represented by an async node in the DPST. The async nodes are always part of the non-leaf internal nodes in the DPST. In Figure 2.2 (d), the nodes A0 and A1 are the async nodes in the DPST, which represent the *spawn* constructs in line 3 and line 5, respectively, in the example program in Figure 2.1.

[1]A Node R is to the *right* of a node L if R occurs after L in the depth-first traversal of the DPST.

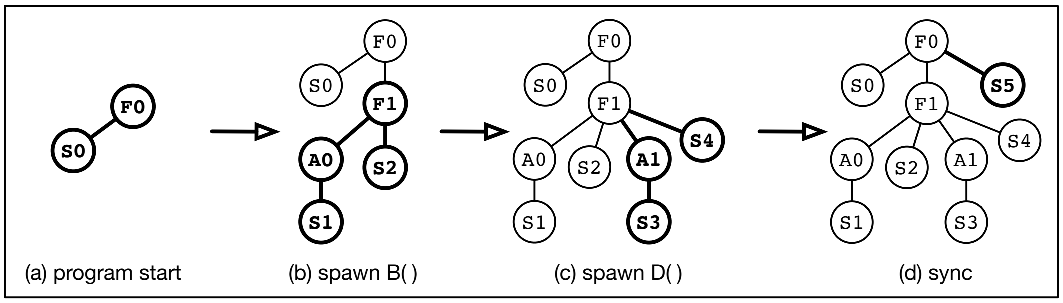


Figure 2.2: Illustration of the DPST construction for a trace of the program in Figure 2.1 with the following sequence - (a) program start, (b) the first spawn statement: `spawn B()`, (c) a subsequent spawn: `spawn D()`, and (d) a `sync` statement. The nodes that are newly added in each step are highlighted in bold.

Finish Nodes

Unlike step and async nodes, finish nodes do not explicitly represent a concrete fragment in the dynamic execution. Instead, finish nodes capture the series relation introduced by the *sync* construct. Specifically, a finish node captures the series relation between the code executed before the *sync* construct and the code executed after the *sync* construct. The code executed before the *sync* construct starting from the first *spawn* after a previous *sync* is represented by the nodes in the sub-tree under the finish node. The execution after the *sync* construct is represented by the sibling nodes to the right of the finish node and their descendants. Hence, similar to async nodes, the finish nodes have the property that nodes in the sub-tree under a finish node will be in series with sibling nodes that are to the right of the finish node and their descendants. The finish nodes are also a part of the non-leaf internal nodes of the DPST. In Figure 2.2 (d) the nodes F0 and F1 are the finish nodes in the DPST.

2.2.1 DPST Construction

In this section we describe a high-level procedure to construct the DPST from an execution of a task parallel program. Nodes are added to the DPST when a program begins execution and when the task runtime constructs, *spawn* and *sync*, are executed.

Program Start

When a program begins execution, a finish node is created as the root node of the DPST, since the program has to wait for all tasks that are created in the program to complete before it completes. A step node is then added as the child of the root finish node to represent the starting computation in the program before the first task runtime construct is encountered.

Figure 2.2 shows the nodes that are added to the DPST as instructions are encountered in the execution trace of the program in Figure 2.1. Figure 2.2 (a) shows the addition of the root finish node and a step node as a child of the root node to the DPST when the program begins execution.

Task Spawn

To capture the series property introduced by a *sync* construct, all the nodes representing the code starting from the first *spawn* after the previous *sync* are enclosed in the sub-tree under the finish node. However, when a program executes the first *spawn* construct, we would not know if there will be a future *sync* construct in the execution trace. Hence, we need to anticipate that the task may execute a *sync* construct.

When the first task is spawned, a finish node is added anticipating that a *sync* construct would be executed in the future. The finish node is added as a child node of the node corresponding to the task that executed the *spawn* construct. A finish node needs to be added not only on the first *spawn* construct of the program, but also for the first *spawn* after every *sync* construct. On subsequent *spawn* constructs the finish node is not added since the first *spawn* would have already anticipated the *sync* and added the finish node.

To represent the newly created task, an async node is added as the child of the of the node corresponding to the task that executed the *spawn* construct. Then, a step node is added as the child of the async node to represent the initial computation within the newly spawned task. Finally, a step node is added as a sibling to the async node to represent the computation after the *spawn* construct.

Figure 2.2 (b) shows the additions to the DPST after the first *spawn* construct in line 3 of the example program in Figure 2.1 is encountered. A finish node, F1, is added as the child of the node F0 since it is the first *spawn* construct. The async node A0 is then added as the child of the finish node F1 to represent the newly spawned task. Step node S1 is added as a child of the async node A0 to represent the computation within the task. Step node S2 is added as a sibling of A0 and represents the computation after the *spawn* construct.

Figure 2.2 (c) shows the additions to the DPST after the *spawn* construct in line 5 of the example program in Figure 2.1 is encountered. The key difference here is that the *spawn* at line 5 is not the first *spawn*. Hence, the async node representing the *spawn*, A1, can be directly added as the child of F1. The step nodes S3 and S4 are added as child and right sibling of async node A1 to represent the computation within the newly spawned task and the computation after the *spawn* construct, respectively.

Task Sync

On a *sync* construct, the DPST should already contain the finish node that encloses all the nodes that represent the code before the *sync* construct. Hence, the only addition to the DPST on a *sync* construct is a step node as the right sibling of the finish node that is added to represent the series property of a *sync* construct. The newly added step node represents the computation after the *sync* construct.

Figure 2.2 (d) shows the addition to the DPST after the *sync* construct in line 7 of the example program in Figure 2.1 is encountered. A step node, S5, is added as the right sibling of the finish F1 to represent the computation after the *sync* construct.

2.2.2 Properties of the DPST

In this section, we highlight the key properties of the DPST that enable us to design efficient techniques to compute logical parallelism and perform what-if and differential analyses. The properties of the DPST are;

- The parent-child relations between nodes in a DPST do not change once set during node creation.
- All the leaf nodes of a DPST are step nodes. All the non-leaf internal nodes are either async or finish nodes. The root node of a DPST is a finish node.
- The children of a node in the DPST are ordered left-to-right to reflect the serial execution of various fragments of the task represented by that node.
- All nodes in the sub-tree under an async node can logically execute in parallel with all sibling nodes to the right of the async node and their descendants.
- All nodes in the sub-tree under a finish node are in series with all sibling nodes to the right of the finish node and their descendants.
- Two step nodes can logically execute in parallel if the child of the least common ancestor of the step nodes, which is also an ancestor of the left-most step node, is an async node.
- The DPST and the series-parallel relations encoded in the DPST do not change across different schedules of a program for a given input provided the program does not contain races that cause non-determinism in the creation of tasks.

Let us illustrate some of the properties of the DPST using the DPST in Figure 2.2 (d). The leaf nodes in the DPST comprise of only the step nodes S0-S5. The internal nodes comprise of the async nodes A0 and A1 and finish nodes F0 and F1. Finish node F0 is the root node of the DPST. To illustrate the parallel property of an async node, consider for instance async node A0. According to the property of the async node, step node S1 which is in the sub-tree under A0 will execute in parallel with step nodes S2, S3, and S4 which are the right siblings and descendants of the right siblings of A0. Similarly, to illustrate the series property of a finish node, consider for instance the finish node F1. According to the property, the nodes in the sub-tree under F1, which includes step nodes S1-S4 execute in series with step node S5, which is a right sibling of F1.

We can clearly identify if two step nodes logically execute in parallel using the DPST. Step nodes S1 and S3 in Figure 2.2 (d) logically execute in parallel since their least common ancestor is F1 and the immediate child of F1 on the path to S1, which is the left-most among S1 and S3, is an async node (*i.e.*, A0). In contrast, nodes S2 and S3 in Figure 2.2 (d) execute serially because the immediate child of their least common ancestor (*i.e.*, F1) on the path to S2 (left-most among S2 and S3) is not an async node.

Since the parent-child relations do not change in the DPST, it enables us to construct the DPST in parallel as the program executes. The series and parallel properties represented by the async and the finish nodes are key to computing the parallelism of the program and performing what-if analyses. Since the DPST does not change across different executions of the same input we have been able to use it determine if a program has adequate parallelism for a machine with a given number of processors even while profiling the program on a different machine with a different number of processors. In the subsequent chapters, we make use of the DPST to design novel analyses to identify performance and scalability issues in task parallel programs.

Chapter 3

Parallelism Profiling

This chapter describes a profiling technique to measure logical parallelism and task runtime overhead in task parallel programs. Parallelism and task runtime overhead are common factors that influence the speedup of task parallel programs. Parallelism quantifies the amount of parallel work in a dynamic execution of a program for a given input and is key to determining if a program has sufficient parallel work to achieve scalable speedup. Task runtime overhead captures the overhead from creating parallel work and is useful in determining if the parallel work in a program is too fine-grain to be worth executing in parallel. Hence, the goals of our parallelism profiling technique are to (1) evaluate if the program has adequate parallel work to obtain speedup on a machine, (2) assess if the program can scale to machines with larger processor count, and (3) identify parts of a program that have low parallelism and high runtime overhead.

3.1 Motivation and Overview

Parallel programs can exhibit scalable speedup when they have adequate parallel work and the parallel work is distributed evenly among the threads. Task parallelism ensures the second aspect by using an efficient work-stealing scheduler to distribute the parallel work equally among the executing threads. However, task parallel programs must still have adequate parallel work that can be distributed to achieve scalable speedup.

When a program does not have adequate parallel work, it leads to load imbalance and introduces serialization in parts of the execution. Excessive serialization reduces the speedup of the program since the speedup is limited by the serial sections in a program (according to Amdahl's law [13]). Load imbalance causes threads to wait on one or more parallel threads to complete execution, thereby increasing the overall execution time of the program. Furthermore, a program may not experience load imbalance or serialization in an execution with a particular thread count. But, it may experience load imbalance and serialization when executed with a larger thread count, thus inhibiting the scalability of the program.

While a task parallel program must have adequate parallel work, the parallel work must be sufficiently coarse-grain. If the parallel work is too fine-grain, then the overhead from the task runtime to create the parallel work will become significant with respect to the work itself. The program will spend a major portion of the execution creating the parallel work rather than executing useful computation, which can affect the speedup of the program. Hence, for a task parallel program to have scalable speedup and optimal performance, it requires a careful balance of adequate parallel work and low task runtime overhead.

3.1.1 Logical Parallelism and Task Runtime Overhead

To help programmers quickly analyze programs that do not have scalable speedup, we need a mechanism to assess the parallel work and the task runtime overhead in task parallel programs.

Assessing Parallel Work

Intuitively, a program can expect to have scalable speedup if there is adequate parallel computation expressed in the program. Quantifying the parallel computation expressed in the program can provide a way to assess if a program can have scalable speedup. One possible way to quantify the parallel computation is if we can determine the maximum possible speedup that the program can have considering the parallel computation expressed in the program. If the maximum possible speedup is not adequate enough, then the program does not have sufficient parallel computation to exhibit scalable speedup.

Logical parallelism is a metric that quantifies the speedup of the program in the limit. It defines the maximum speedup the program could possibly attain from the parallel computation expressed in the program. Logical parallelism is constrained by the critical path of the program, which is the longest (in terms of work) sequence of code that

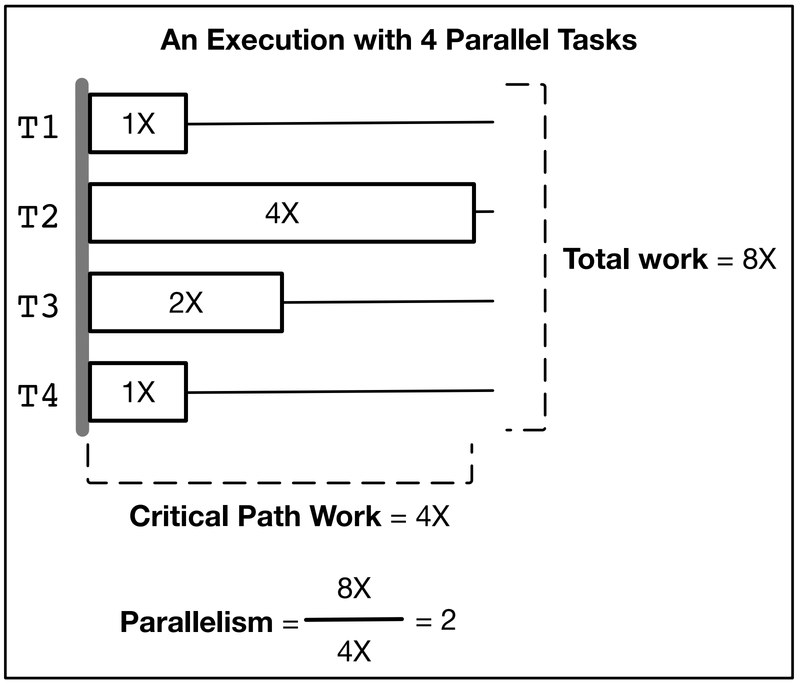

An Execution with 4 Parallel Tasks
T1
1X
T2
4X
T3
2X
T4
1X
Total work = 8X
Critical Path Work = 4X
Parallelism = 8X / 4X = 2

Figure 3.1: Illustration of the parallelism computation for a program with 4 tasks, denoted T1, T2, T3, and T4. The tasks execute in parallel according to the parallel constructs expressed in the program. The work performed in each task is shown inside the rectangular boxes denoting the tasks.

executes serially considering the series and parallel constructs expressed in the program. It can be computed by calculating the ratio between the total work and work on the critical path of the program.

We illustrate the parallelism computation using a simple example containing four tasks shown in Figure 3.1. The four tasks execute in parallel according to the parallel constructs specified in the program. The parallelism can be computed by aggregating the total work and the work on the critical path of the four tasks. The total work is the sum of the work from the four tasks. Since all the four tasks execute in parallel, the work on the critical path is work performed by the task that performs the highest work, which is work on task T2.

Since logical parallelism quantifies the maximum speedup in a program, it is an ideal metric for assessing if a program has adequate parallel work to exhibit scalable speedup. Specifically, for a program to have scalable speedup with an execution on a given number of processors, the parallelism of the program should be significantly higher than the number of processors to account for overhead from other sources like the task runtime and parallel execution on hardware. At the same time, if a program has parallelism that is less than number of processors used for the execution, then the program will not have scalable speedup.

Another key aspect of logical parallelism is that it can be used to evaluate if a program has adequate parallel work to achieve scalable speedup on any number of processors. Since logical parallelism defines the speedup in the limit, it is a property of the program for a given input. Logical parallelism remains the same for a given input irrespective of the number of processors used to execute the program or how tasks get mapped to threads in any particular execution. This enables us to compute the parallelism from a particular execution, and use the parallelism to evaluate if the program can achieve scalable speedup for the same input on a completely different execution with a different processor count.

Assessing Task Runtime Overhead

A task parallel program incurs overhead from the task runtime when it creates new tasks and when the runtime scheduler assigns tasks to threads. Our goal is to identify if the parallel work in the program is too fine-grain with respect to the work performed to create the parallel work. Hence in assessing if the overhead from the task runtime is high, we only consider the overhead from creating tasks and not the overhead when the runtime scheduler assign tasks to threads.

A significantly high logical parallelism in a program indicates that the program is creating many tasks. However, if the program is performing sufficient work in each of the parallel tasks, then the overhead from the creation of the tasks is not significant with respect to the parallel work in the program. Therefore, a significantly high parallelism alone does not indicate that the overhead from the task runtime is also high. Instead, by comparing the work performed in creating tasks with the work performed in the tasks, we can identify if the overhead from the task runtime is dominating the execution.

Hence, we express *task runtime overhead* as the ratio between the total task creation work and the total work in the program.

3.1.2 Overview of our Approach

We propose a technique to compute the logical parallelism and the task runtime overhead in task parallel programs. To compute the logical parallelism we need a mechanism to measure the total work performed in the program and the work on the critical path of the program considering the series and parallel constructs expressed in the program.

An execution of a task parallel program can be visualized as a set of code fragments, which are separated by task runtime constructs like, *spawn* or *sync*. According to the task runtime constructs expressed in the program, some of the code fragments execute in series and other code fragments can execute in parallel. In any particular execution of a program, code fragments that are supposed to execute in parallel according to the task runtime constructs in the program can get mapped to the same thread and execute serially. However, to measure the logical parallelism of the program we need to determine the logical series and parallel relations expressed in the program and not series and parallel relations that actually manifest in a given execution. In addition to determining the logical series-parallel relations, we need to record the work performed in the program and the work performed in the task runtime when tasks are created to compute logical parallelism and the task runtime overhead.

The Dynamic Program Structure Tree (DPST) [138] that we described in detail in Section 2.2, encodes the logical series-parallel relations in a program from a given execution of the program. The step nodes in the DPST represent the parts of the program where computation is performed. The async nodes in the DPST represent locations in the program where a task is spawned. Hence, our technique constructs the DPST and performs fine-grain measurement of work at each step node and the cost incurred to create tasks at each async node. The DPST, enriched with fine-grain measurements of computation and task creation work, is a performance model of the program for computing the logical parallelism and the task runtime overhead in the program. Figure 3.2 (b) shows a sample performance model. Beyond parallelism computation, the performance model has enabled novel analyses to identify parts of a program that matter for parallelization and pinpoint regions experiencing secondary effects. We discuss the two analyses in Chapters 4 and 5, respectively.

We have implemented a concrete instantiation of our technique to measure parallelism and task runtime overhead in TaskProf, a profiler for task parallel programs. TaskProf executes the input task parallel program, constructs the performance model, and performs the parallelism analysis using the performance model to compute the parallelism and the task runtime overhead in the entire program. In addition to computing the parallelism and task runtime overhead of the entire program, our goal is to identify parts of the program that have low parallelism or high task runtime overhead. Hence, TaskProf attributes the parallelism and runtime overhead from the performance model to the program source code and provides feedback to the programmer in the form of a profile called the *parallelism profile*. The parallelism profile specifies the parallelism of the entire program and parallelism at each spawn site[1] in the program. The parallelism profile quantifies the total task runtime overhead as a percentage of the total work in the program. In addition to the total task runtime overhead, TaskProf attributes the task runtime overhead to the corresponding spawn sites in the program. Along with the parallelism and the task runtime overhead, TaskProf attributes the work on the critical path of the program to the corresponding spawn sites. Figure 3.2 (c) shows a sample parallelism profile. The programmer can use the parallelism profile to identify spawn sites in the program that have low parallelism or high task runtime overhead and perform significant work on the critical path.

3.1.3 Contributions

We make the following contributions in assessing the logical parallelism and the task runtime overhead in task parallel programs.

[1] A spawn site is a location in the program source code where a task is created.

1. We propose a novel performance model for measuring logical parallelism that records the series and parallel relations in the program and fine-grain work performed in the program and in the creation of tasks (see Section 3.2).

2. We design a technique that uses the performance model to compute the logical parallelism and the task runtime overhead (see Section 3.3).

3. We develop an on-the-fly technique to construct the performance model and compute the logical parallelism during program execution (see Section 3.3.2).

4. We design a technique to attribute the parallelism and the task runtime overhead from the performance model to the program source code that enables programmers to assess the parallelism and the task runtime overhead in the entire program and at various parts of the program (see Section 3.4).

3.2 Performance Model

TaskProf measures the logical parallelism and the task runtime overhead of a task parallel program by constructing a performance model that encodes the logical series-parallel relations in the program and the fine-grain work performed in the execution of the program. In this section, we describe the contents of performance model in detail and present our approach to construct the performance model from an execution of the program.

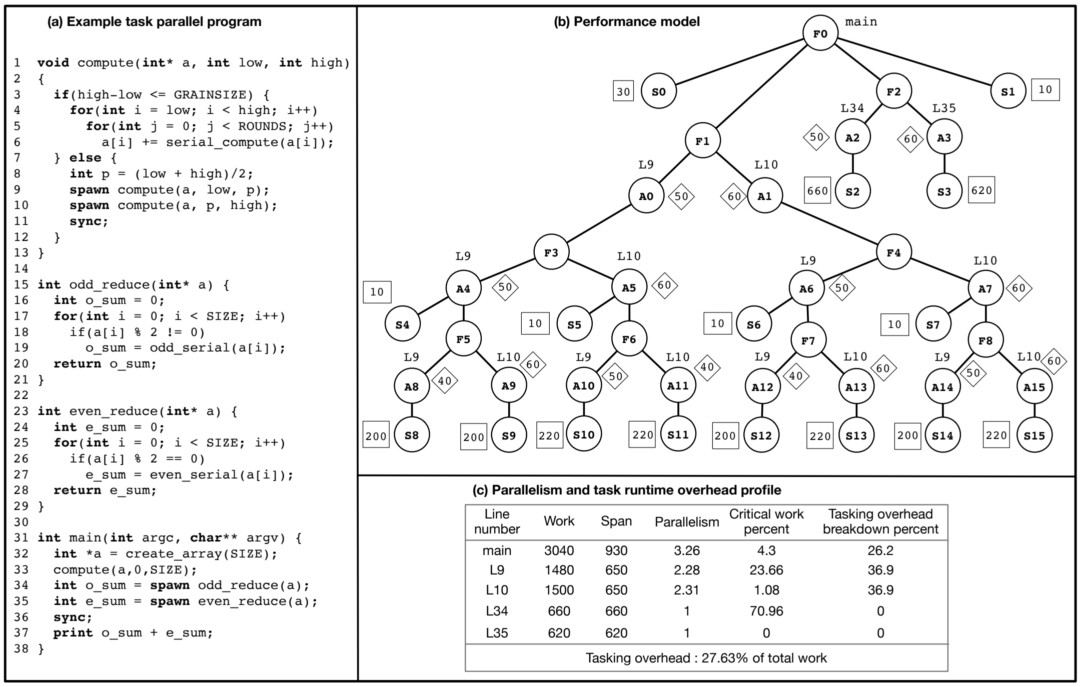
To measure the logical parallelism, TaskProf has to measure the work in the parts of the program that execute in series according to the series and parallel constructs expressed in the program. The Dynamic Program Structure Tree (DPST) is a data structure that captures the logical series-parallel relations that have been expressed in the program from a given execution of the program. The step nodes in the DPST represent the code fragments between task runtime constructs. The async and finish nodes in the DPST capture the logical series-parallel relations between the step nodes. The logical series-parallel relations encoded in the DPST are specific to a given input, and do not change across different executions with the same input, irrespective of the number of processors used to execute the program or the mapping of tasks to threads [2]. TaskProf uses the logical series-parallel relations encoded in the DPST to identify the parts of the program that execute in series according to the series and parallel constructs expressed in the program.

In addition to determining the series-parallel relations, to compute the parallelism TaskProf has to measure the total work performed in the program and the work on the critical path, which consists of code fragments that logically execute in series and perform the highest work. TaskProf performs fine-grain measurement of work in the sequence of instructions represented by step nodes in the DPST, and attributes the measured work to the corresponding step nodes. By performing fine-grain measurement of work in the step nodes and maintaining logical series-parallel relations between the step nodes, TaskProf measures the total work in the program as well as the work on the critical path. To measure the total work, TaskProf computes the aggregate of the fine-grain work of all the step nodes in the DPST. Similarly, to compute the critical path work, TaskProf identifies the logically serial set of step nodes that perform the highest aggregate of work among all such logically serial sets of step nodes.

To pinpoint sources of high task runtime overhead, TaskProf has to compare the work performed to create tasks with the total work performed in the tasks. The step nodes of the DPST contain fine-grain work information which is sufficient to compute the total work performed in the tasks. Since the async nodes in the DPST represent locations in the program where a task is spawned, TaskProf measures the work performed in the runtime to create a task and attributes the task creation work to the corresponding async nodes in the DPST. Using the task creation work in the async nodes and the work in the step nodes, TaskProf computes the task runtime overhead in the program.

The series-parallel relationships encoded as a DPST along with the fine-grain measurement of work in the step nodes and task creation work in the async nodes is a performance model that enables the computation of logical parallelism and task runtime overhead. Figure 3.2 (b) illustrates the performance model of the program in Figure 3.2 (a). It

[2]Series-parallel relations remain the same for a given input only under the assumption that there are no races in the program that cause non-determinism in the creation of certain tasks.



(a) Example task parallel program

```
1  void compute(int* a, int low, int high)
2  {
3    if(high-low <= GRAINSIZE) {
4      for(int i = low; i < high; i++)
5        for(int j = 0; j < ROUNDS; j++)
6          a[i] += serial_compute(a[i]);
7    } else {
8      int p = (low + high)/2;
9      spawn compute(a, low, p);
10     spawn compute(a, p, high);
11     sync;
12   }
13 }
14
15 int odd_reduce(int* a) {
16   int o_sum = 0;
17   for(int i = 0; i < SIZE; i++)
18     if(a[i] % 2 != 0)
19       o_sum = odd_serial(a[i]);
20   return o_sum;
21 }
22
23 int even_reduce(int* a) {
24   int e_sum = 0;
25   for(int i = 0; i < SIZE; i++)
26     if(a[i] % 2 == 0)
27       e_sum = even_serial(a[i]);
28   return e_sum;
29 }
30
31 int main(int argc, char** argv) {
32   int *a = create_array(SIZE);
33   compute(a,0,SIZE);
34   int o_sum = spawn odd_reduce(a);
35   int e_sum = spawn even_reduce(a);
36   sync;
37   print o_sum + e_sum;
38 }
```

(c) Parallelism and task runtime overhead profile

Line number	Work	Span	Parallelism	Critical work percent	Tasking overhead breakdown percent
main	3040	930	3.26	4.3	26.2
L9	1480	650	2.28	23.66	36.9
L10	1500	650	2.31	1.08	36.9
L34	660	660	1	70.96	0
L35	620	620	1	0	0
Tasking overhead : 27.63% of total work					

Figure 3.2: (a) An example task parallel program. (b) The performance model for an execution of the program in (a). The numbers in the rectangular and the diamond boxes represent the work performed in the step nodes and in the runtime to create tasks in the async nodes, respectively. (c) Parallelism profile reported by TaskProf.

shows the series-parallel relations encoded as the async, finish and step nodes of the DPST. Nodes A0-A15 are the async nodes, F0-F8 are the finish nodes, and S0-S15 are the step nodes in the DPST. Each step node shows the amount of work attributed to it. Each async node shows the work performed to create the task spawned at the async node.

In summary, TaskProf's performance model to compute logical parallelism and task runtime overhead consists of three components: (1) the series-parallel relationships encoded as the DPST, (2) the fine-grain measurement of computation at each step node, and (3) the fine-grain measurement of task creation work at each async node. In the rest of this section, we will describe our approach to construct the performance model from an execution of a task parallel program.

3.2.1 DPST Construction

To construct the performance model as a task parallel program executes, we need to construct the DPST along with measuring the work performed in regions of the program represented by the step nodes. We need to ensure the work measurements are not skewed by the construction and maintenance in memory of the DPST. Hence our goal is to construct the DPST without perturbing the program execution. This poses two key challenges - (1) The DPST for an entire execution can be significantly large. The DPST construction approach should not maintain the entire DPST to ensure a small memory footprint that does not perturb the execution. (2) Frequent synchronization to access the DPST nodes can perturb and slow down the program execution. Hence, the DPST construction approach should minimize synchronization while adding and removing nodes from the DPST.

TaskProf executes in the context of the program and constructs the DPST by intercepting calls to the task runtime. We address the challenge of perturbation due to frequent synchronization by making use of the properties of task parallel programs and the DPST. In a task parallel program, once a task is assigned to execute on a particular thread, it executes to completion on the same thread and cannot be preempted. This implies that the corresponding nodes in the DPST will not be accessed by multiple threads. In addition, the parent child relations will not change once they are set in the DPST. Using these properties TaskProf avoids synchronization and maintains the DPST nodes in a per-thread data structure. Further, in a task parallel program execution, a task does not complete execution until all the descendant tasks complete. In the context of the DPST, a node corresponding to a task will complete only after all of the nodes in its sub-tree complete. Hence, TaskProf maintains the nodes of the DPST in per-thread stacks. TaskProf creates a stack for each thread that is created by the runtime. Each per-thread stack maintains the nodes in the DPST that correspond to the tasks executed by the thread. The nodes of the DPST are pushed to and popped from the per-thread stacks on program start, execution of a spawn construct, execution of a sync construct, task begin, and task completion. Maintaining the nodes in the per-thread stacks ensures a small memory footprint since at any point in the execution the stacks contain only the nodes that represent the active tasks.

Algorithm 1: Updates to the DPST when a program begins execution.

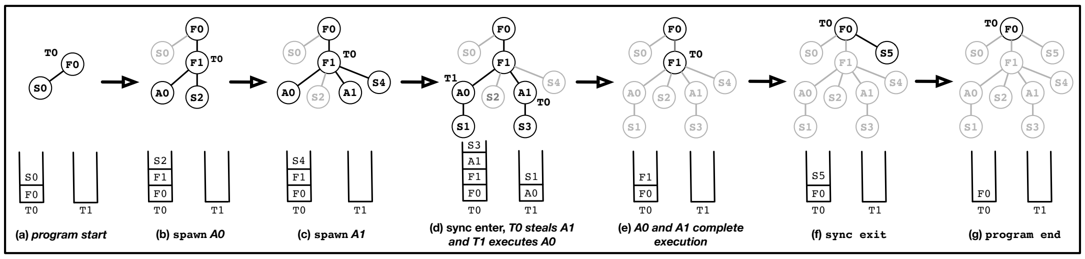
```
1  procedure OnProgramStart(tid, Stacks)
2      $F \leftarrow CreateNode(FINISH)$
3      $Stacks[tid].Push(F)$
4      $S \leftarrow CreateNode(STEP)$
5      $S.parent \leftarrow F$
6      $Stacks[tid].Push(S)$
```

Program Start

Algorithm 1 shows the updates to the DPST when the program begins execution. TaskProf first creates a finish node as the root of the DPST and adds it to the executing thread's stack. TaskProf then creates a step node as the child of the root finish node to represent the fragment of execution before the first task spawn. The step node is also added to the executing thread's stack. Figure 3.3(a) illustrates the updates to the DPST and the per-thread stacks when a program begins execution.

Algorithm 2: Updates to the DPST when a task is spawned.

```
1  procedure OnTaskSpawn(tid, Stacks)
2      $Stacks[tid].Pop()$                          ▷ Pop Step node
3      $P \leftarrow Stacks[tid].Top()$
4      if $first\_spawn$ then
5          $F \leftarrow CreateNode(FINISH)$
6          $F.parent \leftarrow P$
7          $Stacks[tid].Push(F)$
8          $P \leftarrow F$
9      $A \leftarrow CreateNode(ASYNC)$
10     $A.parent \leftarrow P$
11     $S \leftarrow CreateNode(STEP)$
12     $S.parent \leftarrow P$
13     $Stacks[tid].Push(S)$
```



(a) program start
(b) spawn A0
(c) spawn A1
(d) sync enter, T0 steals A1 and T1 executes A0
(e) A0 and A1 complete execution
(f) sync exit
(g) program end

Figure 3.3: Illustration of the DPST construction for a program with the following sequence of instructions - (1) the first spawn statement: `spawn A0`, (2) a subsequent spawn: `spawn A1`, and (3) a `sync` statement. The program is executed with two threads (`T0` and `T1`). The per-thread stack for each runtime thread along with the changes to the state of the stack as the sequence of instructions execute is shown. The nodes that have completed and removed from the per-thread stack have been grayed out.

Task Spawn

Algorithm 2 shows the updates to the DPST when the program spawns a task. On the execution of a spawn statement, the code fragment being executed by the thread before the spawn statement completes execution. Hence, TaskProf pops the step node at the top of the executing thread's stack since it represents the code fragment that completed.

When the program executes a spawn statement, TaskProf does not know at that moment whether there will be a future sync statement in the execution. TaskProf anticipates that the task may execute a sync statement. Hence, when the first task is spawned, TaskProf creates a finish node as the child of the node at the top of the executing thread's runtime stack. TaskProf also pushes the newly created finish node to the top of the per-thread stack. TaskProf needs to add a finish node not only for the first spawn in the program but also for the first spawn after every sync statement. On subsequent spawns, TaskProf does not need a finish node as it has already anticipated a sync with the previous spawn statement. In Figure 3.3(b), `F1` corresponds to the newly created finish node on the first spawn and it is also pushed to `T0`'s per-thread stack.

To represent the newly created task, TaskProf creates an async node as the child of the finish node at the top of the executing thread's per-thread stack. This async node is not pushed to any per-thread stack because the newly created task is still in the work queue maintained by the runtime. In Figure 3.3(b), `A0` is the async node that is added to the DPST. It is associated with the task in the work stealing queue of the task parallel runtime. TaskProf also adds a step node to the DPST as a child of the finish node at the top of the executing thread's per-thread stack and pushes it on to the stack. The step node represents the computation performed in the task after the task spawn. In Figure 3.3(b), the step node S2 represents the computation after spawning the task.

Algorithm 3: Updates to the DPST when a task begins execution.

```
1 procedure OnTaskExecute(tid, Stacks, P)
2     $Stacks[tid].Push(P)$                        ▷ Push async node
3     $S \leftarrow CreateNode(STEP)$
4     $S.parent \leftarrow P$
5     $Stacks[tid].Push(S)$
```

Execute Task

When a spawned task begins to execute, the *async* node corresponding to the task is added to the executing thread's per-thread stack, as shown in Algorithm 3. TaskProf adds a step node as the child of the async node to represent the execution of the code within the task. For instance in Figure 3.3(d), A0 executes on thread T1 and TaskProf adds it to T1's per-thread stack.

Algorithm 4: Updates to the DPST when a task completes execution.

```
1 procedure OnTaskReturn(tid, Stacks)
2     $Stacks[tid].Pop()$                          ▷ Pop Step node
3     $Stacks[tid].Pop()$                          ▷ Pop Async node
```

Complete Task

When the task finishes its execution, TaskProf pops the step and async nodes from the top of the executing thread's per-thread stack, as shown in Algorithm 4. Figure 3.3(e) shows the per-thread stacks after the step and async nodes corresponding to the tasks A0 and A1 have been popped following the completion of the two tasks.

Algorithm 5: Updates to the DPST on a sync statement.

```
1 procedure OnTaskSyncEnter(tid, Stacks)
2     $Stacks[tid].Pop()$                          ▷ Pop Step node
3 procedure OnTaskSyncExit(tid, Stacks)
4     $Stacks[tid].Pop()$                          ▷ Pop Finish node
5     $P \leftarrow Stacks[tid].Top()$
6     $S \leftarrow CreateNode(STEP)$
7     $S.parent \leftarrow P$
8     $Stacks[tid].Push(S)$
```

Task Sync

Algorithm 5 shows the updates to the DPST when the program executes a sync statement. On a sync statement, TaskProf first pops the step node that represents the code fragment before the sync statement. On completion of the sync statement, TaskProf pops the finish node that was added to the top of the executing thread's per-thread stack in the anticipation of a sync statement. In Figure 3.3(f), the finish node `F1`, that was added in anticipation of a future sync earlier, is popped from the per-thread stack of the executing thread (*i.e.*, `T0`). TaskProf also adds a step node to represent the computation after the sync statement. It is added to the DPST as a child of the node on the top of the executing thread's per-thread stack.

In summary, TaskProf maintains the nodes of the DPST in per-thread stacks, which ensure that no synchronization is required while constructing the DPST. Further, as the program executes and makes calls to the task runtime, the DPST nodes are added to the per-thread stacks. As tasks complete execution, the nodes corresponding to them in the DPST are popped from the per-thread stacks. At the end of the program only the root finish node remains in the per-thread stacks (as shown in Figure 3.3(g)). By removing the nodes from the stacks after completion, TaskProf maintains a slice of the DPST, whose size is bounded by the number of active tasks in the task runtime.

3.2.2 Fine-Grain Measurement of Computation

To accurately measure the work performed in the program, we must ensure that only the computation in the source program is measured and not the computation in the task runtime. Hence, TaskProf measures the computation performed in fragments of execution between task runtime calls. It records the work measurement in the corresponding step nodes of DPST that represent the fragment. TaskProf starts the measurement after a step node is added to the DPST and stops the measurement before the step node is removed from the DPST after it executes. Specifically, TaskProf starts the measurement immediately after the calls to functions `OnProgramStart`, `OnTaskSpawn`, `OnTaskExecute`, and `OnTaskSyncExit` shown in Algorithms 1, 2, 3, and 5 respectively. It stops the measurement before the calls to functions `OnTaskSpawn`, `OnTaskReturn`, and `OnTaskSyncEnter` shown in Algorithms 2, 4, and 5 respectively.

Along with computation in the step nodes, TaskProf measures the work performed in the task runtime to create tasks, to provide feedback about the parts of the program that have high task runtime overhead. Since our goal is to identify parts of the program that have high task creation work in comparison with the actual work performed in the task, we only measure task creation cost and not the work performed in stealing or scheduling tasks. TaskProf attributes the task creation work to the corresponding async node in the DPST.

To perform lightweight measurement, TaskProf uses hardware performance counters to measure work in code fragments corresponding to step and async nodes. Modern processors have performance monitoring units (PMU) that can be programmatically accessed to measure various hardware performance counter events during program execution. TaskProf can measure work in terms of any hardware performance counter event that is supported on the execution machine. However, by default TaskProf uses execution cycles to measure the work. While the hardware PMUs internally employ low-latency sampling to record event counts, TaskProf records precise work information and does not use sampling.

The DPST, along with fine-grain work measurements in the step and async nodes is a performance model of the program for a given input. Figure 3.2(b) shows the performance model of the example program in Figure 3.2(a) for a given input generated from a single execution of the program. The performance model consists of the DPST with step nodes S0-S15, async nodes A0-A15, and finish nodes F0-F8. The performance model also depicts the work in the computation represented by each step node and in the task creation represented by each async node.

TaskProf constructs the performance model and uses it to measure the parallelism in the program. In the offline mode, TaskProf writes the performance model to a profile data file and performs the parallelism analysis as a post-mortem process after reconstructing the performance model. Specifically, as each node is about to be popped from the per-thread stack, TaskProf writes the information corresponding to the node to a profile data file. If the node being written is a step node, the information includes the work performed in the step node. Similarly, if the node is an async node, TaskProf writes the task creation work to the profile data file. In the on-the-fly mode, TaskProf does not write the performance model to a file, but instead, performs the parallelism analysis on-the-fly as nodes are added and removed from the performance model.

3.3 Profiling for Parallelism and Task Runtime Overhead

In this section, we describe TaskProf's approach to compute the parallelism and the task runtime overhead. We design this analysis in two ways. One, as an offline post-mortem analysis, and the other as an on-the-fly analysis. In both the approaches, we use the performance model to compute parallelism and task runtime overhead.

The offline analysis constructs the performance model from an execution of the program and writes the performance model to a profile data file. Subsequently, the offline analysis reconstructs the performance model from the profile data file and computes the parallelism and the task runtime overhead. In contrast, the on-the-fly analysis does not write the performance model to a profile data file. Instead, it performs the parallelism computation as the program executes and as nodes are added to and removed from the performance model.

The primary rationale for designing two approaches was to profile long running applications and also provide the ability to perform multiple analyses on the stored profile data for smaller applications. The offline analysis is best suited for short running programs where the entire performance model can be stored on disk or in memory.

Further analyses (for *e.g.*, what-if and differential analyses discussed in Chapters 4 and 5) can be performed using the profile data and does not require re-execution of the program. The on-the-fly analysis is well suited for long running applications, but requires re-execution of the program to perform further analyses.

We first describe our approach as an offline analysis using the entire performance model. In section 3.3.2, we describe our approach to perform the computation on-the-fly.

3.3.1 Offline Parallelism and Task Runtime Overhead Analysis

TaskProf would have constructed the performance model for the program execution and produced a profile data file. In the offline analysis phase, TaskProf re-constructs the performance model of the program from the profile data file and computes the parallelism and the task runtime overhead. Given the entire performance model, it computes the parallelism by computing the work and critical path work at each internal node in the performance model. Along with the work and critical path work, TaskProf also computes the set of step nodes that execute on the critical path. This enables us to accurately attribute the critical path work to the static spawn sites in the program. Section 3.4 describes our approach to attribute the critical path work to the static spawn sites. To quantify the task runtime overhead, TaskProf computes the total task creation work at each internal node. To attribute the task creation work to the static spawn sites in the program, TaskProf also computes the task creation work that is exclusively performed by each internal node. Overall, TaskProf computes five quantities with each internal node in the performance model: (1) work (w), (2) critical path work (s), (3) set of step nodes on the critical path (l), (4) total task creation work (t), and (5) exclusive task creation work (et).

The above five quantities can be computed at each internal node in the performance model based on the nodes in the sub-tree rooted at the internal node. Hence, TaskProf performs a depth-first traversal of the performance model and computes the above five quantities at each internal node in a bottom-up manner. After the traversal of the entire performance model, the root node will hold the work, the critical path work, the task creation work and the set of step nodes on the critical path of the entire program. Algorithms 6 and 7 describe TaskProf's approach to compute the parallelism profile.

Algorithm 6: Generate the parallelism profile given a performance model T.

```
1  function ParallelismProfile($T$)
2      foreach $N$ in depth-first traversal of $T$ do
3          $C_N \leftarrow$ ChildNodes($N$)                          ▷ Return all child nodes of N
4          $N.w \leftarrow \sum_{C \in C_N} C.w$
5          $\langle N.s, N.l \rangle \leftarrow$ CriticalPathWork($N$)
6          $A_N \leftarrow$ AsyncChildNodes($N$)               ▷ Return async child nodes of N
7          $F_N \leftarrow$ FinishChildNodes($N$)              ▷ Return finish child nodes of N
8          $N.t \leftarrow \sum_{A \in A_N} A.c + \sum_{C \in C_N} C.t$
9          $N.et \leftarrow \sum_{A \in A_N} A.c + \sum_{F \in F_N} F.et$
10         AggregatePerSpawnSite($N$)                    ▷ Aggregate at spawn site
11     GenerateParallelismProfile()                          ▷ Produce profile
```

Computing Work, Critical Work, and Set of Step Nodes on the Critical Path

The total work at an internal node in the performance model is the sum of the work of all the step nodes in the sub-tree rooted at the internal node. As TaskProf performs the depth-first traversal, it tracks the total work at an internal node (variable w in Algorithm 6) as the sum of work in all child nodes of the internal node (line 4 in Algorithm 6).

To compute the critical path work, we need a strategy to identify the critical path at each internal node. Among all the step nodes in the sub-tree under an internal node, some subset of step nodes have to execute serially. There could be multiple such subsets. To identify the critical path we have to select the subset that has the highest aggregate of the work from all the step nodes in the subset.

In the absence of async and finish children at an internal node, all the step children have to execute sequentially. Hence, all the step children of an internal node form one subset of step nodes that execute sequentially. If an internal node contains a finish child, then according to the property of a finish node, the nodes in the sub-tree under the finish node execute in series with the step node siblings of the finish node. Hence, step nodes on the critical path at the sub-tree rooted at a finish child and the step node children form a subset of step nodes that execute sequentially. Finally, if an internal node contains async children, then each async node will introduce a separate subset since the nodes in the sub-tree under the async node execute in parallel with the right siblings of the async node. Each such subset will contain the step nodes in the critical path under the async node and the step nodes to the left[3] of the async node, which execute in series with the async node. These left step nodes are the left step node siblings and step nodes on the critical path of left finish node siblings of the async node.

Consider, for example, the sub-tree rooted at node A14 in the performance model shown in Figure 3.2 (b). Since A14 has a single step node S14 as its child, the critical path at A14 contains a single step node, S14. Similarly, node F8 contains two async children, A14 and A15, which introduce two separate subsets of step nodes, S14 and S15, in the sub-tree rooted at F8. Since the subset containing S15 performs the higher work, the critical path at F8 is made up of a single step node S15.

Algorithm 7 describes TaskProf's approach to compute the critical path work, s and the set of step nodes on the critical path, l of a given input node. The algorithm initially selects the step node children and step nodes on the critical path of the finish node children as the set of step nodes on the critical path of the input node. It also sets the work from this initial set as the critical path work (lines 4 and 5 in Algorithm 7). Now if the input node has async children, the algorithm has to check if the serial subset of step nodes at each async child forms the critical path at the input node. Hence, the algorithm examines the serial subset of step nodes created by each async node and chooses the path with the maximum critical path work. Similarly, the set of step nodes on the critical path (l) is also updated whenever the critical path work is updated (lines 6-12 in Algorithm 7).

[3]Node A is to the *left* of node B if A occurs before B in the depth-first traversal.

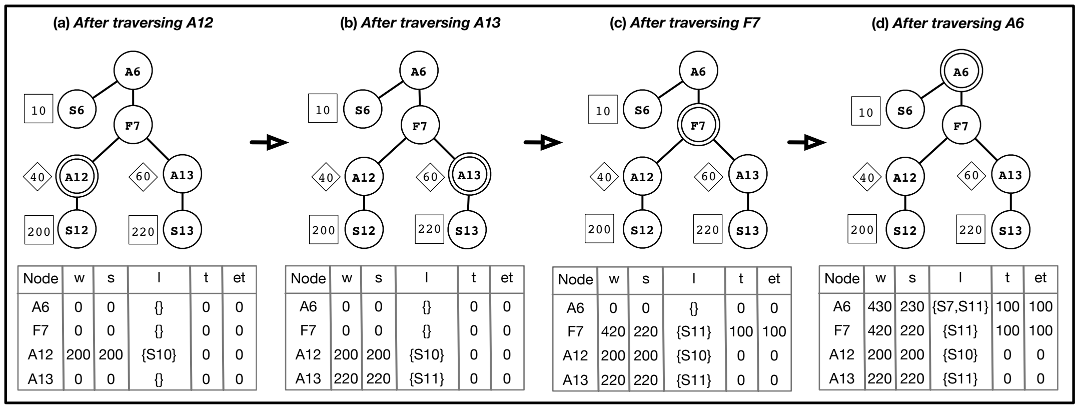
Algorithm 7: Compute the critical path work s and the list of step nodes on the critical path l for a given input node N.

```
function CriticalPathWork($N$)
    $S_N \leftarrow$ StepChildNodes($N$)                         ▷ Step child nodes of N
    $F_N \leftarrow$ FinishChildNodes($N$)                       ▷ Finish child nodes of N
    $s \leftarrow \sum_{S \in S_N} S.w + \sum_{F \in F_N} F.s$
    $l \leftarrow (\bigcup_{F \in F_N} F.l) \cup S_N$
    foreach $A \in$ AsyncChildNodes($N$) do
        $LS_A \leftarrow$ LeftStepSiblings($A$)            ▷ Step siblings to the left of A
        $LF_A \leftarrow$ LeftFinishSiblings($A$)        ▷ Finish siblings to the left of A
        $lw \leftarrow \sum_{S \in LS_A} S.w + \sum_{F \in LF_A} F.s$
        if $lw + A.s > s$ then
            $s \leftarrow lw + A.s$
            $l \leftarrow (\bigcup_{F \in LF_A} F.l) \cup LS_A \cup A.l$
    return $\langle s, l \rangle$
```

Computing Task Runtime Overhead

TaskProf quantifies the task runtime overhead in a program by computing the total task creation work and expressing it as a fraction of the total work in the program. To accurately identify the tasks that have high task runtime overhead, we need to attribute the task creation cost to the task that is spawning a new task rather than the newly spawned task. Further, to clearly separate the tasking overhead of a task from that of its descendants, we need to compute the task creation work from spawning only the immediate child tasks and not their descendants. Hence, along with the total task creation work, TaskProf also computes the exclusive task creation work incurred from spawning the immediate child tasks at each node in the performance model.

Each async node in the performance model, contains the task creation work incurred to spawn the new task (variable c in Algorithm 6). From the task creation work, TaskProf computes the total task creation work (variable t in Algorithm 6) and the exclusive task creation work (variable et in Algorithm 6) at each internal node in the performance model. The total task creation work at an internal node is the sum of the



Node	w	s	l	t	et
A6	0	0	{}	0	0
F7	0	0	{}	0	0
A12	200	200	{S10}	0	0
A13	0	0	{}	0	0

Node	w	s	l	t	et
A6	0	0	{}	0	0
F7	0	0	{}	0	0
A12	200	200	{S10}	0	0
A13	220	220	{S11}	0	0

Node	w	s	l	t	et
A6	0	0	{}	0	0
F7	420	220	{S11}	100	100
A12	200	200	{S10}	0	0
A13	220	220	{S11}	0	0

Node	w	s	l	t	et
A6	430	230	{S7,S11}	100	100
F7	420	220	{S11}	100	100
A12	200	200	{S10}	0	0
A13	220	220	{S11}	0	0

Figure 3.4: Illustration of the offline parallelism and task runtime overhead computation using the sub-tree rooted at A6 in Figure 3.2 (b). Figures (a), (b), (c), and (d) show the updates to the five quantities as nodes A12, A13, F7, and A6 are traversed in depth-first order, respectively. The node being visited in each figure is highlighted with a double edge.

task creation work of all async children and the total task creation work of all children, including the async children (line 8 in Algorithm 6). Similarly, the exclusive task creation work at an internal node is the sum of the task creation work of all the async children and the exclusive task creation work of the finish children (line 9 in Algorithm 6). After the traversal of the entire performance model, the root node will contain the total task creation work in the entire program, and the internal nodes will contain the exclusive task creation work incurred from spawning the immediate child tasks.

Offline Algorithm Illustration

Figure 3.4 illustrates using the sub-tree rooted at A6 in the performance model shown in Figure 3.2 (b), TaskProf's approach to track the work, the critical path work, the set of step nodes on the critical path, and the total and the exclusive task creation work with each internal node. The algorithm updates the six quantities as nodes A12, A13, F7, and A6 are traversed in depth-first order. The sub-tree rooted at node A12 contains a single step node S12. Hence the algorithm updates the work (w) and the critical path work (s) at A12 to the work from S12 and adds S12 to the set of step nodes on the critical path(l) of A12 (Figure 3.4(a)). The algorithm updates the quantities at node A13 similarly, since A13 contains a single step node S13 in its sub-tree (Figure 3.4(b)).

At node F7 the algorithm checks if the critical path work from either of the two async children, A12 and A13, is the critical path work at F7. Since the critical path work at A13 is greater than A12, the algorithm updates the critical path work and the set of step nodes on the critical path at F7 with the corresponding quantities of A13. Further, the algorithm updates the total and the exclusive task creation work (t and et) at F7 to the sum of the task creation work of the two async children (Figure 3.4(c)). Finally, since node A6 does not contain any async children the algorithm updates the critical path work at A6 with the sum of the work from the child step node (S7) and critical path work of the child finish node (F7). It also propagates the task creation work from F7 to A6 (Figure 3.4(d)).

3.3.2 On-the-fly Parallelism and Task Runtime Overhead Analysis

In the on-the-fly parallelism analysis mode, TaskProf does not write the performance model to a profile data file. Unlike the offline analysis, TaskProf's on-the-fly analysis has to perform the parallelism and the task runtime overhead computations as the nodes are added to and removed from the performance model. Performing the computations on-the-fly poses four main challenges.

First, as nodes are removed from the performance model after they complete execution, the parallelism and task runtime overhead computation at a node will not have access to other nodes, which have already been removed from the performance model. Hence, the analysis has to compute the parallelism independent of the nodes in the performance model that have completed.

Second, as the program executes in parallel, the nodes in the performance model will be added and removed as soon as parallel execution begins and ends. The parallelism and the task runtime overhead computation must be consistent irrespective of the order in which nodes are added to and removed from the performance model.

Third, as the computations are performed on-the-fly, they must not interfere with the measurement of work at step nodes or in task creation at async nodes. Hence, the on-the-fly analysis must ensure that the computation of parallelism and task runtime overhead is performed only when TaskProf is not profiling the execution to measure work.

Finally, to attribute the critical path work to the spawn sites in the program (described in Section 3.4), the offline analysis tracks the step nodes that execute on the critical path. But, tracking the step nodes on the critical path is possible only when the entire performance model is available. The on-the-fly analysis needs a mechanism to identify the parts of the program that execute on the critical path even when step nodes that execute on the critical path are removed from the performance model after execution.

We address the challenge of tracking information about the nodes that have completed by summarizing the information at their parent. To compute the critical path work at a node in the performance model, we only need the work and critical path work of the left step and finish siblings, respectively (lines 7-8 in offline analysis Algorithm 7). Since the left step and finish siblings execute serially with the node, they are guaranteed to have completed and removed from the performance model. Hence, for every step and finish node that has completed, we track the work and the critical path work at the parent node. Further, we can use this information to initialize the serial work from the left siblings at each node. This enables us to independently determine if a node is on the critical path by checking if the critical path work of the node and the serial work from its left siblings forms the critical path in the sub-tree rooted at its parent node.

We address the challenge in attributing the critical path work to the static spawn sites in the absence of the entire performance model by maintaining the set of static spawn sites that correspond to the step nodes that perform work on the critical path. Specifically, with each internal node in the performance model, we track the set of static spawn sites representing the step nodes on the critical path and the critical path work that can be attributed to each static spawn site.

Algorithm 8 presents our on-the-fly technique to compute the parallelism and the task runtime overhead. With each node in the performance model, we track six quantities. (1) The total work (w) performed in the sub-tree under the node. (2) The set of spawn sites that perform the work on the critical path (CS) in the sub-tree under the node. (3) The set of spawn sites that perform the serial work from step and finish children (SS). (4) The set of spawn sites that perform the serial work in the left step and finish siblings (LS). The set of spawn sites are tracked as a set of tuples, where each tuple consists of the spawn site and the work attributed to the spawn site. (5) The total task creation work (t) under the sub-tree of a given node. (6) The exclusive task creation work (et) to attribute the task runtime overhead to specific spawn sites.

TaskProf updates these six quantities when nodes are added to and removed from the performance model. To ensure that the computation does not interfere with the measurement, TaskProf updates these quantities only after a node has completed or before a node is added since the work measurements are performed only when a step node or async node is executing.

Algorithm 8: On-the-fly parallelism and task runtime overhead computation performed when parent node P creates child node N and when N completes.

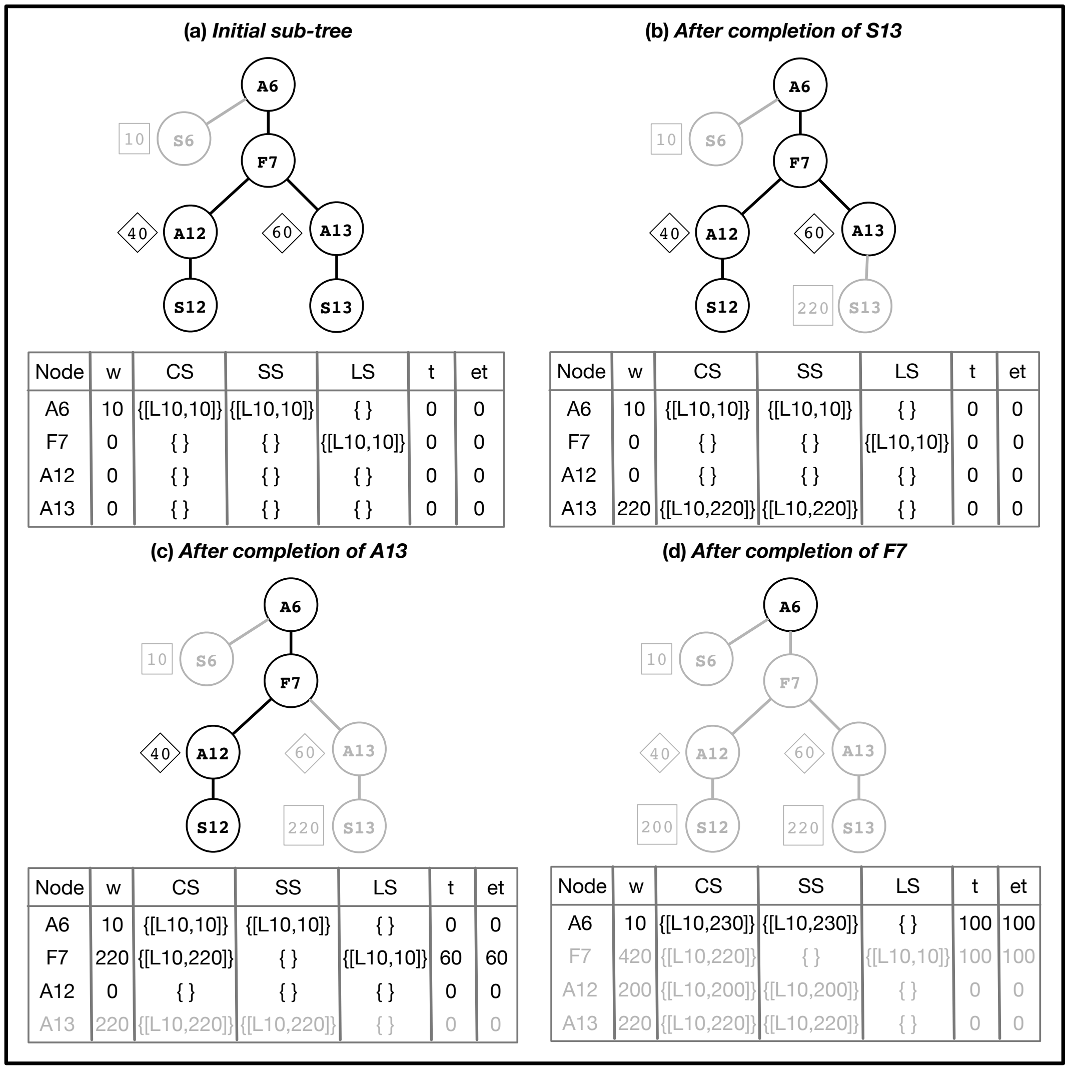
```
1  procedure OnNodeCreation(N, P)
2      $N.w \leftarrow 0$                          ▷ work
3      $N.t \leftarrow 0$                          ▷ total task creation work
4      $N.et \leftarrow 0$                         ▷ exclusive task creation work
5      $N.CS \leftarrow \emptyset$                 ▷ set CS tracks critical path work
6      $N.SS \leftarrow \emptyset$                 ▷ set SS tracks serial work
7      $N.LS \leftarrow P.SS$                      ▷ set LS tracks left serial work
8  procedure OnNodeCompletion(N, P)
9      $P.w \leftarrow P.w + N.w$
10     if $\sum_{L \in N.LS} L.w + \sum_{C \in N.CS} C.w > \sum_{S \in P.CS} S.w$ then
11         $P.CS \leftarrow N.LS \uplus N.CS$
12     if $N$ is an ASYNC node then
13         AggregatePerSpawnSite($N$)            ▷ Aggregate at spawn site
14         $P.t \leftarrow P.t + N.c + N.t$
15         $P.et \leftarrow P.et + N.c$
16     else
17         $P.SS \leftarrow P.SS \uplus N.CS$
18         $P.t \leftarrow P.t + N.t$
19         $P.et \leftarrow P.et + N.et$
```

When a node is added to the performance model, TaskProf initializes the left serial work (LS) of the node with the serial work (SS) of the parent node (line 7 in Algorithm 8). When a new node is created the serial work (SS) of the parent node will reflect the serial work of the left step and finish siblings of the new node. To determine if the newly created node is on the critical path under the parent node, TaskProf requires the serial work of the left siblings of the new node. By initializing the left serial work (LS) of the new node with the serial work (SS) of the parent node, TaskProf can independently determine if the new node is on critical path when the new node completes execution.

When a node completes, TaskProf summarizes the node's work, critical path work, and task creation work at the parent node. TaskProf first adds the work of the node to its parent node (line 9 in Algorithm 8). Then it checks if the node contributes to the critical path under the parent node. Specifically, TaskProf checks if the serial work done by the left siblings and critical path work in the sub-tree under the node is greater than the work on the critical path for the parent node (line 10 in Algorithm 8). If the node indeed contributes to the critical path under the parent node, TaskProf updates the set of spawn sites that perform the work on the critical path (CS) under the parent node with spawn sites performing the left serial work (LS) and the critical path work (CS) of the child node (line 11 in Algorithm 8). Finally, if the node is a finish or a step node, TaskProf has to update the serial set (SS) of the parent since the step and finish children perform the serial work under the parent node (line 17 in Algorithm 8). The serial set (SS) is useful to initialize the left sibling work (LS) for future children of the parent node. If the node is an async node, TaskProf aggregates the work, critical path work, and task runtime overhead to the static spawn site in the program (line 13 in Algorithm 8)

In the on-the-fly analysis, TaskProf computes the total task creation work and the exclusive task creation work based on the type of the given node. It adds the total task creation work under the sub-tree of the async node to the total task creation work of the parent (line 14 in Figure 8). It adds the cost of task creation to the exclusive task creation work of the parent (line 15 in Figure 8). If a finish node completes, TaskProf propagates the total task creation work and the exclusive task creation work to the parent (lines 18-19 in Figure 8).

At the end of the execution of the program, only the root node of the performance model will remain while all the other nodes would have completed. The root node will have the total work in the program in the w variable, the spawn sites executing on the



(a) *Initial sub-tree*

Node	w	CS	SS	LS	t	et
A6	10	{[L10,10]}	{[L10,10]}	{ }	0	0
F7	0	{ }	{ }	{[L10,10]}	0	0
A12	0	{ }	{ }	{ }	0	0
A13	0	{ }	{ }	{ }	0	0

(b) *After completion of S13*

Node	w	CS	SS	LS	t	et
A6	10	{[L10,10]}	{[L10,10]}	{ }	0	0
F7	0	{ }	{ }	{[L10,10]}	0	0
A12	0	{ }	{ }	{ }	0	0
A13	220	{[L10,220]}	{[L10,220]}	{ }	0	0

(c) *After completion of A13*

Node	w	CS	SS	LS	t	et
A6	10	{[L10,10]}	{[L10,10]}	{ }	0	0
F7	220	{[L10,220]}	{ }	{[L10,10]}	60	60
A12	0	{ }	{ }	{ }	0	0
A13	220	{[L10,220]}	{[L10,220]}	{ }	0	0

(d) *After completion of F7*

Node	w	CS	SS	LS	t	et
A6	10	{[L10,230]}	{[L10,230]}	{ }	100	100
F7	420	{[L10,220]}	{ }	{[L10,10]}	100	100
A12	200	{[L10,200]}	{[L10,200]}	{ }	0	0
A13	220	{[L10,220]}	{[L10,220]}	{ }	0	0

Figure 3.5: Illustration of the on-the-fly parallelism and task runtime overhead computation using the sub-tree rooted at A6 in Figure 3.2 (b). Figures (a) shows the initial sub-tree. Figures (b), (c), and (d) show the updates to the sub-tree and the six quantities after the completion of step node S13, async node A13, and finish node F7. The nodes that have completed and their corresponding entries have been greyed out.

critical path of program in the set *CS*, the critical path work as the aggregate of the work in the spawn site entries in the set *CS*, the total task creation work in the variable *t*, and the exclusive task creation work of the root node in the variable *et*. TaskProf uses the information from the root node to generate the parallelism profile similar to Figure 3.2 (c).

On-the-fly Algorithm Illustration

Figure 3.5 illustrates the on-the-fly algorithm using the sub-tree rooted at A6, similar to Figure 3.4. Figure 3.5 (a) shows the initial sub-tree after all the nodes have been added and step node S6 has completed. The algorithm updates the work (w), and the set of spawn sites performing critical path work (CS) and serial work (SS) of node A6 after the completion of S6. It also sets the left serial work (LS) of F7 using the serial work of A6, when F7 is added to the performance model. When step node S13 completes, the algorithm updates the work, and the set of spawn sites performing critical path work and serial work of node A13 (Figure 3.5 (b)). When async node A13 completes, the algorithm checks if the critical path under A13 performs the critical path work under F7. Since the critical path work at A13 is the highest critical path work under F7 when A13 completes, the algorithm updates critical path work of F7 (Figure 3.5 (c)). The algorithm performs a similar check when A12 completes. It is important to point out here that irrespective of the order in which A12 and A13 complete the on-the-fly algorithm will select the highest of the two paths as the critical path, which is the path through A13 in this scenario. Finally, when finish node F7 completes, the algorithm updates the critical path work of the parent node A6 and propagates the total and the exclusive task creation work (Figure 3.5 (d)).

3.4 Programmer Feedback

After the parallelism and task runtime overhead analysis, either on-the-fly or offline, the performance model contains information about the total work, the critical path work, and the task runtime overhead of the entire program. The programmer can use this information to determine if there is adequate parallel work in the program to achieve scalable speedup. If the program has low parallelism or high task runtime overhead, the programmer would benefit from knowing the specific parts of the program that have low parallelism and high runtime overhead. Hence, in addition to computing the parallelism and the runtime overhead of the entire program, we compute the parallelism and the runtime overhead at various static code locations in the program.

The performance model during the on-the-fly parallelism analysis or after the offline parallelism analysis contains valuable information about the parallelism and the task runtime overhead in the entire program, and at various parts of the program. To provide actionable feedback to the programmer, TaskProf attributes the parallelism and the task runtime overhead from the performance model to the program source code. Specifically, TaskProf attributes the parallelism and the task runtime overhead to each spawn site in the program, which is the static location in the program where a task is spawned. To attribute the parallelism and the task runtime overhead to spawn sites, TaskProf has to store additional information with the performance model to identify the spawn sites. TaskProf stores the name of the file that contains the spawn site and the specific line number in the file where the spawn site is located. Since the async nodes in the performance model represent the spawn sites in the program, TaskProf records the spawn site information along with the async nodes in the performance model. Figure 3.2(b) shows the performance model with static source code location recorded with each async node for attribution.

In the offline analysis mode, TaskProf aggregates the parallelism and task runtime overhead information after the node has been processed while traversing the performance model (line 10 in Algorithm 6). In the on-the-fly mode, TaskProf aggregates the information when a node completes execution and after work and critical path work have been computed at the parent node (line 13 in Algorithm 8). TaskProf uses a hash table indexed by the spawn site for aggregation. Each entry in the hash table contains the total work, the critical path work, and the exclusive task creation work corresponding to a spawn site in the program. If a node that is processed corresponds to a spawn site, TaskProf add the total work, the critical path work, and the exclusive task creation work from the node to the entry corresponding to the spawn site in the hash table. Once TaskProf traverses the entire performance model in the offline mode, or the program completes execution in the on-the-fly mode, the hash table will contain the total work, the critical path work, and the exclusive task creation work performed at each spawn site in the program. TaskProf uses the information from the hash table to generate the parallelism and tasking overhead profile.

When aggregating the work and critical path work at each spawn site, TaskProf has to ensure that it does not double count work from recursive task spawns. In the presence of recursive spawns, a descendant of an async node in the performance model will have the same spawn site information as the async node. If we naively add the descendant's work, it leads to double counting as the work and critical path work of the current async node already considers the work and critical path work of the descendant async node. Hence, when TaskProf encounters an async node in a bottom-up traversal of the DPST, it checks whether the ancestors of the async node have the same spawn site information. When an ancestor with the same spawn site exists, TaskProf does not add the work and critical path work information to the entry in the hash table. For instance, the async node A4 is a recursive task spawn since its ancestor A0 has the same spawn site (line L9) as A4. Hence, TaskProf does not add the work and the critical path work from A4 to the hash table entry corresponding to line L9. In contrast, the async node A5 is not a recursive call since none of its ancestors have the same spawn site (line L10) as A5. Hence, TaskProf adds the work and critical path work from A5 to the hash table entry corresponding to line L10.

For each spawn site in the program, the parallelism and tasking overhead profile presents the work, the critical work, the parallelism, the percentage of critical work performed by the spawn site, and the percentage of the task runtime overhead attributed to the spawn site. Figure 3.2(c) shows the profile for the program in Figure 3.2(a) produced from the performance model shown in Figure 3.2(b). It shows that the program has a low parallelism of 3.26 and a high tasking overhead of 27.63% for the entire program (see the tasking overhead percent in the last line of Figure 3.2(c)). The profile also indicates that the two spawn sites (line 9 and line 10 in Figure 3.2(a)) together account for almost 75% of the task runtime overhead. We can reduce tasking runtime overhead by decreasing the parallelism until the work done by the step nodes is reasonably higher than the average task creation overhead. At the same time, if a spawn site has low parallelism and performs a significant proportion of the critical path work, then parallelizing the task spawned by the spawn site may increase the parallelism in the program. This profile information provides a succinct description of the parallelism and the task runtime overhead in the entire program and at various parts of the program.

3.5 Summary

Sufficient parallel work and low runtime overhead are two important factors that are essential for a task parallel program to have scalable speedup. In this chapter, we have presented an effective profiling technique to determine whether programs have insufficient parallel work or high task runtime overhead and pinpoint parts of the program that are experiencing these pathologies.

Our technique measures the logical parallelism and the total task creation in the program and uses these metrics to quantify the parallel work and the task runtime overhead in the program. A key enabler of the parallelism and task runtime overhead computation is a performance model that encodes logical series-parallel relations and fine-grain work performed in the computation and task creation. We have developed a concrete instantiation of this technique in TaskProf, a profiler we have designed for task parallel programs. Using the parallelism and task runtime overhead computation in the performance model, TaskProf produces a profile that provides the programmer insightful feedback on the parts of the program that have insufficient parallel work and high task runtime overhead.

The parallelism and the task runtime overhead performance metrics are intuitive. If a program does not have sufficient parallelism on a given execution machine, then it will not exhibit scalable speedup. The programmer can use the parallelism profile to identify spawn sites that are having low parallelism. Similarly, if a program has high task runtime overhead, then the profile will highlight the spawn sites that are responsible for the high overhead.

We have shown that by attributing the parallelism and the task runtime overhead to the program source code the programmer can obtain valuable insight into the performance of a task parallel program. A strength of our work is that by measuring the logical parallelism, we can identify if a program has sufficient parallelism to obtain scalable speedup on any machine with any processor count. The performance model itself has enabled further avenues for identifying parts of the program that matter for improving the performance and speedup of programs.

Chapter 4

Parallelism-Centric What-if Analyses

Profiling for parallelism is an important first step in understanding if a program has sufficient parallel work to achieve scalable speedup. If a program has low parallelism, then the programmer would want to address the bottlenecks in the program to increase the parallelism. However, parallelizing parts of a program with low parallelism may not increase the overall parallelism or the speedup of the program if the work performed on the critical path remains the same. In this chapter, we explore the question - *what parts of the program should a programmer focus on to improve the parallelism in the program?*

4.1 Motivation and Overview

As modern machines are progressively adding parallel execution units, programmers can improve performance by adding more parallelism to their applications. However, manually identifying the parts of a program that have to be parallelized to improve the performance is an arduous task. Instead, programmers rely on profilers to identify parts of the program that must be parallelized. While numerous profilers have been proposed [10, 47, 48, 51, 65, 90, 150, 151, 158, 168], most of them identify the "*hotspots*", *i.e.*, they measure utilization of various resources (*e.g.*, execution time, cycles, *etc.*) and attribute them to the program source code. The idea is to identify parts of the program where resources are most utilized and parallelize the identified parts to increase the performance of the program.

Merely measuring where the program performs the most work is not sufficient to identify parts of the program that must be parallelized. Consider for example, a profiler that reports the percentage of work performed in various parts of a program.

Figure 4.1 shows a profile generated by such a profiler for the example task parallel program in Figure 3.2(a). The profile highlights the region between lines 4 and 6 as the region that performs the most work in the program. However, parallelizing the region does not improve the speedup of the program since most of the work is performed in parallel and only a small fraction of the work is performed on the critical path.

Region	Work percentage
L4 - L6	55.26
L16 - L20	21.71
L24 - L28	20.4
L8 - L9	1.32
L32 - L33	0.98
L38 - L39	0.33

Figure 4.1: Profile showing the percentage of the total work performed by each region in an execution of the program in Figure 3.2(a).

To identify parts of the program that must be parallelized, it therefore appears that the programmer must identify code that performs significant work on the critical path. But optimizing the code that performs work on the critical path may not increase the speedup of the program if the critical path shifts to another parallel path that performs slightly lesser work than the original critical path. For instance, TaskProf's parallelism profile in Figure 3.2(c) shows that the spawn site at line 34 performs the highest work on the critical path. But parallelizing the task spawned at line 34 causes the spawn site at line 35 to execute on the critical path. As a result, the parallelism and the speedup of the program will remain almost the same.

Along with measuring the parallelism of a program, the programmer needs to identify code that executes on the critical path and project how the critical path changes when parts of the program are parallelized. The programmer can manually parallelize regions of code and check if the parallelism or the speedup of the program improves as a result. However, designing concrete parallelization strategies can require considerable effort. *What if* we could identify the parts of the program that have to be parallelized before even designing a concrete parallelization strategy?

We propose *what-if analyses*, a technique to estimate the parallelism of the program on hypothetically parallelizing a region of code. To perform what-if analyses, TaskProf uses the performance model and mimics the effect of parallelizing a region by reducing the region's contribution to the critical path work of the program by some factor. TaskProf then re-computes the parallelism of the program using the reduced critical path work. If the estimated parallelism is greater than the original parallelism of the program, then parallelizing the region will increase the parallelism of the program. Parallelizing the region will likely increase the speedup of the program, unless the parallelism is so high that the task runtime overhead dominates the execution. Section 4.2 details our approach to perform what-if analyses in both the offline and the on-the-fly contexts.

To use what-if analyses for performance debugging, one approach is that the programmer would identify a set of regions in the program as candidates for what-if analyses. The regions identified can either be based on the parallelism profile or the programmer's intuition on parallelizing the regions. However, such an approach requires the programmer to manually identify the regions. Instead, we can use what-if analyses to automatically identify a set of regions that have to be parallelized to increase the parallelism of the program. Hence, TaskProf is both a parallelism profiler and an adviser for task parallel programs.

To automatically identify regions to parallelize, TaskProf finds the region performing the highest work on the critical path and performs what-if analyses to estimate the parallelism on parallelizing the region. TaskProf repeats this process until the estimated parallelism increases to a pre-defined level specified by the programmer. The set of regions considered for what-if analyses in each iteration are the regions that are presented to the programmer. Section 4.3 presents an algorithm and describes our approach to automatically identify regions that have to be parallelized.

Overall, TaskProf provides two interfaces to the programmer to perform what-if analyses. First, it identifies a set of regions that have to be parallelized to increase the parallelism of the program. The programmer can explore strategies to parallelize the regions reported by TaskProf. Second, it provides a method to estimate the parallelism of the program after parallelizing a region of code in the program. If the programmer has intuition about the parts of the program that can be parallelized, then the programmer can use TaskProf to check if parallelizing the regions increases the parallelism of the program.

A key aspect of TaskProf's what-if analyses is that it enables the programmer to ascertain if parallelizing parts of the program matters even before designing a concrete parallelization strategy. Since parallelism is a property of the program for a given input and is independent of a given execution, we can execute the program on one machine and identify regions that must be parallelized to achieve a parallelism that is necessary for scalable speedup on a completely different machine. Hence, what-if analyses enables us to predict from a single execution, the program regions that must be parallelized to achieve scalable speedup on any machine.

4.2 Profiling for What-if Analyses

To parallelize a serial piece of code, a common approach is to perform the same overall computation as the serial code, but split the computation into parallel threads of execution. For instance, consider a piece of code that contains a serial for-loop performing N iterations. To parallelize the for-loop, we can split the N iterations among T threads with each thread performing N/T-th fraction of the computation in parallel. In other words, the parallel code performs the same amount of work as the serial code, but the work is performed in parallel.

When a region of code is parallelized, the performance model used by TaskProf to compute the parallelism also changes. Therefore, we can use the performance model to determine the effect of parallelization on the parallelism of the program. Consider, for example, the performance model shown in Figure 4.2(a). If the region corresponding to step node S3 is parallelized by spawning two parallel tasks, the step node S3 is replaced by a sub-tree containing two async nodes as shown in Figure 4.2(b). On computing the parallelism in Figure 4.2(b) we find that the total work remains the same, but the critical path work is reduced, and the parallelism has increased since the series-parallel relations have changed as a result of parallelizing the region corresponding to S3.

In Figure 4.2(b), the parallelism of the program increased after parallelizing the region at S3 because S3 executes on the critical path of the program. But, if a region does not perform any work on the critical path, parallelizing the region will not increase

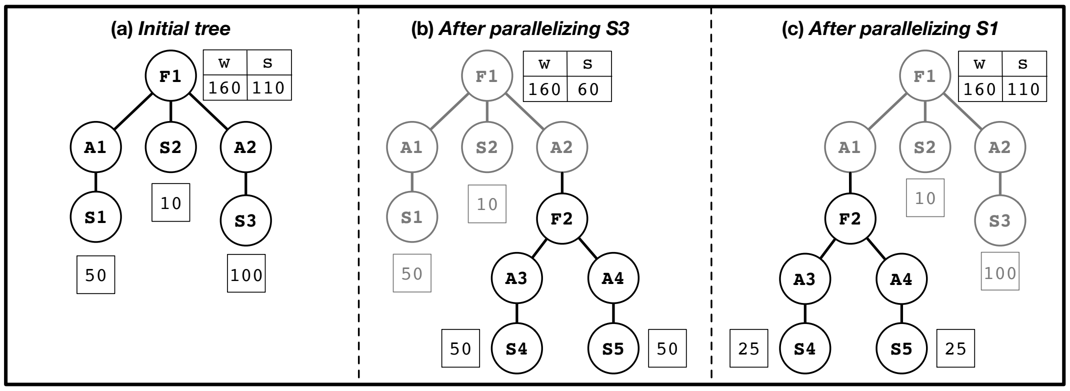


Figure 4.2: Illustration of the how the performance model changes when a region of code corresponding to a step node is parallelized. Figure (a) shows the initial performance model with the work performed in each step node shown next to the step node in rectangular boxes and the work (w) and critical path work (s) of the program shown at the root node `F1`. Figure (b) shows the performance model along with the work and critical path work if region of code corresponding to step node `S3` in Figure (a) is parallelized. Similarly, Figure (c) shows the performance model if step node `S3` is parallelized. The sub-tree corresponding to the region that is parallelized has been highlighted by greying out the rest of the nodes.

the parallelism. For example, even after parallelizing the region at step node S1 in Figure 4.2(a) the work and critical path work remain the same since S1 does not perform any work on the critical path, as shown in Figure 4.2(c).

TaskProf's what-if analyses technique does not explicitly modify the performance model. Instead, it mimics the effect of parallelization by reducing the contribution to the critical path work. It takes as input a set of static program regions to parallelize and a corresponding parallelization factor for each region, which determines the degree of parallelization for each region. The programmer can specify the regions and the parallelization factor as annotations in the program. TaskProf provides two annotations, `WHAT_IF_BEGIN` and `WHAT_IF_END`, to demarcate the regions for what-if analyses. The programmer can also specify the parallelization factor along with the annotations. If the programmer does not specify the parallelization factor, TaskProf assumes a default value. The annotated regions need not cover the entire source code region represented by a step node. Instead, it can even be a small segment of code with the region represented by a step node. Further, there can be multiple such annotated regions within a single step node. TaskProf performs what-if analyses on all dynamic executions of the statically annotated regions. Figure 4.3(a) shows the use of these annotations to demarcate source code regions for what-if analyses.

Similar to parallelism profiling, TaskProf performs what-if analyses either as an offline or an on-the-fly analysis. In the offline analysis, TaskProf executes the program, constructs the performance model, and measures work performed in the regions designated for what-if analyses. It writes the performance model and the what-if analyses regions to a profile data file. Then, in an offline post-mortem execution, TaskProf performs what-if analyses to estimate the parallelism on parallelizing the regions designated for what-if analyses. With the offline what-if analyses, the program does not need to be re-executed if the annotated region covers an entire step node. This is helpful in the analysis to identifying regions to parallelize, where TaskProf iterates over the performance model to automatically identify regions to parallelize. If the annotated region for what-if analyses is a small segment in the region represented by a step node, TaskProf would need to re-execute the program to record the work information for the annotated region.

For long running programs where storing the performance model on disk or in memory would not be feasible, TaskProf can perform what-if analyses on-the-fly as the program executes. Unlike the offline analyses, TaskProf would need to re-execute the program for what-if analyses. After what-if analyses in either the offline or the on-the-fly mode, TaskProf generates a profile called the *what-if profile* which specifies the estimated parallelism for the entire program from hypothetically parallelizing the annotated regions. Figure 4.3(c) shows a sample what-if profile.

In the rest of this section, we describe in detail our approach to perform what-if analyses using the performance model. First, we describe the enhancements to the performance model that enable what-if analyses in Section 4.2.1. Subsequently, we present our approach to perform what-if analyses as a part of TaskProf's offline parallelism analysis in Section 4.2.2. In Section 4.2.3, we present our approach to perform on-the-fly what-if analyses.

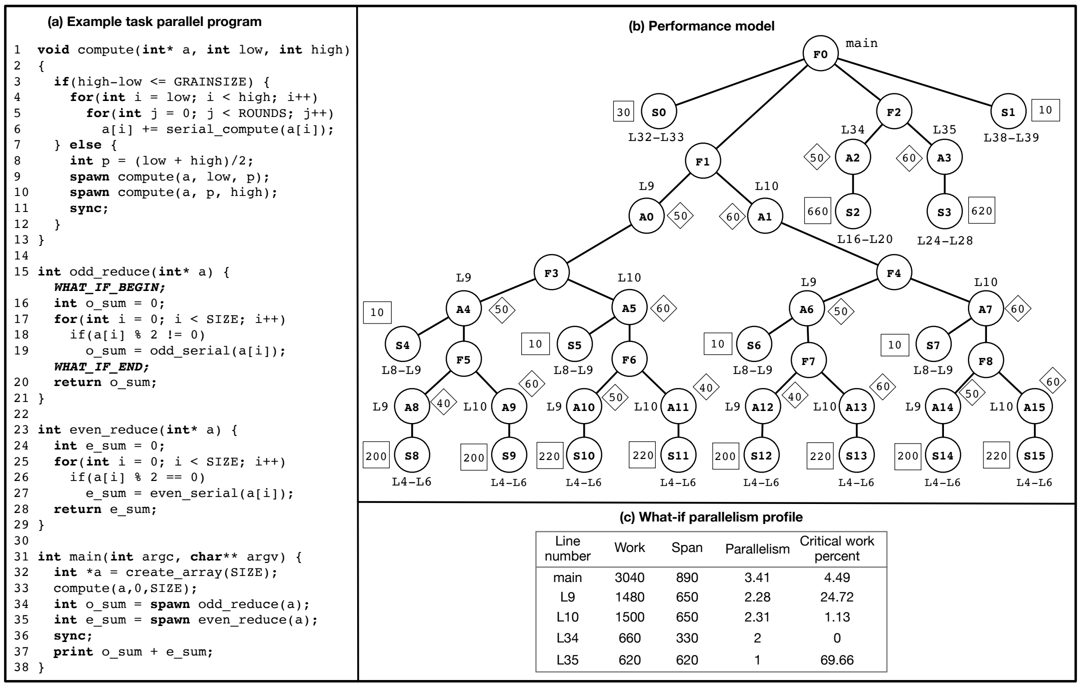
4.2.1 Performance Model for What-If Analyses

TaskProf's performance model for parallelism computation contains series-parallel relations encoded in the Dynamic Program Structure Tree (DPST), fine-grain work performed in computations represented by step nodes, and task creation work represented at asnyc nodes. To perform what-if analyses, we need additional information in the performance model about the work performed in the regions annotated for what-if analyses. According to the semantics of the DPST, every region of code in the program between task management constructs is represented by a step node. Hence, one possible approach is to record the work performed in the regions annotated for what-if analyses in the corresponding step nodes in the performance model. There can be multiple such annotated regions within the same step node. In such a scenario, TaskProf has to store a list of regions along with the work performed in the regions with corresponding step nodes in the performance model. However, such an approach would require iterating through the list of regions to perform what-if analyses. Instead, we create a separate step node in the performance model for every execution of an annotated region.

Algorithm 9: Updates to the performance model on `WHAT_IF_BEGIN` and `WHAT_IF_END` annotations.

```
1 procedure OnWhatIfBeginAndEnd(tid, Stacks, Region)
2     RecordWorkAndRegion(Region)        ▷ Add work, region to step node
3     $Stacks[tid].Pop()$                                  ▷ Pop Step node
4     $P \leftarrow Stacks[tid].Top()$
5     $S \leftarrow CreateNode(STEP)$
6     $S.parent \leftarrow P$
7     $Stacks[tid].Push(S)$
```

When TaskProf encounters the `WHAT_IF_BEGIN` annotation during profile execution, it terminates the executing step node and creates a new step node. This new step node represents the region starting from the `WHAT_IF_BEGIN` annotation to the `WHAT_IF_END` annotation. TaskProf records the static region information and the work performed in the region at the step node. In the case where the annotated region covers the entire region represented by the step node, TaskProf does not create a new step node, but instead records the region information and the work along with the existing step node.



```
1  void compute(int* a, int low, int high)
2  {
3    if(high-low <= GRAINSIZE) {
4      for(int i = low; i < high; i++)
5        for(int j = 0; j < ROUNDS; j++)
6          a[i] += serial_compute(a[i]);
7    } else {
8      int p = (low + high)/2;
9      spawn compute(a, low, p);
10     spawn compute(a, p, high);
11     sync;
12   }
13 }
14
15 int odd_reduce(int* a) {
     WHAT_IF_BEGIN;
16   int o_sum = 0;
17   for(int i = 0; i < SIZE; i++)
18     if(a[i] % 2 != 0)
19       o_sum = odd_serial(a[i]);
     WHAT_IF_END;
20   return o_sum;
21 }
22
23 int even_reduce(int* a) {
24   int e_sum = 0;
25   for(int i = 0; i < SIZE; i++)
26     if(a[i] % 2 == 0)
27       e_sum = even_serial(a[i]);
28   return e_sum;
29 }
30
31 int main(int argc, char** argv) {
32   int *a = create_array(SIZE);
33   compute(a,0,SIZE);
34   int o_sum = spawn odd_reduce(a);
35   int e_sum = spawn even_reduce(a);
36   sync;
37   print o_sum + e_sum;
38 }
```

Line number	Work	Span	Parallelism	Critical work percent
main	3040	890	3.41	4.49
L9	1480	650	2.28	24.72
L10	1500	650	2.31	1.13
L34	660	330	2	0
L35	620	620	1	69.66

Figure 4.3: (a) An example task parallel program with the region between lines 16 and 19 annotated using `WHAT_IF_BEGIN` and `WHAT_IF_END` annotations for performing what-if analyses. (b) The performance model for the program in (a) containing additional information about the static region of code represented by each step node. (c) What-if profile reported after TaskProf's what-if analyses for the annotated region between lines 16 and 19 in (a).

In contrast to iterating through a list of regions, creating a separate step node for each annotated region enables TaskProf to check if the step node region is one of the regions selected for what-if analyses. Algorithm 9 shows the changes to the performance model, on encountering `WHAT_IF_BEGIN` and `WHAT_IF_END` annotations.

In summary, the enhanced performance model of TaskProf to perform what-if analyses has four components: (1) series-parallel relations encoded as the DPST, (2) fine-grain computation work with step nodes, (3) fine-grain task creation work with async nodes, and (4) static source code region information with each step node. Further, TaskProf creates a new step node for each execution of an annotated region. Figure 4.3(b) shows the enhanced performance model generated for the program in 4.3(a). The annotated region in 4.3(a) represents the entire step node S2 in the performance model in Figure 4.3(b). The performance model holds sufficient information that enables TaskProf to perform what-if analyses in both offline and on-the-fly profiling.

4.2.2 Offline What-if Profiling

In the offline what-if analyses mode, TaskProf records the performance model along with the regions that have been annotated for what-if analyses in the profile data file. In a post-mortem analysis, TaskProf reconstructs the performance model and produces the what-if parallelism profile that estimates the parallelism on hypothetically parallelizing the regions annotated for what-if analyses.

To estimate the parallelism of the program for what-if analyses, TaskProf computes the parallelism similar to the offline parallelism analysis described in Section 3.3.1. Given the entire performance model, TaskProf computes the following four quantities at each internal node by traversing the performance model in a depth-first manner. The quantities are - (1) work, (2) critical path work, (3) set of step nodes on the critical path, and (4) total task creation work. The total task creation work is used by TaskProf for automatically identifying the regions to parallelize (see Section 4.3).

TaskProf computes the total work using the work information of all the step nodes before they are reduced for what-if analyses. The total work at an internal node is the sum of the work of all the step nodes in the sub-tree rooted at the internal node. Hence the total work computed by TaskProf while performing what-if analyses will be the same as the total work computed in the parallelism computation without what-if analyses. The critical path work, and the list of step nodes on the critical path can change while performing what-if analyses.

To perform what-if analyses, TaskProf mimics the effect of parallelization and reduces the critical path work. Specifically, before computing the critical path work at an internal node, TaskProf reduces the work of all the step nodes in its sub-tree that represent regions that have been annotated for what-if analyses. Once the work in the step nodes has been reduced, TaskProf computes the critical path work and the list of step nodes on the critical path by identifying the subset of step nodes that execute sequentially and have the highest aggregate work.

To provide a concrete example of what-if analyses, consider the sub-tree rooted at node A6 in the performance model in Figure 4.3(b). The total work at A6 is the sum of the work of all the step nodes in its sub-tree, which is 430. The critical path work at A6 is 230, since the critical path comprises of the step nodes S6 and S13. Consider for instance, the programmer annotates the region between lines 4 and 6 in Figure 4.3(a) for what-if analyses with a parallelization factor of 2. To perform what-if analyses, TaskProf computes the total work by taking into consideration the original work of the step nodes in the sub-tree under A6, which is 430. Before computing the critical path work, TaskProf mimics parallelization by reducing by a factor of 2, the work of the two step nodes S12 and S13 in the sub-tree under A6, which represent the region between lines 4 and 6. Finally, TaskProf computes the critical path work at A6 as 120, which is the aggregate of the work of the two step nodes S12 and S13 that form the critical path. After what-if analyses the total work under the sub-tree remains the same, but the critical path work reduces.

Algorithm 10: Offline What-if analyses given a performance model T, and regions annotated for what-if analyses RG.

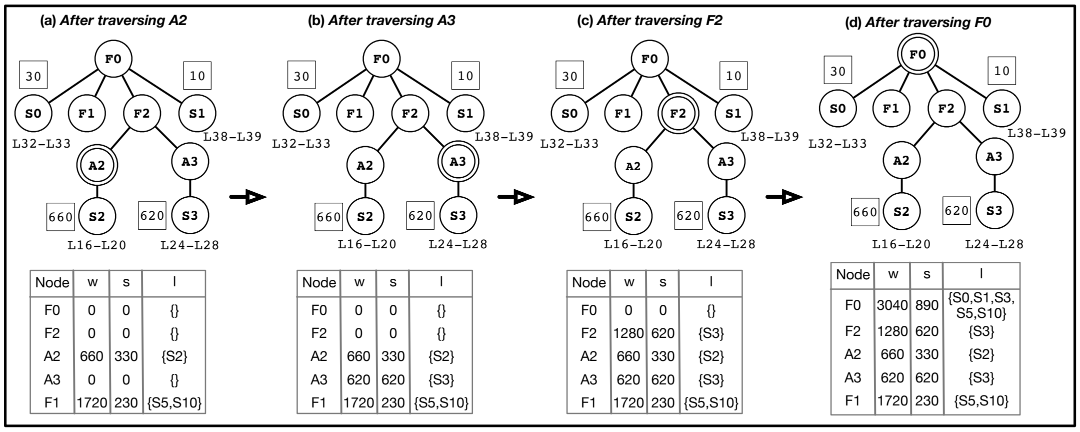
```
1  function WhatIfProfile($T$, $RG$)
2      foreach $N$ in depth-first traversal of $T$ do
3          $C_N \leftarrow$ ChildNodes($N$)
4          $N.w \leftarrow \sum_{C \in C_N} C.w$
5          $S_R \leftarrow \{S | (S \in C_N \wedge S.r \in RG)\}$        ▷ r holds the source code region
6          foreach $S \in S_R$ do
7              $R \leftarrow \{R | (R \in RG \wedge S.r == R)\}$
8              $S.w \leftarrow S.w / R.f$
9          $\langle N.s, N.l \rangle \leftarrow$ CriticalPathWork($N$)
10         $A_N \leftarrow$ AsyncChildNodes($N$)
11         $F_N \leftarrow$ FinishChildNodes($N$)
12         $N.t \leftarrow \sum_{A \in A_N} A.c + \sum_{C \in C_N} C.t$
13     AggregateWhatifProfile($T$)
14     return $\langle T.w, T.s, T.l, T.t \rangle$
```

Offline What-if Analyses Algorithm Description

Algorithm 10 outlines our approach to perform what-if analyses. It takes as input the performance model, and the list of regions annotated for what-if analyses. In a depth-first traversal of the performance model (line 2 in Algorithm 10), it computes the work (w), the critical path work (s), the set of step nodes on the critical path (l), and the total task creation work (t) at each internal node. The algorithm first computes the work from all the child nodes in the sub-tree rooted at the node (line 4 in Algorithm 10). Then, before computing the critical path work, the algorithm checks if there are any child step nodes that represent regions that are also present in the input list of regions annotated for what-if analyses (line 5 in Algorithm 10). If there are step nodes with annotated regions for what-if analyses, the algorithm reduces the work in the step nodes by the parallelization factor (line 6-8 in Algorithm 10). After reducing the work of the step nodes, the algorithm continues to compute the critical path work and the list of step nodes on the critical path (line 9 in Algorithm 10) using Algorithm 7. After the traversal of the entire performance model, the algorithm attributes the work, the critical path work, and the list of step nodes on the critical path to the spawn sites and generates the what-if profile that specifies the estimated parallelism of the program and at various spawn sites in the program after hypothetically parallelizing the regions identified for what-if analyses (line 13 in Algorithm 10).

Offline What-if Analyses Illustration

Figure 4.4 provides a concrete illustration of the what-if analyses algorithm on the performance model in Figure 4.3(b) with the region between lines 16 and 20 annotated for what-if analyses and a parallelization factor of 2. The figure shows the changes to the work, the critical path work, and the list of step nodes on the critical path as the algorithm traverses the nodes A2, A3, F2, and F0 in depth-first order. When the algorithm reaches node A2 while traversing the performance model, it has already processed node F1 which occurs before A2 in the depth-first traversal. This is reflected in the Figure 4.4(a) where the work, the critical path work, and the step nodes on the



(a)

Node	w	s	l
F0	0	0	{}
F2	0	0	{}
A2	660	330	{S2}
A3	0	0	{}
F1	1720	230	{S5,S10}

(b)

Node	w	s	l
F0	0	0	{}
F2	0	0	{}
A2	660	330	{S2}
A3	620	620	{S3}
F1	1720	230	{S5,S10}

(c)

Node	w	s	l
F0	0	0	{}
F2	1280	620	{S3}
A2	660	330	{S2}
A3	620	620	{S3}
F1	1720	230	{S5,S10}

(d)

Node	w	s	l
F0	3040	890	{S0,S1,S3, S5,S10}
F2	1280	620	{S3}
A2	660	330	{S2}
A3	620	620	{S3}
F1	1720	230	{S5,S10}

Figure 4.4: Illustration of the offline what-if analyses algorithm for the performance model in Figure 4.3(b) given annotated region L16-L20 and parallelization factor 2. The sub-tree under node F1 is not shown. Figures (a), (b), (c), and (d) show the updates to the work, critical path work, and list of step nodes on critical path as nodes A2, A3, F2, and F0 are traversed in depth-first order, respectively. The node being visited in each figure is highlighted with a double edge.

critical path of F1 have been computed. At node A2, the algorithm computes the total work from the work in the child step node S2. Before computing the critical path work at A2, the algorithm checks if of any of its step node children represent regions that have been annotated for what-if analyses. Since, S2 represent the region between lines 16 and 20 which has been annotated for what-if analyses, the algorithm reduces the work at S2 by a factor of 2 (parallelization factor) and then computes the critical path work and the list of step nodes on the critical path at A2. After traversing the async node A3 (Figure 4.4(b)), the algorithm updates the work and the critical path work at A3 from the work of the step node child S3. But, it does not have to perform what-if analyses since there are no child step nodes of A3 that represent regions annotated for what-if analyses. Similarly, the algorithm updates the work and the critical path work at nodes F2 and F0 and does not have to perform what-if analyses since no other step node in the performance model represents regions annotated for what-if analyses (Figures 4.4(c) and (d)). It is important to note that, as a result of reducing the critical path work of node S2, the critical path of the program shifted from passing through the node S2 to passing through the node S3. Hence, the critical path work of the entire program reduced by a negligible amount, which resulted in a minor increase in the program's parallelism.

4.2.3 On-the-fly What-If Profiling

In the on-the-fly mode, TaskProf performs what-if analyses as it constructs the performance model during profile execution. At the end of the profile execution, TaskProf produces the what-if parallelism profile that estimates the parallelism on hypothetically parallelizing the regions annotated for what-if analyses.

In the absence of the entire performance model, the key challenge in profiling for parallelism on-the-fly is to compute the parallelism even as nodes in the performance model that have completed are deallocated and removed. TaskProf addresses the challenge by summarizing the work and the critical path work of each node that completes at the parent node of the node. Similarly, to perform what-if analyses on-the-fly, TaskProf has to summarize the effect of performing what-if analyses for a node at its parent node.

When a step node completes execution, TaskProf has to check if the path through the completing step node forms the critical path in the sub-tree under the parent node. If the step node has been annotated for what-if analyses, then TaskProf mimics the effect of parallelizing the step node by first reducing the work of the step node and subsequently checking if the path through the completing step node forms the critical path in the sub-tree under the parent node. Specifically, when a step node that represents a region that has been annotated for what-if analyses completes, TaskProf has to update the work of the parent node using the entire work of step node, but update the critical path work of the parent after reducing the work of the step node by the parallelization factor. This has the same effect of performing what-if analyses when the entire performance model is available, where TaskProf performs what-if analyses on all the child step nodes that have been annotated for what-if analyses (lines 5-8 in Algorithm 10).

On-The-Fly What-If Analyses Algorithm Description

Algorithm 11 describes TaskProf's approach to perform what-if analyses on-the-fly as the program executes. It extends the on-the-fly parallelism computation algorithm shown in Algorithm 8 to perform what-if analyses. With each node in the performance model, the algorithm tracks five quantities, similar to the on-the-fly parallelism and task runtime overhead computation described in Algorithm 8. The quantities are, (1) the total work (w), (2) the set of spawn sites that perform the work on the critical path (CS), (3) the set of spawn sites that perform the serial work from step and finish children (SS), (4) the set of spawn sites that perform the serial work in the left step and finish siblings (LS), and (5) the total task creation work (t). TaskProf tracks the total task creation work since it uses the task creation work while automatically identifying the regions to parallelize (see Section 4.3).

Algorithm 11: On-the-fly what-if analyses given the set of regions annotated for what-if analyses RG.

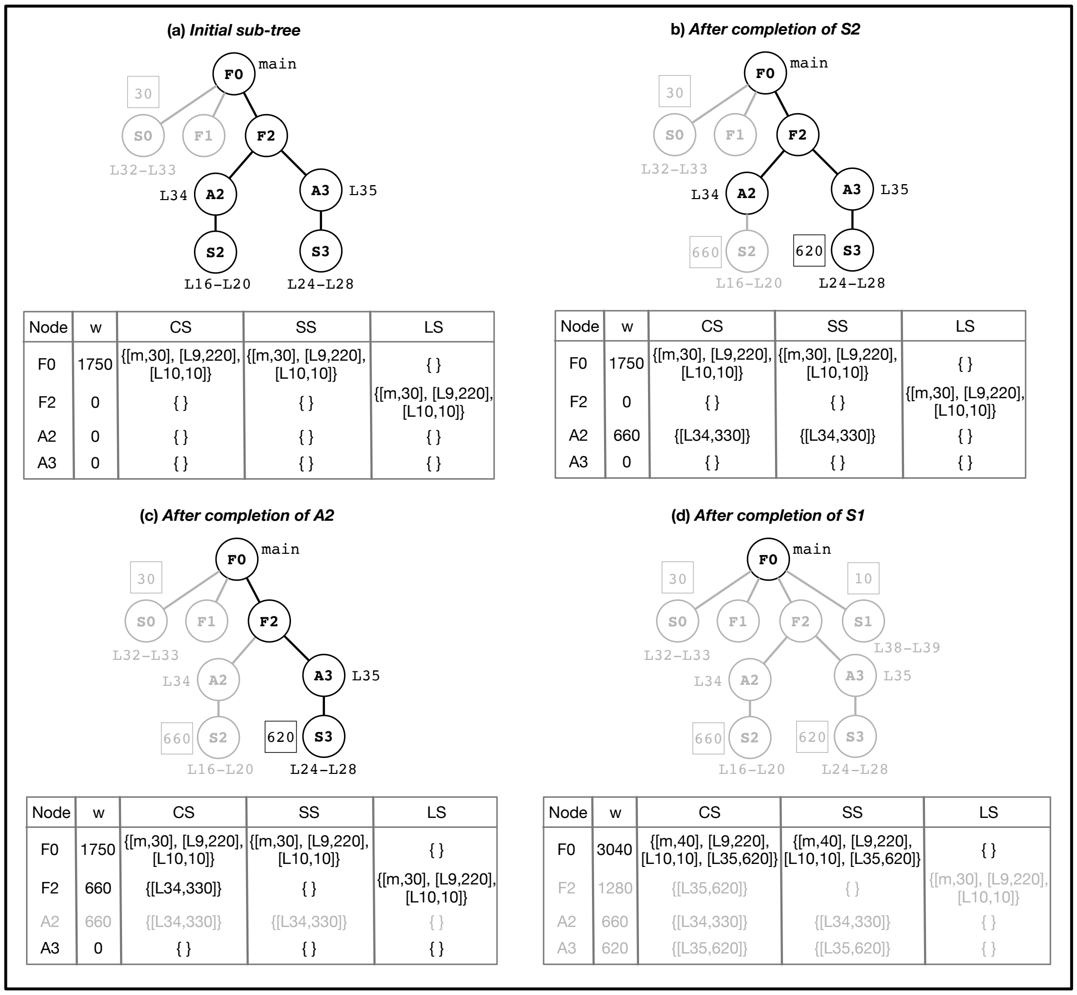
```
procedure OnNodeCreation($N$, $P$)
    $N.w \leftarrow 0$
    $N.t \leftarrow 0$
    $N.CS \leftarrow \emptyset$
    $N.SS \leftarrow \emptyset$
    $N.LS \leftarrow P.SS$
procedure OnNodeCompletion($N$, $P$, $RG$)
    $P.w \leftarrow P.w + N.w$
    if $N.r \in RG$ then
        $R \leftarrow \{R | (R \in RG \wedge N.r == R)\}$        ▷ r holds the source code region
        $N.w \leftarrow N.w / R.f$
    if $\sum_{L \in N.LS} L.w + \sum_{C \in N.CS} C.w > \sum_{S \in P.CS} S.w$ then
        $P.CS \leftarrow N.LS \uplus N.CS$
    if $N$ is an ASYNC node then
        AggregatePerSpawnSite($N$)
        $P.t \leftarrow P.t + N.c + N.t$
    else
        $P.SS \leftarrow P.SS \uplus N.CS$
        $P.t \leftarrow P.t + N.t$
```

When a node is added to the performance model, the algorithm copies the serial work of the parent node (SS) to the left serial work of the newly created node (LS) (line 6 in Algorithm 11). The newly created node can use the serial work from its left step and finish siblings to independently determine if it forms the critical path under its parent. When a node completes execution, the algorithm summarizes the work and the critical path work at the parent node, while performing what-if analyses if the completing node is a step node that has been annotated for what-if analyses.

First, it adds the work from the node to the parent node (line 8 in Algorithm 11). Then, before summarizing the critical path work the algorithm checks if the region represented by the node (variable r) has been annotated for what-if analyses (line 9 in Algorithm 11). It is implied that only step nodes in the performance model will have concrete regions represented in variable r. If the completing node represents a region identified for what-if analyses, the algorithm reduces the work of the region by the parallelization factor specified with the input region (lines 10-11 in Algorithm 11). Using the reduced work, the algorithm summarizes the critical path work at the parent node. It checks if the aggregate of the left serial work (LS) and the critical path work (CS) at the completing node is greater than the critical path work (CS) of the parent node. If so, the algorithm updates the spawn sites that perform the work on the critical path (CS) under the parent node with spawn sites performing the left serial work (LS) and the critical path work (CS) of the child node (line 13 in Algorithm 11). This updated critical path work in the parent reflects the reduced critical path work in the case where what-if analyses has been performed on the completing node. Finally, the algorithm updates the serial set of the parent node if the completing node is a step or a finish node (line 18 in Algorithm 11). If the completing node is an async node, the algorithm aggregates the work and critical path work at the static spawn site, which is eventually used to generate the what-if profile once the profile execution completes.

On-the-fly What-If Analyses Illustration

Figure 4.5 illustrates the on-the-fly what-if analyses algorithm on the performance model in Figure 4.3(b) with the region between the lines 16 and 20 annotated for what-if



(a) *Initial sub-tree*

Node	w	CS	SS	LS
F0	1750	{[m,30], [L9,220], [L10,10]}	{[m,30], [L9,220], [L10,10]}	{ }
F2	0	{ }	{ }	{[m,30], [L9,220], [L10,10]}
A2	0	{ }	{ }	{ }
A3	0	{ }	{ }	{ }

b) *After completion of S2*

Node	w	CS	SS	LS
F0	1750	{[m,30], [L9,220], [L10,10]}	{[m,30], [L9,220], [L10,10]}	{ }
F2	0	{ }	{ }	{[m,30], [L9,220], [L10,10]}
A2	660	{[L34,330]}	{[L34,330]}	{ }
A3	0	{ }	{ }	{ }

(c) *After completion of A2*

Node	w	CS	SS	LS
F0	1750	{[m,30], [L9,220], [L10,10]}	{[m,30], [L9,220], [L10,10]}	{ }
F2	660	{[L34,330]}	{ }	{[m,30], [L9,220], [L10,10]}
A2	660	{[L34,330]}	{[L34,330]}	{ }
A3	0	{ }	{ }	{ }

(d) *After completion of S1*

Node	w	CS	SS	LS
F0	3040	{[m,40], [L9,220], [L10,10], [L35,620]}	{[m,40], [L9,220], [L10,10], [L35,620]}	{ }
F2	1280	{[L35,620]}	{ }	{[m,30], [L9,220], [L10,10]}
A2	660	{[L34,330]}	{[L34,330]}	{ }
A3	620	{[L35,620]}	{[L35,620]}	{ }

Figure 4.5: Illustration of the on-the-fly what-if analyses algorithm for the performance model in Figure 4.3(b) given annotated region L16-L20 and parallelization factor 2. The sub-tree under node F1 is not shown. Figures (a) shows the initial sub-tree after S0 and F1 have completed and sub-tree under node F2 have been added. Figures (b), (c), and (d) show the updates to the work, the spawn sites tracking the critical path work, the serial work, and the left serial work after the completion of step node S2, async node A2, and step node S1. The nodes that have completed and their corresponding entries have been greyed out.

analyses and a parallelization factor of 2. The figure depicts the changes made to the four quantities - work (w) and spawn sites performing critical path work (CS), serial work (SS), and left serial work (LS) as the nodes S2, A2, and A1 complete execution. After step node S2 completes execution (Figure 4.5(b)), the algorithm first adds the work from S2 to the parent node A2. Before performing the computation for the critical path work, the algorithm reduces the work of S2 by a factor of 2 (parallelization factor), since S2 represents the region between the lines 16 and 20 that has been selected for what-if analyses. After reducing the work, the algorithm summarizes the critical path work under the parent node A2 by updating the spawn sites performing the critical path work with the reduced work from S2. When the async node A2 completes (Figure 4.5(c)) the algorithm updates the work at the parent node F2 and checks if the critical path under A2 performs the critical path work under F2. Since the critical path work at A2 is the highest critical path work under F2 when A2 completes, the algorithm updates spawn sites performing the critical path work at F2. Similarly, the algorithm updates the work and critical path work as nodes S2, A3 and F2 complete. Finally when the last step node S1 completes, the algorithm updates the critical path work of the root node F0 with the spawn sites performing the critical path work and the left serial work of S1(Figure 4.5(d)).

4.3 Identifying Regions to Parallelize with What-If Analyses

A key feature of what-if analyses is that the programmer can determine if parallelizing a region of code increases the parallelism, even before designing a concrete method to parallelize the region. The programmer may have some intuition on what regions of the program may potentially be parallelized. The programmer can use what-if analyses to check if the parallelism of the program will improve from potentially parallelizing those regions. Identifying regions that increase the parallelism enables the programmer to focus parallelization efforts on regions that matter.

In many cases, the programmer may have no intuition about the parts of the program that can potentially be parallelized. In such cases, the programmer would like to identify the regions that must be parallelized to attain a parallelism that is necessary to achieve scalable speedup on the execution machine. To identify regions that can increase the parallelism, one approach is to identify regions that are performing the highest amount of work in the entire program. But, recall from the discussion in Section 4.1 and Figure 4.1, parallelizing the regions performing the highest work may not increase the parallelism of the program if the regions are not executing on the critical path. To increase the parallelism of the program, the programmer has to parallelize regions that reduce the work on the critical path of the program.

The programmer can use the parallelism profile to identify a region that likely executes on the critical path and perform what-if analyses to estimate the parallelism on parallelizing the region. But this may not reduce the work on the critical path if the identified region is not executing on the critical path, or the critical path shifts to another parallel path that performs slightly lesser work than the original critical path. Even if the work on the critical path reduces, the estimated parallelism may not be sufficient to attain scalable speedup on the execution machine. In either case, the programmer has to continue to identify regions that likely execute on the critical path, and perform what-if analyses to estimate the parallelism from hypothetically parallelizing the regions.

Consider for example, the what-if profile in Figure 4.3(c) that specifies the estimated parallelism after performing what-if analyses using the region between lines 16 and 20. After what-if analyses, the parallelism of the program increased by a negligible amount. This means that the programmer has to continue to find other regions in the program and perform what-if analyses to identify all the regions that must be parallelized to reduce the work on the critical path of the program. The entire process can be challenging and time consuming, especially while profiling large programs. Further, while the parallelism profile provides information on the spawn sites that execute on the critical path, the programmer has to use ingenuity to identify the specific regions within the spawn sites that execute on critical path. Hence, a natural question that arises here is - *is it possible to apply what-if analyses and automatically identify a set of regions that must be parallelized to increase the parallelism in the program?*

Using what-if analyses, we design a technique to automatically identify a candidate set of regions in the program, that have to be parallelized to increase the parallelism in the program. The programmer can focus parallelization efforts on the regions reported by TaskProf to increase the parallelism. To automatically identify regions in the program that must be parallelized, we must first clearly define what constitutes a *region*. While profiling a program for what-analyses, the regions specified by the programmer as input could constitute the entire part of the program between task management constructs that are represented by step nodes in the performance model, or even smaller chunks of code within the parts represented by step nodes. However, without programmer intervention, it would not be possible to define regions as small chunks of code within parts represented by step nodes that can potentially be parallelized. To automatically identify regions to parallelize, we use TaskProf's performance model where we have step nodes that represent sequence of instructions in the program bounded by task management constructs. The parts of the program represented by step nodes provide a natural way to define a region. TaskProf identifies the parts of the program between task management constructs that must be parallelized to increase the parallelism in the program.

To automatically identify regions to parallelize, we have designed an iterative process where in each step we attempt to increase the parallelism by selecting a region to perform what-if analyses. We continue the iterative process until we reach a parallelism level that is sufficient to expect scalable speedup on a given execution machine. This anticipated parallelism is set by the programmer. One approach to increase the parallelism of the program is by reducing the critical path work of the program. Hence in each step of the iterative process, we identify the dynamic region that performs the highest amount of work on the critical path. In the performance model, the dynamic region is the step node that performs the highest work on the critical path. Then, we perform what-if analyses to parallelize the region corresponding to the highest step node on the critical path, and estimate the parallelism of the program. If the estimated parallelism is lesser than the anticipated parallelism specified by the programmer, we repeat the process. We again find the step node performing the highest work on the updated critical path, and perform what-if analyses to estimate the parallelism after hypothetically parallelizing the region corresponding to the highest step node. The set of regions that are selected for what-if analyses in each step of this iterative process, are the regions that have to be parallelized to increase the parallelism of the program.

Theoretically, one can increase the parallelism of the program to any level by reducing the critical path work of the program. However, beyond a certain threshold, increasing the parallelism will have very little effect on the speedup of the program and may even adversely affect the speedup because the program would perform more work in creating tasks than performing useful computation in the tasks. Hence, the task runtime overhead would begin to dominate the execution. To provide a realistic view of possible parallelization opportunities, we factor the task runtime overhead into the iterative process to identify regions to parallelize. Specifically, we compute the average task runtime overhead in the program using the performance model and the total task creation work. Then, in every step of the iterative process we check if the work performed by the step node that is selected becomes lesser than a tasking threshold determined for the execution machine after what-if analyses. If so, we terminate the iterative process even if the anticipated parallelism is not reached, since further parallelization of the regions on the critical path will result in the task runtime overhead dominating the execution. The offline and on-the-fly what-if analyses algorithms (Algorithms 10 and 11) track the total task creation work so that the task runtime overhead can be factored while identifying regions to parallelize.

TaskProf can automatically identify regions to parallelize, while profiling the program in both offline and on-the-fly modes. In the offline profiling mode, TaskProf uses the performance model from a single execution to repeatedly perform what-if analyses and identify regions that must be parallelized. In the on-the-fly profiling mode, TaskProf performs what-if analyses on-the-fly as the program executes. Hence, to identify regions to parallelize, TaskProf executes the program repeatedly to perform what-if analyses. In each such re-execution, the serial work for all the identified regions from the previous iterations are reduced during parallelism computation on node completion, which mimics the effect of parallelization. Irrespective of the profiling modes, TaskProf employs the same iterative technique to identify regions until the parallelism reaches anticipated parallelism or the task runtime overhead reaches a threshold. Next, we describe in detail our algorithm to identify a set of regions that have to be parallelized to improve the parallelism in the program.

Description of Algorithm to Identify Regions to Parallelize

Algorithm 12: Identify regions to parallelize by parallelization factor pf to increase parallelism to an anticipated level ap in a program with performance model T.

```
function WhatIfRegions($T$, $ap$, $pf$)
    $RG \leftarrow \emptyset$
    $R \leftarrow \emptyset$
    $ep \leftarrow 0$
    $smax \leftarrow 0$
    $tt \leftarrow 0$
    while $(ep < ap) \wedge (smax \geq tt \times pf)$ do
        $RG \leftarrow RG \uplus \langle R, pf \rangle$
        $\langle w, s, l, t \rangle \leftarrow$ WhatIfProfile($T, RG$)
        $ep \leftarrow w/s$
        $\langle R, smax \rangle \leftarrow$ MaxStepOnCritPath($l$)    ▷ Returns region and work
        $tt \leftarrow k \times$ AvgTaskOverhead($t$)    ▷ Average task creation work
    return $RG$
```

Algorithm 12 presents our technique to identify all static regions in the program that need to be parallelized. It takes as input the performance model of the program, an anticipated parallelism that the program is expected to have, and a parallelization factor that determines amount of parallelization feasible for a region in each iteration (line 1 in Algorithm 12). We present the algorithm with the assumption that it takes the entire performance model as input. But in the on-the-fly profiling mode, the performance model will not be available. In the on-the-fly mode, the algorithm takes the program itself as input and executes the program to identify regions to parallelize by constructing the performance model on-the-fly. Further, when the algorithm uses the performance model to perform what-if analyses, the on-the-fly mode executes the program and perform what-if analyses on-the-fly.

The algorithm tracks four quantities in every iteration. They are, (1) the set of regions that have been selected for what-if analyses (RG), (2) the static region (R) and amount of work ($smax$) of the step node performing the maximum work on the critical path, (3) the estimated parallelism from what-if analyses (*ep*), and (4) the tasking threshold (*tt*).

In each iteration, the algorithm performs what-if analyses to compute the estimated parallelism from hypothetically parallelizing all the regions in the set of regions that have been selected for what-if analyses (lines 9-10 in Algorithm 12). In the first iteration, the set of regions is empty. Hence, the algorithm computes the original parallelism in the program, without performing any what-if analyses. Along with finding the estimated parallelism the algorithm also computes the static region and the work on the step node performing the maximum work on the critical path (line 11 in Algorithm 12).

In the offline analyses, the algorithm computes the maximum step node on the critical path from the set of step nodes on critical path (l) that is computed. The on-the-fly analyses does not compute the set of step nodes on the critical path. To compute the maximum step node on the critical path in the on-the-fly analyses, TaskProf tracks the maximum step node similar to the spawn sites on the critical path in Figure 11.

The algorithm also computes the average task runtime overhead from the total task creation work computed during both offline and on-the-fly what-if analyses. It uses the average task runtime overhead, along with an execution machine dependent constant (k) to determine the tasking threshold (line 12 in Algorithm 12). After every iteration the algorithm checks if the estimated parallelism from what-if analyses is lesser than the anticipated parallelism specified by the programmer. It also checks if the work performed in the maximum step node on the critical path is greater than or equal to the tasking threshold when reduced by the parallelization factor (line 7 in Algorithm 12). If so, the algorithm adds the region corresponding to the maximum step node on the critical path, to the set of regions that have been selected for what-if analyses (RG) and repeats the entire process. The process continues until either the anticipated parallelism is reached or the work performed by every node on the critical path is less than the tasking threshold. If the anticipated parallelism is not achieved when our algorithm terminates, it indicates that the program does not have sufficient parallelism for a given input. Finally, the algorithm outputs the set of regions that were considered in each

(a) Illustration of algorithm to identify regions to parallelize

Iteration	ep	<R,smax>	tt	RG
0	0	<0,0>	0	{ }
1	3.26	<L16-L20, 660>	105	{ }
2	4.49	<L24-L28, 620>	105	{<L16-L20,2>}
3	5.06	<L16-L20, 330>	105	{<L16-L20,2>, <L24-L28,2>}
4	5.24	<L24-L28, 310>	105	{<L16-L20,4>, <L24-L28,2>}

(b) What-if analyses regions

Regions	Parallelization factor
L16-L20	4X
L24-L28	2X

(c) What-if Parallelism profile

Line number	Work	Span	Parallelism	Critical work percent
main	3040	580	5.24	6.90
L9	1480	650	2.28	37.93
L10	1500	650	2.31	1.72
L34	660	165	4	0
L35	620	310	2	53.45

Figure 4.6: (a) Step-by-step execution of the algorithm to automatically identify regions to parallelize for the example program and performance model in Figure 4.2. The illustration shown is for an input anticipated parallelism of 8, parallelization factor of 2×, and tasking constant k of 3. (b) The regions along with their respective parallelization factors, identified by TaskProf to improve the parallelism in the program. (c) The what-if profile generated by TaskProf.

iteration as the regions that need to be parallelized to improve the parallelism of the program (line 13 in Figure 12).

Illustration of Algorithm to Identify Regions to Parallelize

Figure 4.6(a) provides a step-by-step illustration of the algorithm to identify regions to parallelize in the example program shown in Figure 4.2(a). The input anticipated parallelism is set to 8, the parallelization factor is set to 2×, and the tasking constant k is set to 3. The illustration shows the updates performed to the four quantities tracked by the algorithm for each step in the iterative process. The algorithm starts by setting all the four quantities to zero or empty set. Hence, in the first iteration, the algorithm performs parallelism computation with no regions selected for what-if analyses. In the first iteration, the algorithm computes the parallelism in the program to be 3.26 and identifies the region between lines 16 and 20 as the region corresponding to the maximum step node on the critical path. Since the parallelism of the program is lesser than the anticipated parallelism of 8, in the next iteration, the algorithm performs what-if analyses using the region between lines 16 and 20 and a parallelization factor of 2. It reduces the contribution of the region between the lines 16 and 20 to the critical path work of the program by a factor of 2. The estimated parallelism after what-if analyses is still lesser than the anticipated parallelism set as input. Hence, the algorithm continues find the maximum region of the critical path and perform what-if analyses.

After four iterations the algorithm discovers that performing further what-if analyses to reduce the critical path work would result in the task runtime overhead dominating the execution, since reducing the work of the maximum step node on the critical path would make the work lesser than the tasking threshold. Hence the algorithm terminates the iterative process before the estimated parallelism finally converges to the anticipated parallelism. After the entire process, the algorithm identifies two regions in the program that have to be parallelized to attain a parallelism of the program to 5.24. This is reflected in the what-if regions and what-if parallelism profile shown in Figures 4.6(b) and (c), respectively. It is important to point out that in iteration 4, the algorithm identified a region that is already present in the set of regions. In such a case, the algorithm increases the parallelization factor for the specific region.

4.4 Discussion

The algorithm we propose to automatically identify regions to parallelize is a greedy algorithm where in each step, we select the step node that performs the maximum work on the critical path. We also considered several other heuristic-based approaches to select a region to parallelize. For instance, we considered randomly selecting a step node region from the critical path in each step of the iterative process. But, randomly selecting a region on the critical path does not guarantee that all the identified regions matter in improving the parallelism of the program. Some of the selected regions may not have to be parallelized to increase the parallelism of the program. In contrast, by selecting the step node region that performs the maximum work on the critical path, our algorithm guarantees that all the regions identified have to be parallelized to increase the parallelism of the program. Further, by selecting the maximum step node in each iteration, our algorithm is optimal with respect to the least number of iterations performed to identify regions to parallelize.

The set of regions identified by our algorithm can be one among the several other sets of regions in the program that when parallelized attain the same parallelism level. But, the set of regions identified by our algorithm are minimal in terms of the least number of regions identified, and maximal in terms of the highest amount of work reduced to attain the set parallelism level. In many programs, there can be other regions that when parallelized increase the parallelism to the level set by the program. The programmer may even have some intuition on how to parallelize those regions. In such a scenario, the programmer can use our annotation-based what-if analyses to determine if the region that can be parallelized increases the parallelism in the program. Our what-if analyses using annotations and our what-if analyses to automatically identify regions to parallelize are complementary techniques that enable the programmer to determine the regions in the program that have to be parallelized.

Our what-if analyses technique assists the programmer in identifying a set of regions that benefit from parallelization. However, it does not determine if the identified regions can be parallelized. The programmer has to design a parallelization strategy for the regions identified by what-if analyses. Further, a region identified by what-if analyses may not be parallelizable due to the presence of dependencies within the region. If the regions reported by what-if analyses are not parallelizable, the programmer will have to manually identify regions that are parallelizable and use what-if analyses to check if those regions matter in increasing the parallelism of the program.

Parallelizing the regions identified by what-if analyses will improve the parallelism of the program. But, the parallelism of a program is an over approximation of the speedup of the program since parallelism defines the speedup of a program in the limit. The speedup of a given execution of the program may not improve after parallelization if the program already has adequate parallelism or if the parallel execution experiences overhead due to secondary effects. However, the programmer will still benefit from knowing the regions to parallelize, especially to address scalability bottlenecks that might occur when the program is executed at scale.

Our what-if analyses mimics the parallelization of a region of code in a homogenous execution environment where all cores run at similar clock speeds. It divides the serial work in the region by a constant parallelization factor and ensures that each parallel chunk performs the same amount of work. However, our what-if analyses does not does not mimic parallelization in the presence of heterogeneity both at the micro-architecture level with cores running at different clock speeds [93] (for *e.g.*, ARM big.LITTLE [102]) and at the system level with customized chips. We plan to explore what-if analyses in the context of heterogenous execution environments as future work.

4.5 Summary

When programs do not have sufficient parallelism to achieve scalable speedup, programmers rely on profilers to identify parts of the program that must be parallelized to increase the parallelism of the program. Most existing profilers are useful in highlighting the "hot" code, which are the parts of the program that perform the most amount of work in the program. In this chapter, we argue that identifying regions where the program performs the most work may not be sufficient to identify the parts of the program that must be parallelized. To identify parts of the program that matter in increasing the parallelism, we make a case for identifying regions that reduce the critical path work rather than regions that perform the most work. To accomplish it, we have designed what-if analyses that estimate the parallelism of the program on hypothetically parallelizing regions of code.

The key insight in our what-if analyses technique is that while computing the parallelism of a program, we mimic the effect of parallelizing a piece of code by computing the total work of the code as is, but reducing the critical path work of the code by a parallelization amount. If the parallelism of the program increases on performing what-if analyses, then concretely parallelizing the designated piece of code will improve the parallelism of the program. Further, we showed that what-if analyses can be applied to identify a set of regions that can be parallelized to increase the parallelism of the program.

Our what-if analyses technique can predict from a single execution, the program regions that must be parallelized to improve the parallelism to a level that is needed to achieve scalable speedup on any machine. This enables programmers to identify the scalability bottlenecks in the program and focus on program regions that matter in addressing them. The combination of what-if analyses and automatically identifying regions to parallelize enable effective performance debugging by quickly identifying the parts of the program that matter for increasing the parallelism of the program.

Chapter 5

Identifying Secondary Effects using Differential Performance Analysis

Beyond parallelism and task runtime overhead, task parallel programs can experience performance degradation due to secondary effects of parallel execution that manifest when parallel tasks contend for hardware and system resources. Secondary effects are perhaps the hardest performance issues to identify and debug since they can occur due to interference in various resources. In this chapter, we present a novel differential analysis technique that uses TaskProf's performance model to highlight parts of the program that are experiencing secondary effects as well as the likely source of the secondary effects.

5.1 Secondary Effects in Parallel Execution

Task parallel programs are expected to have sufficient parallelism and low task runtime overhead to achieve scalable performance. If a program has low parallelism, then creating additional fine-grain tasks can create more parallelism in the program to keep all the processors busy during execution. In contrast, if a program has high task runtime overhead, then coarsening the tasks can reduce the overhead. However, even if a task parallel program has sufficient parallelism and low runtime overhead, the parallel execution on hardware itself can adversely affect the performance of a program.

Modern computer systems are highly complex with large number of processor cores and deep memory hierarchies comprising multiple levels of caches and memory. In many modern systems, the processor cores are placed on multiple sockets with each socket having a directly attached memory resulting in non-uniform memory access latencies (NUMA). In addition, modern systems are further complicated by custom hardware accelerators that have their own memory hierarchy. Given the complexity, contention for hardware resources and frequent data movement across multiple sockets or accelerators can degrade the performance of a program. For instance, frequent contention for cache blocks causes excessive coherence traffic that affects the performance. Similarly, frequent remote memory accesses that cause data movement across sockets or accelerators also hurts the performance. We call such performance issues that are caused by contention in hardware and system resources as *secondary effects*.

While secondary effects can occur during a sequential execution of a program, in this chapter we explore the problem of secondary effects that primarily occur due to a parallel execution of a program. For instance, consider a program that spawns a set of parallel tasks where each task accesses a disjoint dataset during execution. Assume that while each dataset is small enough to fit in the cache, the combined dataset of all tasks does not fit in the cache. Now, if each task is executed sequentially, the dataset of the task fits in the cache and the program has good locality. But when all the tasks execute concurrently, the combined dataset will not fit in the cache. In such a scenario, the parallel execution will have poor locality, which causes frequent cache evictions and affects the performance of the program. Similarly, a program may have memory access patterns during parallel execution that cause frequent data movement across private caches due to true or false sharing. In a multi-socket machine, a program that lacks NUMA-aware design can suffer from excessive remote memory accesses and contention on the memory bus during parallel execution. Such patterns that occur during parallel execution can reduce the performance of the program.

Task parallel programs in particular are vulnerable to secondary effects of parallel execution. The work-stealing scheduler maps tasks to executing threads whenever a thread becomes idle. But due to the randomized mapping of tasks to threads, a task can be mapped to a particular thread on a socket, whereas the data that it operates on might exist in the memory of a different socket. This lack of affinity especially in NUMA systems can increase remote memory accesses and impact the performance of task parallel programs.

Manually detecting such secondary effects in programs is challenging since secondary effects are primarily caused by interactions in hardware and system resources. Consider for instance, the example task parallel program shown in Figure 3.2(a). In the `compute` function (lines 1-13 in Figure 3.2(a)), the program recursively spawns tasks until the computation in the tasks reaches a pre-defined granularity (`GRAINSIZE`). Then, each parallel base task performs a computation and writes the result to an array. At the outset, this seems like a reasonable strategy to parallelize the computation. However, the computation is actually experiencing secondary effects due to false sharing in the cache. Although the program has sufficient parallelism, it will not have scalable performance due to the presence of secondary effects of parallel execution. Similarly, secondary effects can occur due to contention in any hardware component, which makes them hard to detect. Moreover, such secondary effects manifest only in specific executions, sometimes intermittently when tasks that cause the secondary effects are scheduled concurrently.

There have been numerous tools that have been proposed to identify specific types of secondary effects. They include tools to detect cache contention [34, 39, 60, 87, 103, 104, 112, 127, 165, 182], identify data locality issues [105–107, 110], and find bottlenecks due to NUMA effects [108, 115]. These tools are tailored to detect particular kinds of secondary effects. But, secondary effects can practically occur due to any hardware or system resource. Hence, we need tools that can detect secondary effects that can occur due to contention in any hardware component. Further, while secondary effects can occur in any part of a program, not all secondary effects impact the performance of the program. Optimizing secondary effects from parts of the program that perform a small fraction of the total work or the work on the critical path will not improve the performance of the program. Hence, reporting such secondary effects is not useful to the programmer. Instead, we need tools that can assess the impact of the secondary effects on the total work and the critical path work of the program and highlight only those that matter in improving the performance.

5.2 Overview of our Approach

We have developed a technique to identify parts of a task parallel program that are experiencing secondary effects due to contention in any hardware or system resource. A program that is experiencing secondary effects performs more work in the parallel execution compared to an oracle execution [8, 111, 131]. Using this insight, our idea is to use TaskProf's performance model to identify regions in the program that are performing additional work in the parallel execution compared to the oracle execution. The performance model measures the total work performed in the program, as well as fine-grain work performed in various parts of the program. Now, if we perform a fine-grain comparison of the performance model for the parallel execution with the performance model of the oracle execution, the step nodes in the two models that are experiencing secondary effects will have higher work in the performance model of the parallel execution than in the performance model of the oracle execution. We call this analysis to compare the performance models to identify regions experiencing secondary effects as *differential analysis*. By comparing the work recorded in the two performance models, our differential analysis can identify regions that are experiencing secondary effects due any hardware or system resource. Section 5.3 describes in detail TaskProf's approach to perform differential analysis.

Along with pinpointing regions in the programs experiencing secondary effects, gaining insights into the specific hardware or system resource that is causing secondary effects can guide a programmer in designing appropriate optimizations. To provide the programmer with such insights, TaskProf performs differential analysis not only with the work measured in hardware execution cycles, but also with any hardware performance counter event supported on the execution machine. For instance, to identify regions experiencing secondary effects due to false sharing in the program in Figure 3.2, TaskProf can perform differential analysis with the HITM performance counter event, which measures the number of cache hits in the modified state. Similarly, we can identify regions experiencing reduced locality in shared caches due to parallel execution by measuring last level cache misses. Hence, TaskProf's differential analysis enables the programmer to

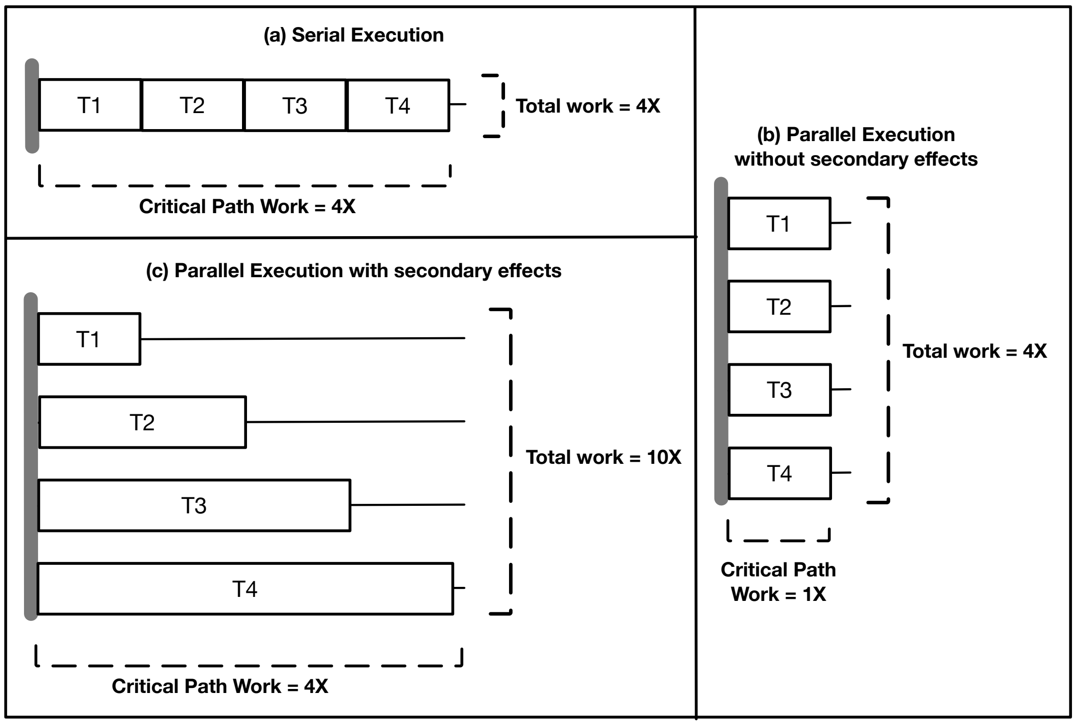


Figure 5.1: Illustration showing the inflation in work due to secondary effects. Figure (a) shows a timeline of the serial execution of four tasks where each task performs `X` units of work. Figure (b) shows a timeline of the parallel execution of the same four tasks without secondary effects. Figure (c) shows the timeline of a parallel execution that is experiencing secondary effects. Each figure highlights the total work and the work on the critical path. The parallel execution with secondary effects performs more work that both the serial execution, and parallel execution without secondary effects.

identify regions experiencing secondary effects and also the likely source of the secondary effects. Section 5.4 details our technique to perform differential analysis with multiple hardware performance counter events.

To perform differential analysis, TaskProf constructs two performance models - one for the oracle execution and other for the parallel execution - for multiple performance counter events of interest. Using the performance models, TaskProf computes the total inflation and the inflation on the critical path of each event of interest. TaskProf computes the inflation as the ratio of the work in the parallel execution and the work in the oracle execution. Along with the inflation of a metric of interest for the entire program, TaskProf can localize the inflation of the metric at various parts of the program at two levels. TaskProf can either aggregate the inflation in a metric of interest at each static spawn site in the program or at each static region in the program between task management constructs represented by the step nodes in the performance model. By measuring the total inflation and the inflation on the critical path of a metric of interest, TaskProf highlights the secondary effects that affect the performance of the program. TaskProf finally generates a profile called the *differential profile*, which specifies the total inflation and the inflation on the critical path of each metric of interest for the entire program and for each spawn site or static program region in the program. Figure 5.2(c) shows a sample differential profile generated for the program in Figure 3.2(a).

5.2.1 Contributions

In this chapter, we make the following contributions in identifying regions experiencing secondary effects.

1. We present a technique to determine if a program execution is experiencing secondary effects and the possible source of secondary effects by measuring inflation in work and in various metrics of interest for the parallel execution over the serial execution.

2. We highlight the specific regions in the program that are experiencing secondary effects by performing fine-grain differential analysis using the parallel and serial performance models.

3. We identify code regions that matter for addressing secondary effects by highlighting regions that are having work inflation on the critical path.

5.3 Profiling for Differential Analysis

A common approach to parallelize a serial program is to perform the same computation as the serial program, but to split the computation on to tasks or threads and execute them in parallel. The expectation is that after parallelization, the parallel version of the program will perform the same amount of work as the serial version of the program, but the work on the critical path will reduce. This will enable the parallel program to have scalable speedup since the execution time will also reduce as the critical path reduces. However, this is true only when the parallel program is not experiencing secondary effects. When a program is having secondary effects, the parallel version performs more work than the serial version and this can degrade the speedup of the program.

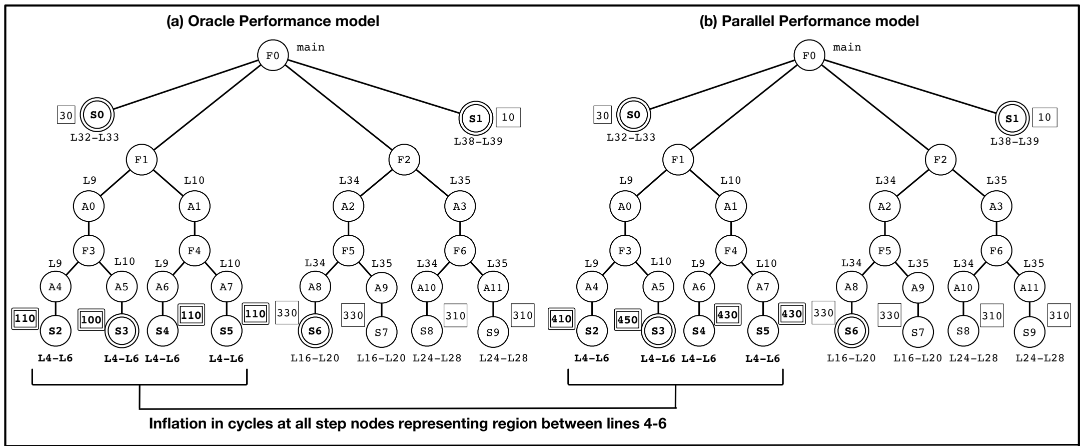
To illustrate how secondary effects cause work inflation in the parallel execution, consider a task parallel program that has four tasks where each task performs `X` units of work. Figure 5.1 shows the execution of the program when the tasks are executed serially, in parallel without secondary effects (ideal parallel execution), and in parallel with secondary effects. When these tasks are executed serially one after another as shown in Figure 5.1(a), the total work performed in the program will be the sum of the work of all the tasks, which is `4X`. When the same four tasks are executed in parallel on a machine where they are not experiencing secondary effects, the tasks will perform the same amount work (`4X`) overall, but the critical path work reduces to `1X`, as shown in Figure 5.1(b). Accordingly, the ideal parallel execution will have scalable speedup over the serial execution of the program.

When parallel tasks are experiencing secondary effects from accessing the same resource concurrently, the tasks have to wait for the resource to become free or perform extra work to ensure consistency. Figure 5.1(c) shows the parallel execution of the same program on a machine where the program is experiencing secondary effects due to contention for a resource. In the parallel execution in Figure 5.1(c), task T1 gets first access to the contended resource and does not perform extra work. But, task T2 has to wait for task T1 to complete accessing the hardware resource and hence, performs extra work by either waiting or moving data after T1 completes. Similarly, the rest of the tasks also perform additional work to access the contended resource. This causes the parallel execution with secondary effects to perform `10X` amount of work whereas the serial execution and the ideal parallel execution perform just `4X` work. In addition to the inflation in the total work, the program also experiences inflation in the critical path work with respect to the ideal parallel execution. Hence, a program that is experiencing secondary effects will perform additional total work and work on the critical path than the serial execution and the ideal parallel execution.

As work inflation in a program is an indicator of the presence of secondary effects, we can specifically measure the work and check for inflation in the program to determine if it is experiencing secondary effects. One approach is to measure the overall work performed in both the parallel and serial programs. If there is any inflation in the work, then that could indicate the presence of secondary effects. Prior tools [8, 131] used this approach to identify secondary effects in parallel programs. While knowing if a program is experiencing work inflation is useful in understanding the reason for sub-optimal speedup in the program, it is not useful in gaining insight into the parts of the program that are experiencing secondary effects. Hence, to provide actionable feedback to the programmer we need a way to isolate the parts of the program that are experiencing secondary effects.

Our idea is that we can use TaskProf's performance model to identify parts of the program that are experiencing secondary effects. When a program is experiencing secondary effects, not only will the total work and the critical path work in the program show inflation, the specific parts of the program that are causing the secondary effects will also show inflation in total work and work performed on the critical path. The performance model we designed for TaskProf records the work performed in fine-grain regions of the program between task managements constructs at the corresponding step nodes in the performance model. Consequently, the performance model of the parallel execution of the program that is experiencing secondary effects will record the inflated work in the step nodes. However, by just inspecting the performance model of the parallel execution, we will not be able to distinguish the step nodes that have work inflation from the step nodes that do not have work inflation. Instead, we can isolate the step nodes by comparing the performance model of the parallel execution with the performance model of an oracle execution that is not experiencing secondary effects.

Consider for instance, the performance models shown in Figure 5.2(a) and (b), generated from an oracle execution without secondary effects and a parallel execution of the task parallel program in Figure 3.2(a), respectively. The structure of the performance



(c) Differential Profile

Region	Inflation cycles		Inflation HITM		Inflation loc DRAM		Inflation rem DRAM	
	w	cp	w	cp	w	cp	w	cp
main	**1.74X**	**1.74X**	**23X**	**28X**	1X	1X	1X	1X
L4 - L6	**4X**	**4.5X**	**41X**	**46X**	1X	1X	1X	1X
L16-L20	1X	1X	1.02X	1.03X	1X	1X	1X	1X
L24-L28	1X	0	1X	1X	1X	1X	1X	1X
L32-L33	1X	1X	1.01X	1.01X	1X	1X	1X	1X
L38-L39	1X	1X	1X	1X	1X	1X	1X	1X

Figure 5.2: (a) and (b) represent the performance model generated from the oracle and parallel executions of the program in Figure 3.2 (a) after parallelizing the regions identified by what-if analyses in Figure 4.6 (b) and reducing task runtime overhead identified in Figure 3.2 (c). The step nodes on the critical path of the parallel performance model and the corresponding path in the oracle performance model are highlighted with double edges. The work in the step nodes having secondary effects are highlighted with double-edge boxes. (c) The differential profile showing inflation of various metrics in parallel execution over oracle execution. We show four hardware performance counter event types: execution cycles, HITM, local DRAM accesses, and remote DRAM accesses.

model including the numbers and types of the nodes and the series-parallel relations is the same for both the oracle execution and the parallel execution, assuming the program is race-free. The work recorded in the step nodes of both the performance models is almost the same in all the step nodes, except for four step nodes S2, S3, S4, and S5. These four step nodes, show an inflation in the work in the parallel performance model over the oracle performance model, indicating the presence of secondary effects in the region represented by the step nodes, which is between line 4 and line 6 in Figure 3.2(a).

The oracle performance model has to be generated from an execution that is not experiencing secondary effects. Further, to enable us to perform differential analysis, the structure of the oracle performance model has to be the same as the structure of the parallel performance model. As discussed earlier in this section, one way to generate the oracle performance model is from the parallel execution of the program on a machine where the program is not experiencing secondary effects. But there is no practical way to guarantee that the parallel execution on the different machine does not introduce other secondary effects. Further, the memory access patterns in the program could be such that any parallel execution will experience secondary effects irrespective of the machine of execution. Hence, such a strategy would not be feasible.

The serial execution of the parallel program on the same machine is a good approximation of the oracle performance model. During the serial execution of the parallel program, the parallel tasks created in the program will execute sequentially on the same thread without any interference. As a result, there will be no contention for any hardware resource and the execution will not experience any work inflation. Further, serial execution of the program will create the same parallelism by spawning the same number of parallel tasks as the parallel execution. This ensures that the structure of the performance model of the serial execution is the same as the structure of the performance model of the parallel execution. Hence, to isolate regions experiencing secondary effects, TaskProf's differential analysis performs fine-grain comparison of the performance model of the parallel execution with the performance model of the serial execution.

TaskProf can perform differential analysis in both the offline and on-the-fly profiling modes. In both the modes, TaskProf executes the program twice to generate the performance models of the serial and parallel executions. In the offline profiling mode, TaskProf writes the performance models from both the executions to a profile data file. Then in an offline post-mortem analysis, TaskProf re-constructs the two performance models and performs differential analysis by traversing the models concurrently.

To perform differential analysis in the on-the-fly mode, TaskProf does not perform the serial and parallel execution of the program simultaneously. In both the executions, instead of writing the entire performance model to a file, TaskProf aggregates the work performed at each static spawn site or static region of code and writes the data to a profile data file. Then, in an offline analysis TaskProf reads the individual files for each execution, computes the work inflation for each static spawn site or region, and generates the differential profile.

In the remainder of this section we will describe in detail TaskProf's offline and on-the-fly differential analysis algorithms.

5.3.1 Offline Differential Analysis

In the offline differential analysis mode, TaskProf first constructs the performance model of the serial and the parallel executions from the profile data file. Then, using the performance models, TaskProf performs differential analysis by computing the inflation in the total work and the work on the critical path for the entire program and for each static source location associated with the step nodes in the performance model. A static source location can be the static region represented by a step node or a spawn site associated with the step node.

To compute the overall work inflation and the critical path work inflation for the entire program, TaskProf has to measure the total work and the critical path work from both the serial and parallel performance models. Similarly, to compute the work inflation and the critical path work inflation associated with each static source code location, TaskProf has to compute the total work performed at each source code location and the work performed by each location on the critical path from both performance models.

For example, consider the serial (oracle) and the parallel performance models shown in Figure 5.2(a) and (b). To compute the overall work inflation and the critical path inflation for the program, TaskProf needs to compute the total work in all the step nodes and the work in step nodes executing on the critical path from both the performance models. To compute the work inflation and the critical path work inflation at the region between lines 16 and 20, TaskProf has to compute the total work performed by the region and the work performed by the region on the critical path in both performance models. Since only the step nodes S6 and S7 represent the region between the lines 16 and 20, the total work performed by the region is the aggregate of the work from the two step nodes, S6 and S7 in each performance model. The work on the critical path of the region between lines 16 and 20, is the sum of all the step nodes that correspond to the region and perform work on the critical path. Since the step node S6 executes on the critical path, the work from S6 is the work on the critical path of the region between lines 16 and 20.

Overall, TaskProf computes four quantities for both the serial and the parallel performance models separately. They are - (1) the total work, (2) the total critical path work, (3) the work at each static code location associated with step nodes, and (4) the work performed on the critical path by each static code location. TaskProf can compute the total work in each performance model by aggregating the work from all the step nodes in the respective performance model. In contrast, computing the critical path work from the performance models is more involved than merely aggregating the work from all the step nodes on the critical paths of the two performance models.

If the critical paths are guaranteed to follow the same path in both the performance models then TaskProf can identify the critical path in the parallel and serial performance models separately and measure the work on the paths to compute the critical path work. However, it is possible that the critical paths can differ since any additional work in the parallel performance model can potentially shift the critical path to a different path. The performance models in Figure 5.2(a) and (b) provide a concrete instance where the critical path differs. The critical path in the serial performance model in Figure 5.2(a) contains the step nodes S0, S2, S6, and S1. Whereas, the critical path in the parallel performance model in Figure 5.2(b) has step node S0, S3, S6, and S1. Although in this instance both S2 and S3 perform additional work in the parallel performance model, S3 performs slightly greater work than S2. Hence the critical path shifts from passing through S2 to S3.

To ensure the comparison is consistent, TaskProf needs to select the critical path of one of the serial or the parallel performance models and find the same corresponding path in the other performance model. One approach is to pick the critical path of the serial performance model and find the corresponding path in the parallel performance model. But, this would not be useful in identifying the regions on the critical path of the parallel execution that are experiencing secondary effects. Instead, TaskProf selects the critical path of the parallel performance model and identifies the corresponding path in the serial performance model. Hence, to compute the critical path work for the serial performance model in Figure 5.2(a), TaskProf aggregates the work from the step nodes S0, S3, S6, and S1 which correspond to the step nodes that form the critical path of the parallel performance model in Figure 5.2(b).

TaskProf aggregates the work and the critical path work at each static code location associated with the step nodes by using a hash table. The static code location associated with step nodes can be the static regions of code that correspond to the step nodes, or the static spawn sites to which the step nodes belong. Each entry in the hash table is indexed by the static code location and contains the work at the location in the serial performance model, the work at the location in the parallel performance model, the critical path work corresponding to the location in the serial performance model, and the critical path work corresponding to the location in the parallel performance model. To compute the critical path work at a code location in the parallel performance model, TaskProf aggregates the work of all the step nodes that belong to the location and execute on the critical path. In contrast, to compute the critical path work at a code location in the serial performance model, TaskProf identifies the same step nodes that belong to the location and execute on the critical path of the parallel performance model irrespective of whether they execute on the critical path of the serial performance model. If a static code location does not perform work on the critical path, then the critical path work hash table entries for the code location will be zero.

Consider for instance, the static code location corresponding to region between lines 4 and 6 in Figure 5.2(a) and (b). The total work in the parallel and serial performance models corresponding to the region is the aggregate of the work in step nodes S2, S3, S4, and S5. In the parallel performance model in Figure 5.2(b), only the step node S3 that belongs to the region between lines 4 and 6 executes on the critical path. Hence, the critical path work corresponding to the region between lines 4 and 6 in the parallel performance model consists of the work from step node S3. To compute the critical path work corresponding to the region between lines 4 and 6 in the serial performance model, TaskProf uses the work from the step node S3, which occurs on the critical path of the parallel performance model.

Algorithm 13: Generate the differential profile from oracle performance model T_O, and parallel performance model T_P.

```
1  function DifferentialProfile($T_O, T_P$)
2      foreach $N_O, N_P$ in depth-first traversal of $T_O, T_P$ do
3          $C_O \leftarrow$ ChildNodes($N_O$)                ▷ Return all child nodes
4          $N_O.w \leftarrow \sum_{C \in C_O} C.w$
5          $C_P \leftarrow$ ChildNodes($N_P$)
6          $N_P.w \leftarrow \sum_{C \in C_P} C.w$
7          $\langle N_P.s, N_P.l \rangle \leftarrow$ CriticalPathWork($N_P$)
8          $\langle N_O.s, N_O.l \rangle \leftarrow$ OracleCriticalPathWork($N_O, N_P.l$)
9          if $N_O$, $N_P$ are STEP nodes then
10             AggregateWork($N_O, N_P$)      ▷ Work inflation at step region
11     AggregateCriticalWork($T_O.l, T_P.l$)  ▷ Critical path work inflation
12     GenerateDifferentialProfile()                ▷ Produce profile
```

Offline Differential Analysis Algorithm Description

Algorithm 13 outlines TaskProf's approach to perform differential analysis in the offline profiling mode. The algorithm takes the serial and parallel performance models as input and computes the total work and the total critical path work from the two performance models. It also computes the work and the critical path work at each static code location in both the serial and parallel performance models. The algorithm performs these computations by maintaining three quantities with each node in the performance models. They are - (1) work (w), (2) critical path work (s), and (3) set of step nodes on the critical path (l). To track the work and critical path work at each static code location, the algorithm maintains a hash table indexed by the static source code location. The algorithm traverses the two performance models concurrently in a depth-first manner (line 2 in Algorithm 13). At every node in the traversal of both the performance models, the algorithm first computes the work as the sum of work in all child nodes of the node (lines 3-6 in Algorithm 13). To compute the critical path work at each node, the algorithm first computes the critical path from the parallel performance model (line 7 in Algorithm 13). It subsequently uses the set of step nodes on the critical path under the node in the performance model, to compute the critical path work in the serial performance model (line 8 in Algorithm 13).

TaskProf computes the critical path work and the set of step nodes on the critical path for the parallel performance model in the same way as they are determined in the parallelism computation described in Algorithm 7. TaskProf computes the critical path from all the subsets of step nodes under the input node that have to execute serially. These subsets can be the child step nodes and step nodes on the critical path of the child finish nodes, or serial set of step nodes introduced at each async child node of the input node. TaskProf selects the critical path work and the set of step nodes on the critical path as the subset that has the maximum aggregate of work from all the step nodes.

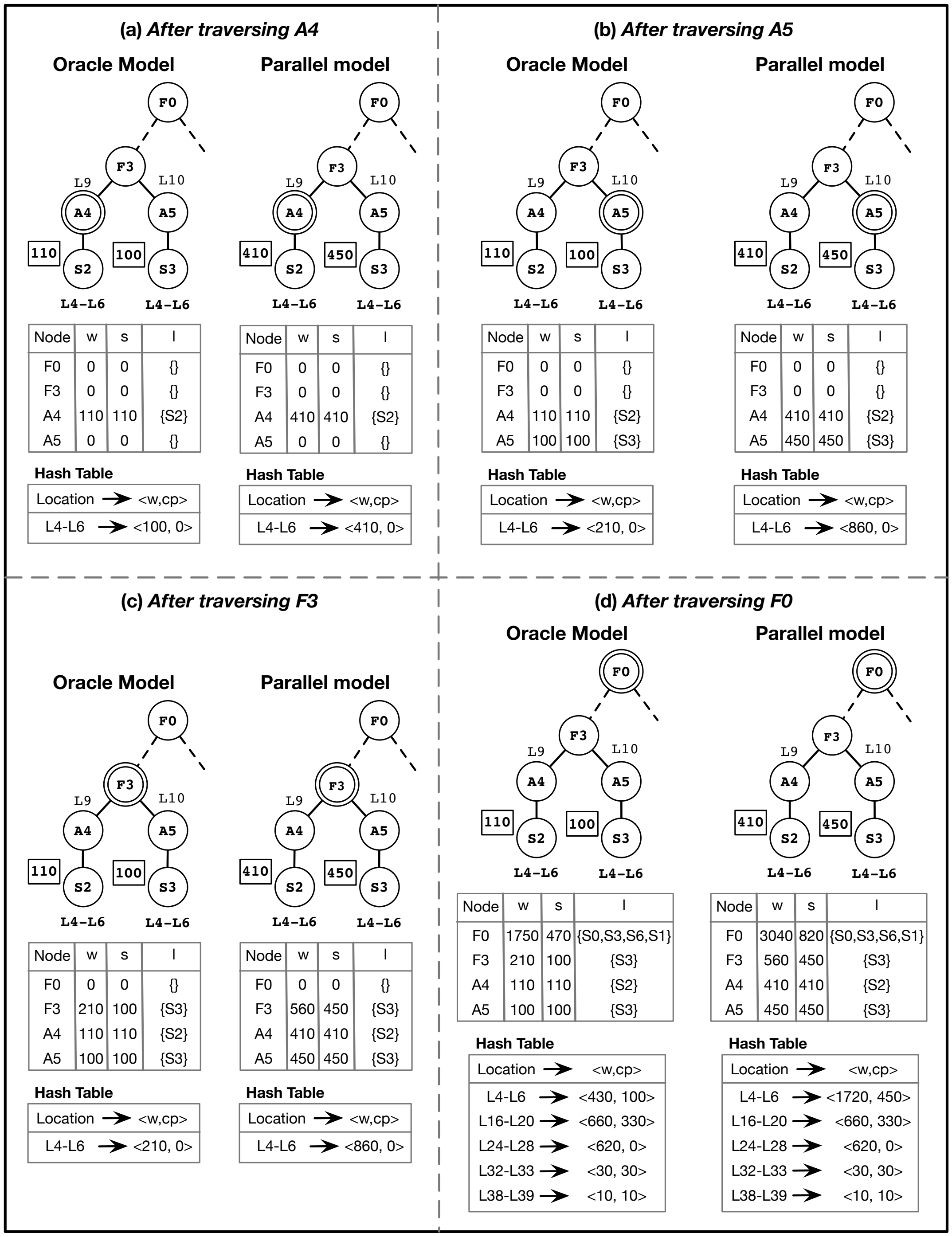
Algorithm 14: Compute work s and the list of step nodes l of path corresponding to the critical path l_p of parallel performance model in the oracle performance model given input node N.

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

Using the set of step nodes on the critical path for a given node in the parallel performance model, TaskProf computes the critical path work and the list of step nodes on the critical path under the same node of the serial performance model. Algorithm 14 describes TaskProf's approach to compute the critical path work in the serial performance model. Algorithm 14 considers each subset of step nodes that execute serially under the input node in the same order as Algorithm 7 considers the subset in the parallel performance model. It initially checks if the child step nodes and step nodes on the critical path of the child finish nodes are the same as the input set of step nodes that form the critical path in the parallel performance model (lines 2-6 in Algorithm 14). If the set of step nodes are not the same, Algorithm 14 checks if the serial set of step nodes introduced at each async child is the same as the input set (lines 8-14 in Algorithm 14). Finally, Algorithm 14 returns the subset of step nodes that match the input set and the aggregate work on the subset as the critical path work and the set of step nodes on the critical path for the input node in the serial performance model.

After computing the work and the critical path work at a node in the depth-first traversal, Algorithm 13 has to update the hash table that maintains per static source code information. Recall that each entry in the hash table is indexed by the static code location and contains the work at the location in the serial performance model, the work at the location in the parallel performance model, the critical path work corresponding to the location in the serial performance model, and the critical path work corresponding to the location in the parallel performance model. If the node being processed is a step node, then the algorithm adds the work from the step node to the entry corresponding to the static source code location of the step node in hash table (lines 9-10 in Algorithm 13). The algorithm updates only the work from the given node in the hash table, and not the critical path work since the node is not guaranteed to be part of the critical path of the entire program. The algorithm can add the critical path work to the hash table only when the set of step nodes that are on the critical path of the program are computed after the depth-first traversal of the performance models.

After the traversal of the serial and the parallel performance models, the root node of the parallel performance model will contain the work, the critical path work, and the set of step nodes on the critical path. The root node of the serial performance model will contain the total work, and the work and set of step nodes in the serial performance



(a) After traversing A4

Oracle Model

Node	w	s	l
F0	0	0	{}
F3	0	0	{}
A4	110	110	{S2}
A5	0	0	{}

Hash Table

Location →	<w,cp>
L4-L6 →	<100, 0>

Parallel model

Node	w	s	l
F0	0	0	{}
F3	0	0	{}
A4	410	410	{S2}
A5	0	0	{}

Hash Table

Location →	<w,cp>
L4-L6 →	<410, 0>

(b) After traversing A5

Oracle Model

Node	w	s	l
F0	0	0	{}
F3	0	0	{}
A4	110	110	{S2}
A5	100	100	{S3}

Hash Table

Location →	<w,cp>
L4-L6 →	<210, 0>

Parallel model

Node	w	s	l
F0	0	0	{}
F3	0	0	{}
A4	410	410	{S2}
A5	450	450	{S3}

Hash Table

Location →	<w,cp>
L4-L6 →	<860, 0>

(c) After traversing F3

Oracle Model

Node	w	s	l
F0	0	0	{}
F3	210	100	{S3}
A4	110	110	{S2}
A5	100	100	{S3}

Hash Table

Location →	<w,cp>
L4-L6 →	<210, 0>

Parallel model

Node	w	s	l
F0	0	0	{}
F3	560	450	{S3}
A4	410	410	{S2}
A5	450	450	{S3}

Hash Table

Location →	<w,cp>
L4-L6 →	<860, 0>

(d) After traversing F0

Oracle Model

Node	w	s	l
F0	1750	470	{S0,S3,S6,S1}
F3	210	100	{S3}
A4	110	110	{S2}
A5	100	100	{S3}

Hash Table

Location →	<w,cp>
L4-L6 →	<430, 100>
L16-L20 →	<660, 330>
L24-L28 →	<620, 0>
L32-L33 →	<30, 30>
L38-L39 →	<10, 10>

Parallel model

Node	w	s	l
F0	3040	820	{S0,S3,S6,S1}
F3	560	450	{S3}
A4	410	410	{S2}
A5	450	450	{S3}

Hash Table

Location →	<w,cp>
L4-L6 →	<1720, 450>
L16-L20 →	<660, 330>
L24-L28 →	<620, 0>
L32-L33 →	<30, 30>
L38-L39 →	<10, 10>

Figure 5.3: Illustration of the offline differential analysis computation as TaskProf traverses nodes A4, A5, F3 and F0 in the Figure 5.2. The node being visited in each figure is shown with double edge. The figures omit the parts of the performance model that are not being traversed in the illustration.

model that correspond to the critical path of the parallel performance model. The algorithm updates the critical path work in the hash table from the set of step nodes in the root node of each performance model (line 11 in Algorithm 13). Finally, the algorithm generates the differential profile using the total work and the critical path work from the root nodes of the performance models, and the work and the critical path work at each static code location in the hash table (line 12 in Algorithm 13).

Illustration of Offline Differential Analysis

Figure 5.3 provides an illustration of TaskProf's offline differential analysis algorithm on the serial and parallel performance models shown in Figure 5.2. It shows the changes to the performance model and the per-static location hash tables as the nodes A4, A5, F3, and F0 are traversed in the two performance models. The figure omits the parts that are not traversed in the illustration, and instead replaces them with dashed edges.

As async node A4 is traversed in the two performance models, the algorithm first computes the work and the critical path work at A4 in the parallel performance model. Since, the sub-tree at A4 has a single child step node, the algorithm computes the work, the critical path work, and the set of step nodes on the critical path from child step node S2. Using the set of step nodes on the critical path in the parallel performance model, the algorithm computes the critical path work at A4 in the serial performance model, as shown in Figure 5.2(a). The algorithm would have updated the entry in both the hash tables corresponding to the location L4-L6, with the work from the step node S2 when the node S2 is traversed. Similarly, the algorithm computes the work and the critical path work at async node A5 in both performance models and updates the entry in the hash table, as shown in Figure 5.2(b).

At finish node F3, the algorithm first computes the critical path work at F3 as the work from step node S3 since it forms the serial path that performs the highest amount of work in F3. Now, using the critical path under F3 from the parallel performance model, the algorithm selects the same path with node S3 as the critical path under F3 in the serial performance model (Figure 5.2(c)). Finally, after traversing the remaining nodes in both the performance models, the algorithm processes the root node F0. Although the algorithm updates the work in the hash table as it traverses through the performance models, it does not update the critical path work. It updates the critical path work of each static code location that executes on the critical path only after the algorithm computes the set of step nodes on the critical path of the entire program in the root node, as shown in Figure 5.2(d).

5.3.2 Differential Analysis for On-The-Fly Profiling

In the on-the-fly profiling mode, TaskProf does not write the performance models of the serial and parallel executions for post-mortem analysis. Instead, TaskProf constructs the performance model as the program executes, either serially or in parallel, and gathers minimum information from the two models to perform differential analysis.

A key challenge in performing differential analysis from on-the-fly profile executions is that TaskProf will not have the entire performance models of the serial and parallel executions available to perform the analysis. In the offline profiling mode, TaskProf re-constructs the serial and parallel performance models after executions and stores the performance models in memory. Since TaskProf has access to the complete performance models of the serial and parallel executions, it can traverse the models concurrently to not only compute the work inflation in the program and at each static source code location, but also compute the inflation on the critical path. Having access to the complete performance models was crucial for identifying the critical path in the parallel performance model and finding the same corresponding path in the serial performance model. Hence, to perform differential analysis from on-the-fly profile executions, we need a strategy to (1) compute the work and critical path work inflation when the entire performance models are not available, and (2) identify the same paths as the critical paths of the parallel and serial executions.

One approach to perform differential analysis on-the-fly is to perform the serial and parallel executions concurrently and compute the work inflation whenever the corresponding nodes in the performance models complete. But such a strategy would defeat the purpose of using the serial execution as the oracle execution since interference with the parallel execution could introduce secondary effects in the serial execution. Moreover, the parallel execution could experience secondary effects due to concurrent execution with the serial execution. Hence, we have to perform the serial and parallel executions separately and record information from the two executions to perform differential analysis. Unlike the on-the-fly parallelism computation and what-if analyses, which perform the computation as the program executes, differential analysis in the on-the-fly mode performs the computation using information recorded from the two executions. So, what is the minimum information that is necessary from the serial and parallel performance models to perform differential analysis?

To highlight regions in the program that are experiencing secondary effects, TaskProf's offline differential analysis reports work and critical path work inflation at the granularity of the static code region or spawn site associated with the step nodes. If we record the total work and the work on the critical path of each static code location associated with the step nodes from the serial and parallel performance models, it would be sufficient to perform differential analysis even if the entire performance models are not available. Hence, in the on-the-fly profiling mode, TaskProf aggregates the work and the critical path work at each static source code location during both serial and parallel executions and records the per-static source location work and critical path work in a file at the end of the executions.

To identify work inflation on the critical path, TaskProf does not identify the critical path of the serial execution, but instead identifies the path that is corresponding to the critical path of the parallel execution. Hence, along with the work and the critical path work at each static source code location, TaskProf records the set of step nodes that execute on the critical path from the parallel performance model. This implies that TaskProf has to perform the parallel execution first to record the set of step nodes on the critical path and subsequently use them to identify the corresponding set of step nodes in the serial execution.

In summary, TaskProf's differential analysis in the on-the-fly profiling mode writes minimal information from the two performance models to a file. TaskProf first executes the program in parallel and constructs the performance model. As the program executes, TaskProf aggregates the work and the critical path work at each static source code location. Once the parallel execution completes, TaskProf writes the work and the critical path work at each location and the list of step nodes on the critical path to a profile data file. Subsequently, TaskProf executes the program serially and records in a separate profile data file, the total work and the work on the path corresponding to the critical path of the parallel execution. Finally, TaskProf reads the two profile data files, and generates the differential profile that specifies the total work inflation and the work inflation on the critical path of the entire program, and at each static region or spawn site in the program.

On-The-Fly Differential Analysis Algorithm Description

Algorithms 15 and 16 present our approach to profile the parallel and serial executions, respectively, to perform differential analysis. In the parallel execution of the input program, TaskProf computes the total work, the critical path work, and the set of step nodes on the critical path of entire program similar to the on-the-fly parallelism computation described in Algorithm 8. Additionally, TaskProf also computes the total work and the work on the critical path at each static source code location in the program.

To perform the computations on the fly, TaskProf tracks four quantities with each node in the performance model. They are - (1) the total work (w) performed in the sub-tree under the node, (2) the set of step nodes on the critical path (CS) in the sub-tree under the node, (3) the set of step nodes that perform the serial work from step and finish children (SS), and (4) the set of step nodes that perform the serial work in the left step and finish siblings (LS). TaskProf tracks the three sets of step nodes at each node to determine the set of step nodes that execute on the critical path of the parallel execution. Along with these four quantities, the algorithm maintains a hash table indexed by the static source code location to track the work and critical path work at each static code location.

TaskProf updates the hash table and the four quantities when nodes are added to and removed from the performance model. When a node is added to the performance model, TaskProf initializes the left serial work (LS) of the node with the serial work (SS) of the parent node(line 5 in Algorithm 15). When a node completes, TaskProf

Algorithm 15: On-the-fly differential analysis computation in the parallel execution when parent node P creates child node N, and when N completes.

```
1  procedure OnNodeCreation(N, P)
2      $N.w \leftarrow 0$                                        ▷ work
3      $N.CS \leftarrow \emptyset$                      ▷ set CS tracks critical path work
4      $N.SS \leftarrow \emptyset$                          ▷ set SS tracks serial work
5      $N.LS \leftarrow P.SS$                       ▷ set LS tracks left serial work
6  procedure OnNodeCompletion(N, P)
7      $P.w \leftarrow P.w + N.w$
8      if $\sum_{L \in N.LS} L.w + \sum_{C \in N.CS} C.w > \sum_{S \in P.CS} S.w$ then
9          $P.CS \leftarrow N.LS \cup N.CS$
10     if $N$ is a STEP or FINISH node then
11         $P.SS \leftarrow P.SS \cup N.CS$
12         if $N$ is a STEP node then
13             AggregateWork(N)                 ▷ Aggregate work at spawn site
```

summarizes the node's work, critical path work, and set of step nodes at the parent node. TaskProf adds the work of the node to its parent node (line 7 in Algorithm 15). Then it checks if the node contributes to the critical path under the parent node and updates the set of step nodes that perform the work on the critical path accordingly (lines 8-9 in Algorithm 15). If the node is a finish or a step node, TaskProf has to update the serial set of the parent since the step and finish children execute serially(line 11 in Algorithm 15). After summarizing the work and the critical path work at the parent node, Algorithm 15 updates the hash table maintaining per static source code information. If the node that completes execution is a step node, TaskProf adds the work from the step node to the entry corresponding to the static source code location of the step node in hash table (lines 12-13 in Algorithm 15).

Once the parallel execution of the program completes, the root node of the performance model will have the total work in the program in the w variable, and the step nodes on the critical path in the set *CS*. TaskProf attributes the work from the set of step nodes on the critical path to the corresponding static source code locations in the hash table. Finally, TaskProf records the total work, the work on the critical path, the set of step nodes on the critical path, and the set of entries in the hash table in the profile data file.

Algorithm 16: On-the-fly differential analysis computation in the oracle execution when parent node P creates child node N, and when N completes.

```
1  procedure OnNodeCreation($N$, $P$)
2      $N.w \leftarrow 0$                                        ▷ work
3      $N.CS \leftarrow \emptyset$                      ▷ set CS tracks critical path work
4  procedure OnNodeCompletion($N$, $P$, $l_p$)
5      $P.w \leftarrow P.w + N.w$
6      if $N$ is a STEP node then
7          if $N \in l_p$ then
8              $N.CS \leftarrow \langle N.r, N.w \rangle$
9          AggregateWork($N$)                  ▷ Aggregate work at spawn site
10     $P.CS \leftarrow P.CS \cup N.CS$
```

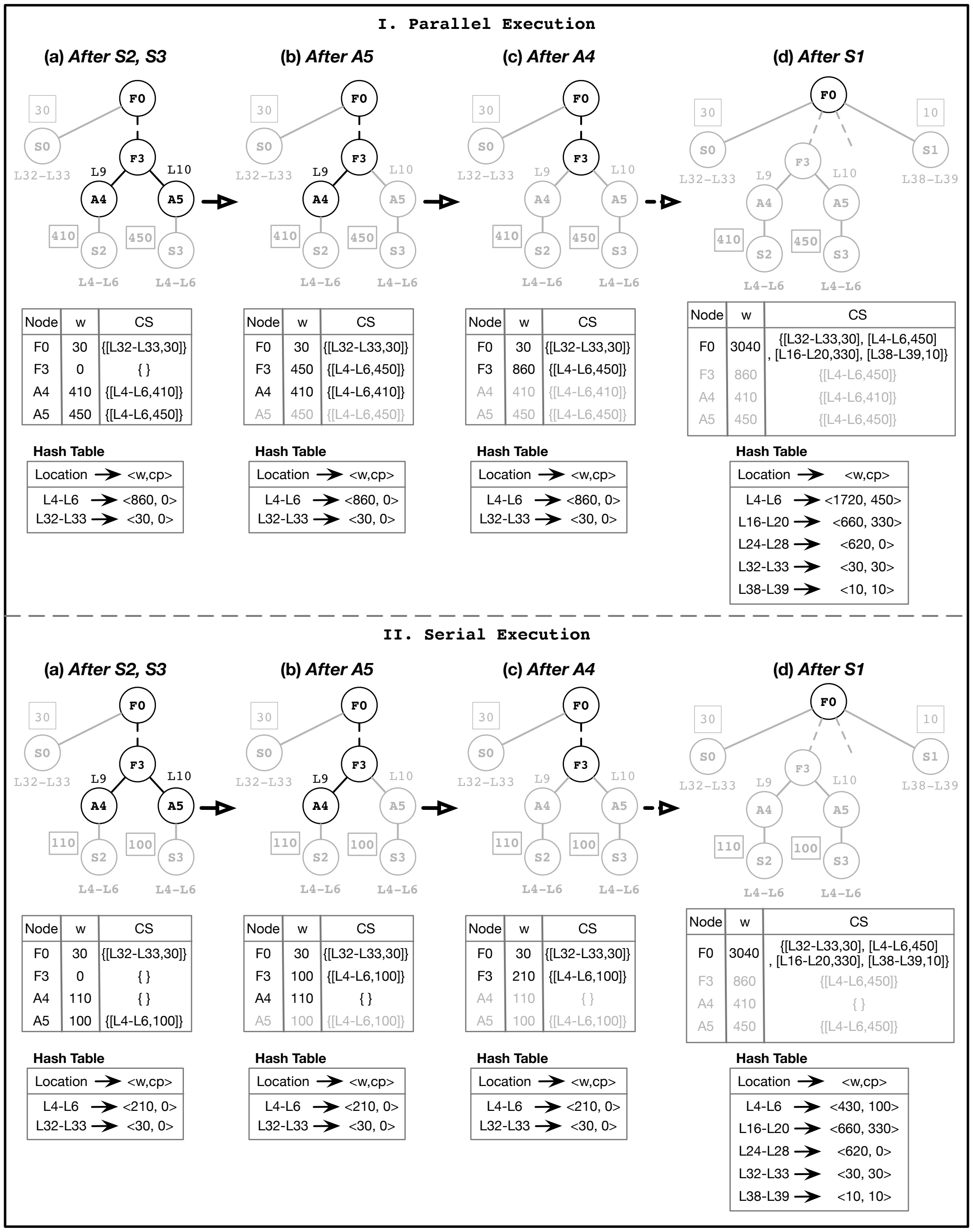
In the serial execution TaskProf does not explicitly compute the set of step nodes executing on the critical path. Instead, it computes step nodes corresponding to the critical path of the parallel execution and the total work, as outlined in Algorithm 16. The algorithm tracks two quantities with each node. They are - (1) the total work (w), and (2) the set of step nodes on path corresponding to the parallel critical path (CS). The algorithm also maintains a hash table that stores the work and critical path work at each static code location.

TaskProf initializes these quantities for each node that is added to the performance model. When a node completes execution, TaskProf summarizes the work from the node at its parent node (line 5 in Algorithm 16). If the node that completes execution is a step node, TaskProf checks if the node corresponds to any of the input set of step nodes on the critical path of the parallel execution (line 7 in Algorithm 16). If the step node indeed corresponds to the critical path of the parallel execution, TaskProf adds the step node to the set of step nodes CS (line 8 in Algorithm 16). For a step node that completes execution, TaskProf also add the work from the step node to the entry corresponding to the source code location of the step node in the hash table (line 9 in Algorithm 16). After the completion of the serial execution of the program, the root node will have the total work in the program in the w variable, and the step nodes corresponding to the critical path of the parallel execution in the set *CS*. TaskProf attributes the work from the set *CS* to the corresponding entries in the hash table. Similar to the on-the-fly parallel profile execution, TaskProf records the total work, the work on the critical path, and the set of entries in the hash table to a profile data file specific to the serial execution. In a post-mortem analysis over the parallel and serial profile data files, TaskProf computes the total work inflation and the work inflation on the critical path of the program, as well as for each static region of code or spawn site in the program.

Illustration of On-The-Fly Differential Analysis Algorithm

We illustrate TaskProf's approach to perform differential analysis in the on-the-fly profiling mode using Figure 5.4. The figure illustrates the updates to the performance model and the per-static location hash table in the parallel and serial execution of the program in Figure 3.2(a) having performance models show in Figure 5.2. The figure shows the changes when nodes S2, S3, A5, A4, and S1 complete their execution.

As nodes S2 and S3 complete in the parallel execution, TaskProf updates the work, and the set of step nodes on the critical path of their parent nodes. Further, since S2 and S3 are step nodes, TaskProf adds the work from the two nodes to the entry corresponding to their static source code location in the hash table, as shown in Figure 5.4(I)(a). When async node A5 completes, TaskProf adds the work in A5 to its parent node and determines if the path through A5 forms the critical path under the parent node. When async node A4 completes, TaskProf performs the same check and finds that the path through A5 containing step node S3 is the critical path since it performs greater work. Since both A4 and A5 are async nodes, TaskProf does not update the hash table after the completion of the two nodes (Figure 5.4(I)(b) and (c)). Finally, after the final node in the sub-tree under the root node S1 completes, the root node F0 contains information about the step nodes on the critical path of the parallel execution. TaskProf uses the set of step nodes on the critical path and updates the entries corresponding their static location in the hash table, as shown in Figure 5.4(I)(d). Subsequently, TaskProf uses the set of step nodes on the critical



I. Parallel Execution

(a) After S2, S3

Node	w	CS
F0	30	{[L32-L33,30]}
F3	0	{ }
A4	410	{[L4-L6,410]}
A5	450	{[L4-L6,450]}

Hash Table

Location	<w,cp>
L4-L6	<860, 0>
L32-L33	<30, 0>

(b) After A5

Node	w	CS
F0	30	{[L32-L33,30]}
F3	450	{[L4-L6,450]}
A4	410	{[L4-L6,410]}
A5	450	{[L4-L6,450]}

Hash Table

Location	<w,cp>
L4-L6	<860, 0>
L32-L33	<30, 0>

(c) After A4

Node	w	CS
F0	30	{[L32-L33,30]}
F3	860	{[L4-L6,450]}
A4	410	{[L4-L6,410]}
A5	450	{[L4-L6,450]}

Hash Table

Location	<w,cp>
L4-L6	<860, 0>
L32-L33	<30, 0>

(d) After S1

Node	w	CS
F0	3040	{[L32-L33,30], [L4-L6,450] , [L16-L20,330], [L38-L39,10]}
F3	860	{[L4-L6,450]}
A4	410	{[L4-L6,410]}
A5	450	{[L4-L6,450]}

Hash Table

Location	<w,cp>
L4-L6	<1720, 450>
L16-L20	<660, 330>
L24-L28	<620, 0>
L32-L33	<30, 30>
L38-L39	<10, 10>

II. Serial Execution

(a) After S2, S3

Node	w	CS
F0	30	{[L32-L33,30]}
F3	0	{ }
A4	110	{ }
A5	100	{[L4-L6,100]}

Hash Table

Location	<w,cp>
L4-L6	<210, 0>
L32-L33	<30, 0>

(b) After A5

Node	w	CS
F0	30	{[L32-L33,30]}
F3	100	{[L4-L6,100]}
A4	110	{ }
A5	100	{[L4-L6,100]}

Hash Table

Location	<w,cp>
L4-L6	<210, 0>
L32-L33	<30, 0>

(c) After A4

Node	w	CS
F0	30	{[L32-L33,30]}
F3	210	{[L4-L6,100]}
A4	110	{ }
A5	100	{[L4-L6,100]}

Hash Table

Location	<w,cp>
L4-L6	<210, 0>
L32-L33	<30, 0>

(d) After S1

Node	w	CS
F0	3040	{[L32-L33,30], [L4-L6,450] , [L16-L20,330], [L38-L39,10]}
F3	860	{[L4-L6,450]}
A4	410	{ }
A5	450	{[L4-L6,450]}

Hash Table

Location	<w,cp>
L4-L6	<430, 100>
L16-L20	<660, 330>
L24-L28	<620, 0>
L32-L33	<30, 30>
L38-L39	<10, 10>

Figure 5.4: Illustration of the on-the-fly differential analysis computation using the performance models in Figure 5.2. Figures (I) and (II) show the updates to the performance models and hash tables after the completion of nodes S2 and S3, A5, A4, and S1 during serial and parallel executions, repectively. The nodes and the quantities for the nodes that have already completed are grayed out. The figures omits the parts of the performance model that are not being traversed in the illustration. The changes to the LS and SS quantities in the parallel execution are also not shown.

path of the performance model of the parallel execution to identify the corresponding path in the performance model of the serial execution.

Figure 5.4(II) shows the changes to performance model and the hash table in the serial execution. In the serial execution, TaskProf does not compute the critical path, but checks if each step node belongs to the path corresponding to the critical path in the parallel execution. Therefore, after the completion of step nodes S2 and S3 in Figure 5.4(II)(a), TaskProf adds S3 to the critical path under A5 since S3 is a part of the critical path of the parallel performance model. TaskProf also adds the work from S2 and S3 to the respective parent nodes and the entries in the hash table corresponding to their source code locations. Similarly, after the completion of async nodes A5 and A4 (Figure 5.4(II)(b) and (c)), TaskProf updates the work in their parent nodes to propagate the total work. Finally, after the last node in the sub-tree under the root node S1 completes, the root node F0 contains the set of step nodes that are on the path corresponding to the critical path of the parallel performance model. Using this set, TaskProf updates the critical path work in the entries in the hash table, as shown in Figure 5.4(II)(d).

5.4 Differential Analysis with Multiple Performance Counter Events

Our differential analysis technique measures the inflation in the work at various parts of a task parallel program and highlights regions that are experiencing significant inflation due to secondary effects. To observe the inflation in the work, TaskProf measures work in terms of hardware execution cycles. Hardware execution cycles generally show an increase in the presence of secondary effects, irrespective of the hardware or system resource that is causing secondary effects. For instance, if a program is experiencing secondary effects due to either true or false sharing of cache lines, then there will be an inflation in cycles measured for the program. Similarly, if a program is having reduced locality, or performing greater remote memory accesses either to another socket or a heterogenous accelerator, it will experience an increase in hardware execution cycles. However, while measuring work inflation in cycles can indicate the presence of secondary effects, providing information about the hardware or system resource that is the likely source of the secondary effects will be beneficial to a programmer.

Modern processors contain hardware performance counters that provide an insight into the execution of a program. These performance counters can record events from various hardware and system components. For instance, performance counters can measure cache misses, cache references, branch misses, page faults, TLB misses, remote memory accesses, and numerous other events from various components. Our idea is that if a program is experiencing secondary effects in the parallel execution due to some component, then along with the work inflation in hardware execution cycles, a performance counter event corresponding to the component will also show inflation. For example, if a program is experiencing secondary effects due to true or false cache sharing, then the parallel execution will show an inflation in the HITM counter which measures the number of cache hits in the modified state. Similarly, if a program is experiencing reduced locality in shared caches due to parallel execution then the counter measuring last level cache misses will show inflation. Also, inflation in the counter measuring remote DRAM accesses can indicate the lack of affinity between the processor producing the data and the one using it. Therefore, in addition to differential analysis with hardware execution cycles, TaskProf can perform differential analysis with any hardware performance counter event that is available on the execution machine.

TaskProf can perform differential analysis with any performance counter event in both the offline and on-the-fly profiling modes. The programmer specifies the set of events to measure as input to TaskProf. In either profiling modes, TaskProf executes the program serially and in parallel for each performance counter event to be measured and compares the two performance models. The measurements from the performance counters are recorded as the work in the step nodes, and TaskProf performs differential analysis in the same approach as shown in Algorithm 13 for offline profiling and Algorithms 15 and 16 for on-the-fly profiling. In either modes, TaskProf generates a profile for each performance counter event highlighting the inflation in the work and the critical path work of the entire program and at the different code regions or spawn sites in the program. Once TaskProf generates the profiles for each performance counter event specified as input, it produces the complete differential profile that shows the inflation for each counter for the entire program and for each region or spawn site in the program. Figure 5.2(c) shows the differential profile for the program in Figure 3.2(a). It shows the inflation in four performance counters - cycles, HITM, local DRAM accesses, and remote DRAM accesses. From the profile, we can notice that the region L4-L6 is experiencing inflation in the total number of hardware execution cycles and the HITM events. This means that the program is likely experiencing secondary effects due true or false sharing in the region L4-L6.

TaskProf's differential analysis, by measuring inflation in work and critical path work for numerous performance counter events at a fine-granularity, provides an insightful technique to identify regions in a program experiencing any kinds of secondary effects and also the possible source of the secondary effects.

5.5 Summary

In this chapter, we introduce TaskProf's differential analysis technique to identify parts of a task parallel program that are experiencing secondary effects of parallel execution due to contention in hardware or system resources. Secondary effects are a common cause of performance degradation in task parallel program. They are also hard to detect since they are caused by contention in hardware and system resources. When a program has secondary effects, the parallel execution shows inflation in the total work and work on the critical path over the serial execution. Using TaskProf's performance model which captures work performed in a program at a fine granularity, our differential analysis technique compares performance models from the serial and parallel executions to highlight regions experiencing secondary effects. We also show that by performing differential analysis over a range of hardware performance counter events, we can identify the hardware or system resources which are possibly the source of the secondary effects.

While TaskProf highlights regions where the secondary effects are seen, these regions could just be the symptoms of the problem. The root cause of the secondary effects in the program could as well be some other region of the program that may be dominating the hardware or system resource that is the source of the secondary effects. We do not claim that TaskProf identifies the root cause of the secondary effects. To perform root-cause diagnosis, techniques like blame shifting have been proposed to identify causes lock contention analysis [157], and idleness in hybrid programs [38]. We intend to explore it in the context of root-cause analysis of secondary effects.

Chapter 6

Experimental Evaluation

We have so far presented techniques to compute the parallelism of task parallel programs, identify regions that matter for increasing parallelism through what-if analyses, and identify regions experiencing secondary effects of parallel execution. We have implemented concrete instantiations of these techniques in our profiler, TaskProf. In this chapter, we evaluate the effectiveness and performance of TaskProf on a suite of benchmarks and describe how TaskProf guided us in identifying performance bottlenecks in the benchmarks.

We begin this chapter by providing a detailed description of the prototype implementation of TaskProf (Section 6.1). We then discuss the experimental setup including the system used for our evaluation, the list of benchmarks that we evaluate, and also the runtime settings used in our evaluation (Section 6.2). Then we present our evaluation of TaskProf on the following four parameters: (1) Is TaskProf effective in identifying regions that matter for parallelism and addressing secondary effects? (Section 6.3) (2) How does TaskProf compare with the state-of-the-art profilers that identify parallelism bottlenecks and secondary effects? (Section 6.4) (3) Is TaskProf efficient in profiling task parallel programs with low overhead? (Section 6.5) (4) Is TaskProf useful for average programmers in quickly identifying parts of the program that must be parallelized? (Section 6.6)

6.1 TaskProf Prototype Implementation

We have built a prototype of TaskProf for task parallel programs written using the Intel Threading Building Blocks (TBB) task parallel library [49]. The prototype is made up of (1) the TaskProf library, (2) analysis tools to perform offline what-if and differential analyses, and (3) driver tools for on-the-fly what-if and differential analyses. The TaskProf library implements TaskProf's functionality to profile the input task parallel program in both offline and on-the-fly modes. In the offline mode, the TaskProf library constructs the performance model, and records work measurements in the step nodes, as well as in the regions selected for what-if analyses. It then writes the performance model and work measurements to a profile data file. In the on-the-fly mode, along with constructing the performance model and recording work measurements, the TaskProf library also performs the parallelism, what-if and differential analyses computation. The offline analyses tools re-construct the performance model and implement the functionality to perform what-if and differential analyses. The prototype also includes tools written in Python to repeatedly drive the program execution to perform on-the-fly what-if and differential analyses.

Along with the TaskProf prototype, we also provide an extended TBB library that contains calls to the TaskProf runtime library. Intel TBB provides library functions to create and manage tasks in C++ programs. Underneath, the TBB runtime library uses the randomized work stealing technique to map the tasks to the threads for execution. Similar to the *spawn* and *sync* constructs, Intel TBB provides the *spawn* and *wait_for_all* functions to create a new task and wait for the child tasks, respectively. In addition to the *spawn* and *wait_for_all* functions, Intel TBB implements several generic parallel patterns like parallel for, parallel reduce, parallel invoke, *etc.*, using the *spawn* and *wait_for_all* functions. The extended TBB library that we provide overrides these TBB functions to insert calls to the TaskProf library. In addition, the extended TBB library tracks file name and line information at each spawn site to attribute the parallelism from the performance model to the static source code locations.

The TaskProf library uses hardware performance counters to perform fine-grained measurement of various metrics. It uses the `perf_events` module in Linux to read hardware performance counters. The prototype can use any available hardware performance counter: dynamic instructions, execution cycles, HITM events, local and remote DRAM accesses, last level cache misses, and floating point operation cycles. We have made the entire prototype open-source [176].

Table 6.1: Applications used to evaluate TaskProf, the benchmark suite each application belongs to, a short description of each application, and the inputs used for evaluation.

Benchmark Suite	Application	Description	Input
Coral	MILCmk	MIMD Lattice Computation	2ˆ18
	LULESH	Shock Hydrodynamics problem	1.728M
PBBS	comparisonSort	Generic sort	10M
	integerSort	Sort key-value pairs	10M
	removeDuplicates	Remove duplicate value	10M
	dictionary	Batch dictionary operations	10M
	suffixArray	Sequence of suffixes	10M
	BFS	Breadth first search	10M
	maxIndependentSet	Maximal Independent Set	10M
	maximalMatching	Maximal matching	10M
	minSpanningForest	Minimumspanningforest	10M
	spanningForest	Spanning tree or forest	10M
	convexHull	Convex hull computation	10M
	nearestNeighbors	K-nearest neighbors	10M
	delaunayTriang	Delaunay triangulation	2M
	delaunayRefine	Delaunay Refinement	10M
	rayCast	Triangle intersection	10M
	nBody	Calculate Nbody force	1M
Parsec	blackscholes	Stock option pricing	native
	bodytrack	Tracking of a human body	native
	fluidanimate	Simulate luid dynamics	native
	streamcluster	Clustering algorithm	native
	swaptions	Price a portfolio	native

6.2 Experimental Setup

Our experiments were conducted on a 16-core dual-socket Intel x86-64 2.1Ghz Xeon server with 64 GB DRAM and hyper-threading disabled. The machine has a 32KB data cache, 32KB instruction cache, 256KB L2 cache, and 20MB L3 cache. Each cache line is 64 bytes. We execute each application five times while performing speedup experiments and consider the average execution time. To perform what-if analyses, we use 128 for both anticipated parallelism and possible parallelization factor because we want the application to have large enough parallelism to obtain scalable speedup on a machine with a large number of cores. We use 10× the average tasking overhead ($k = 10$ in Algorithm 12) as the tasking overhead threshold.

6.2.1 Benchmarks

We present our evaluation on a set of twenty three applications listed in Table 6.1. The applications include the `MILCmk` and the `LULESH` applications from the Coral benchmark suite [1], sixteen applications from the Problem Based Benchmark Suite (PBBS) [152], and five TBB applications from the Parsec benchmark suite [24]. `MILCmk` and `LULESH` applications were originally written in OpenMP. The PBBS applications were written in Cilk. We converted them to use the Intel TBB library.

6.3 Effectiveness in Identifying Performance Bottlenecks

To evaluate TaskProf's effectiveness in highlighting performance issues due to inadequate parallelism, task runtime overhead and secondary effects, we used TaskProf to profile all the twenty three applications. Table 6.2 provides a summary of the results from profiling all the applications.

Parallelism and Task Runtime Overhead

Table 6.2 shows that the speedup of all the applications is less than 16, which is the number of processors on our evaluation machine. However, the parallelism reported by TaskProf shows that applications that have reasonable parallelism exhibit speedup closer to the scalable speedup (16×) than applications with lower parallelism. For example, `nbody` has a parallelism of 126.18. Consequently, it exhibits a high speedup of 12.21×. In contrast, `blackscholes`, `suffix array`, and `minimum spanning forest` applications have low parallelism of 1.39, 6.84, 8.18. Thereby, these applications exhibit low speedups of 1.14×, 2.09×, and 3.49×, respectively.

While most of the twenty three applications have a task runtime overhead of less than 10%, four applications have relatively high task runtime overheads ($> 10\%$) as seen in Table 6.2. Along with highlighting the task runtime overhead in the entire program, TaskProf helped us identify the exact static lines of code corresponding to spawn sites that were experiencing high overhead in all four applications. We designed optimizations to reduce the task runtime overhead in three of the four applications, which we describe

Table 6.2: For each application, we list the initial speedup on a 16-core machine, the logical parallelism, total tasking overhead in the program in contrast to total useful work, the number of regions reported by our what-if analyses to increase the parallelism to 128, and the inflation in work and critical path work in terms of cycles.

Application	Initial speedup	Parallelism	Tasking over-head	# of regions reported	Work inflation in cycles	Critical path inflation in cycles
MILCmk	2.18X	43.83	40.12%	3	3.21X	3.58X
LULESH	4.23X	42.48	7.56%	2	4.02X	4.53X
compSort	4.74X	29.72	2.82%	1	1.32X	1.32X
integerSort	4.97X	35.19	8.27%	2	1.18X	1.06X
remDups	8.11X	48.13	2.61%	1	1.13X	1.08X
dictionary	8.78X	41.29	2.78%	1	1.36X	1.07X
suffixArray	2.09X	6.84	6.12%	3	1.32X	1.15X
BFS	6.69X	25.92	3.37%	3	1.22X	1.18X
maxIndSet	8.39X	26.65	7.67%	2	1.27X	1.10X
maxMatching	9.43X	46.17	7.24%	1	1.21X	1.17X
minSpanForest	3.49X	8.18	2.11%	3	1.28X	1.16X
spanForest	7.43X	38.17	6.23%	1	1.41X	1.16X
convexHull	8.11X	67.17	3.15%	0	1.59X	1.12X
nearestNeigh	4.69X	17.37	7.23%	4	1.48X	1.01X
delTriang	5.91X	58.72	7.24%	1	1.31X	1.43X
delRefine	2.98X	16.17	3.26%	3	1.58X	1.16X
rayCast	9.29X	53.21	21.41%	1	1.34X	1.07X
nBody	12.21X	126.18	16.36%	1	2.12X	1.73X
blackscholes	1.14X	1.39	1.02%	2	1.02X	1.01X
bodytrack	6.29X	31.16	2.71%	2	1.16X	1.22X
fluidanimate	10.13X	64.42	2.41%	1	1.02X	1.02X
streamcluster	12.25X	76.86	9.17%	5	1.47X	1.43X
swaptions	12.21X	73.75	47.36%	0	1.39X	1.37X

below in application case studies. We were unable to reduce the tasking overhead in `rayCast` because the computation to determine the cut-off was closely intertwined with actual computation and it was not straightforward to reduce the tasking overheads.

Regions Reported by What-If Analyses

Table 6.2 shows that none of the applications have a logical parallelism of 128. Hence, when we ran TaskProf's what-if analyses with an anticipated parallelism of 128, we expected TaskProf to report regions in all the benchmarks. However, for two applications, *convexHull* and *swaptions*, TaskProf did not report any regions. This was because parallelizing any region on the critical path would have increased the task runtime overhead in both the applications beyond the threshold determined by TaskProf. Hence, these two applications do not have sufficient parallelism for execution on a 128-core machine for the given input.

While TaskProf reported regions to parallelize in twenty one applications applications, we were able to design concrete parallelization strategies for regions in nine applications. TaskProf identifies regions that matter for improving parallelism. However, it does not determine if the regions can be parallelized. Further, TaskProf also does not perform automatic parallelization. The programmer has to concretely optimize the regions reported by TaskProf. We carefully inspected the regions reported by TaskProf in all the applications and found that only regions in nine applications did not have dependencies and could be parallelized. We devised various strategies to parallelize the regions reported by TaskProf ranging from simple techniques like reducing the task grain size to create more parallelism and parallelizing sequential loops to relatively complex techniques like parallel reduction and introducing parallel algorithms. We discuss how TaskProf's what-if analyses guides us to parallelize the regions in the application case studies in Section 6.3.1.

Applications with Secondary Effects

Table 6.2 shows that a majority of the twenty one applications are experiencing less than significant inflation in both the total work and the work on the critical path. However, two applications, `MILCmk` and `LULESH`, show relatively high inflation in both the total work and the work on the critical path. Using TaskProf's differential profile, we were able to identify the regions in the program that were experiencing the secondary effects. The differential profiles also clearly highlighted the hardware performance counter metrics that were showing an inflation and thereby, enabled us to design optimizations to mitigate the secondary effects in these applications.

Table 6.3: Summary of the speedup improvements after addressing the performance issues and the techniques of TaskProf that enabled us to identify the performance issues. We truncate the technique names for brevity. *Task overhead* stands for task runtime overhead, *what-if* stands for what-if analyses and *diff* stands for differential analysis.

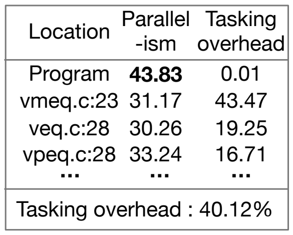
Application	Initial speedup	Speedup after optimization	Technique used to identify performance issue
MILCmk	2.18X	5.92X	task overhead, what-if, diff
nBody	12.21X	14.32X	task overhead, what-if
minSpanForest	3.49X	9.92X	what-if
suffixArray	2.09X	7.38X	what-if
BFS	6.69X	8.16X	what-if
compSort	4.74X	6.41X	what-if
spanForest	7.43X	8.34X	what-if
delRefine	2.98X	7.13X	what-if
LULESH	4.23X	5.90X	diff
swaptions	12.21X	14.19X	task overhead
blackscholes	1.14X	8.15X	what-if

6.3.1 Improving the Speedup of Applications

We demonstrate the effectiveness of TaskProf by describing the insights provided by TaskProf that enabled us to design code optimizations. Overall, TaskProf enabled us to improve the speedup of eleven applications. Table 6.3 lists the improvement in speedup of each application and the techniques of TaskProf that enabled us to address the performance issues in the applications.

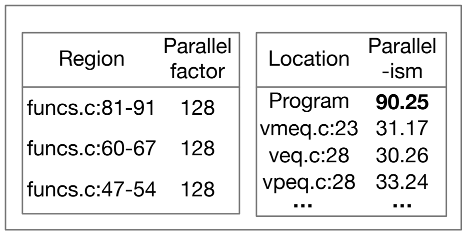
Optimizing `MILCmk`

The `MILCmk` program is a scientific application from the LLNL Coral benchmark suite with 5000 lines of optimized code. This application had a speedup of 2.18× on our 16-core evaluation machine. Figure 6.1(a) shows the parallelism and task runtime overhead profile reported by TaskProf. The profile shows that the program has high task runtime overheads (40% of total useful work) and sufficient parallelism (43.83). The program is spending close to one-third of the execution time in orchestrating parallel execution. The profile also showed that six `parallel_for` calls in the program together account for almost 98% of the task runtime overhead. Figure 6.1(a) shows top three



Location	Parallel -ism	Tasking overhead
Program	**43.83**	0.01
vmeq.c:23	31.17	43.47
veq.c:28	30.26	19.25
vpeq.c:28	33.24	16.71
...	...	...
Tasking overhead : 40.12%		

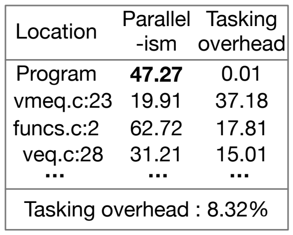
(a) Initial parallelism profile



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

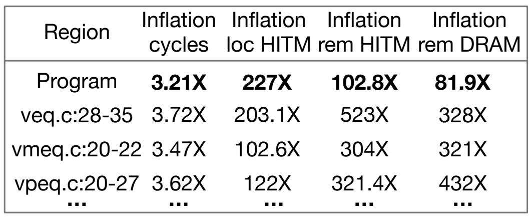
Location	Parallel -ism
Program	**90.25**
vmeq.c:23	31.17
veq.c:28	30.26
vpeq.c:28	33.24
...	...

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



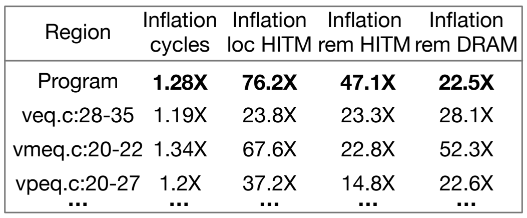
Location	Parallel -ism	Tasking overhead
Program	**47.27**	0.01
vmeq.c:23	19.91	37.18
funcs.c:2	62.72	17.81
veq.c:28	31.21	15.01
...	...	...
Tasking overhead : 8.32%		

(c) Final parallelism profile



Region	Inflation cycles	Inflation loc HITM	Inflation rem HITM	Inflation rem DRAM
Program	**3.21X**	**227X**	**102.8X**	**81.9X**
veq.c:28-35	3.72X	203.1X	523X	328X
vmeq.c:20-22	3.47X	102.6X	304X	321X
vpeq.c:20-27	3.62X	122X	321.4X	432X
...	...	...	...	...

(d) Initial differential profile



Region	Inflation cycles	Inflation loc HITM	Inflation rem HITM	Inflation rem DRAM
Program	**1.28X**	**76.2X**	**47.1X**	**22.5X**
veq.c:28-35	1.19X	23.8X	23.3X	28.1X
vmeq.c:20-22	1.34X	67.6X	22.8X	52.3X
vpeq.c:20-27	1.2X	37.2X	14.8X	22.6X
...	...	...	...	...

(e) Differential profile after optimization

Figure 6.1: Profiles for MILCmk. (a) Initial parallelism profile. (b) Regions identified and the what-if profile. (c) Parallelism profile after concretely parallelizing the reported regions and reducing the tasking overhead. (d) Initial differential profile that reports the inflation in cycles, local HITM, remote HITM, and remote DRAM accesses. (e) Differential profile after addressing secondary effects using TBB's affinity partitioner. We show only the inflation in total work of the top three spawn sites.

calls in the profile. We carefully analyzed the program and increased the cut-off points for these six `parallel_for` calls until the task runtime overhead was less than 10%. As a result, the speedup increased from 2.18× to 5.37×.

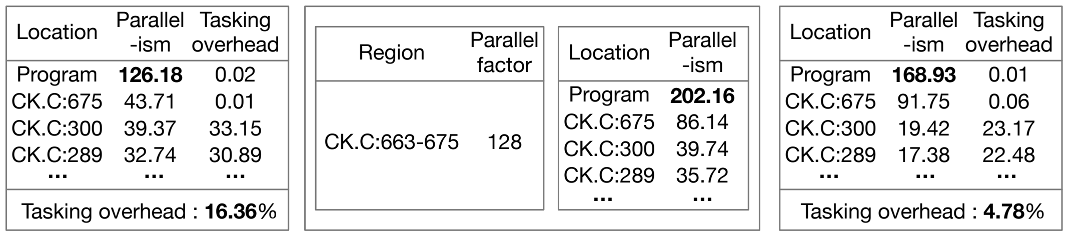
We subsequently used TaskProf's what-if analyses with the input anticipated parallelism set to 128. Figure 6.1(b) presents the three regions reported by TaskProf and the parallelism after hypothetically parallelizing these regions. After carefully examining these reported regions, we found that a part of each reported region was serially computing the sum of a large array of numbers. We parallelized a part of these reported regions with the `parallel_reduce` function in the Intel TBB library, which increased the parallelism to 47.27 and the speedup to 5.67×. Figure 6.1(c) reports the parallelism profile after parallelizing these three regions and reducing the task runtime overhead.

Subsequently, we used TaskProf's differential analyses to determine if the application is experiencing secondary effects. Figure 6.1(d) shows the differential profile, which reports significant inflation with four events (cycles, local HITM events, remote HITM events, and remote DRAM accesses) in the parallel execution when compared to the serial execution. The differential profile in Figure 6.1(d) shows the inflation in top three `parallel_for` regions. On examining them, we found that all the `parallel_for` calls were being made multiple times in a sequential loop. An inflation in remote DRAM accesses in the differential analysis profile led us to suspect that TBB's work stealing scheduler was likely mapping tasks operating on the same data items from multiple invocations of the `parallel_for` to different processors. We explored techniques to maintain affinity between the data and the processor performing computations on the same data over multiple invocations of a `parallel_for` call. We used TBB's affinity partitioner that provides best-effort affinity by mapping the iterations of `parallel_for` to the same thread that executed it previously. We changed the six `parallel_for` calls to use TBB's affinity partitioner. Figure 6.1(e) shows the differential profile after this optimization. It shows a significant decrease for all the four performance counter events. The profile still shows some inflation because the affinity partitioner is a best-effort technique. After this optimization, the speedup of the program improved from 5.67× to 5.92×.

In summary, TaskProf's what-if analyses and differential analyses helped us increase the speedup of MILCmk from 2.18× to 5.92×. The parallelism of the program after all the optimizations also increased from 43.83 to 47.27. Increasing the parallelism further would have resulted in the overhead from creating tasks dominating the execution.

Optimizing `nBody`

The nBody application takes an array of 3-D points as input and computes the gravitational force vector of each point due to all other points. The initial speedup on our 16-core evaluation machine was 12.21×. The parallelism profile (Figure 6.2(a)) generated by TaskProf shows that the program has high parallelism of 126.18. However, the profile also reports a relatively high task runtime overhead of 16.36% of total useful work and three spawn sites corresponding to `parallel_for` calls accounting for 90% of this task runtime overhead. The profile in Figure 6.2(a) shows two such `parallel_for` calls at `CK.C:300`, `CK.C:289`. On careful examination of the code, we observed that these



Location	Parallel -ism	Tasking overhead
Program	**126.18**	0.02
CK.C:675	43.71	0.01
CK.C:300	39.37	33.15
CK.C:289	32.74	30.89
...	...	...
Tasking overhead : **16.36**%		

(a) Initial parallelism profile

Region	Parallel factor
CK.C:663-675	128

Location	Parallel -ism
Program	**202.16**
CK.C:675	86.14
CK.C:300	39.74
CK.C:289	35.72
...	...

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

Location	Parallel -ism	Tasking overhead
Program	**168.93**	0.01
CK.C:675	91.75	0.06
CK.C:300	19.42	23.17
CK.C:289	17.38	22.48
...	...	...
Tasking overhead : **4.78**%		

(c) Final parallelism profile

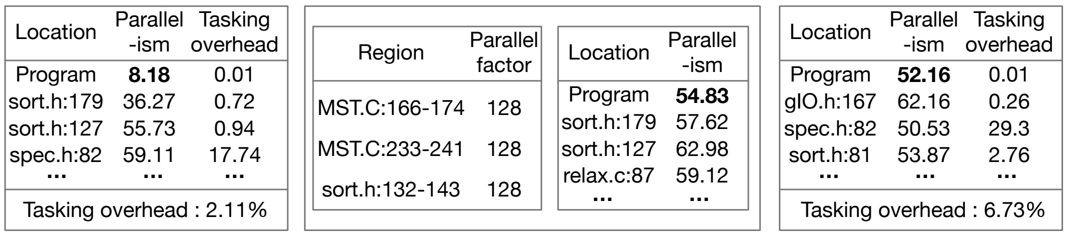
Figure 6.2: The initial parallelism profile with tasking overheads, the regions identified using the what-if analyses, and the parallelism profile after parallelizing the regions reported for `nBody` application.

`parallel_for` calls were nested within other parallel tasks and they were using TBB's default partitioner, which was partitioning the iteration sub-optimally. We changed the code to use a simple partitioner and increased the cut-off until there was reduction in the task runtime overhead. Eventually, we reduced the task runtime overhead to 4.78% and the speedup increased to 13.88×.

Subsequently, the TaskProf's what-if analyses identified a region of the code that when parallelized can increase the parallelism to 202.16 (see Figure 6.2(b)). The region corresponds to the body of a `parallel_for` call at `CK.C:675`, which performs close to 80% of the work on the critical path (see Figure 6.2(a)). We decreased the cut-off to reduce the serial work done by the body of the `parallel_for` call at `CK.C:675`. Figure 6.2(c) reports that the parallelism of the program improved to 168.93 and the task runtime overhead reduced to 4.78% (Figure 6.2(c)). The speedup of the program improved from 12.21× to 14.32×. In summary, we had to increase the cut-off for some spawn sites and decrease the cut-off with a few others to improve speedup. TaskProf enabled us resolve the trade-off between parallelism and task runtime overhead.

Optimizing `minSpanningForest`

The `minimum spanning forest` program in the PBBS suite is a parallel implementation of Kruskal's minimum spanning tree algorithm. The initial speedup of the program is 3.49× over the serial execution. Figure 6.3(a) presents the parallelism profile generated by TaskProf, which reports the program has a parallelism of 8.18. Our what-if



Location	Parallel-ism	Tasking overhead
Program	**8.18**	0.01
sort.h:179	36.27	0.72
sort.h:127	55.73	0.94
spec.h:82	59.11	17.74
...	...	...
Tasking overhead : 2.11%		

(a) Initial parallelism profile

Region	Parallel factor
MST.C:166-174	128
MST.C:233-241	128
sort.h:132-143	128

Location	Parallel-ism
Program	**54.83**
sort.h:179	57.62
sort.h:127	62.98
relax.c:87	59.12
...	...

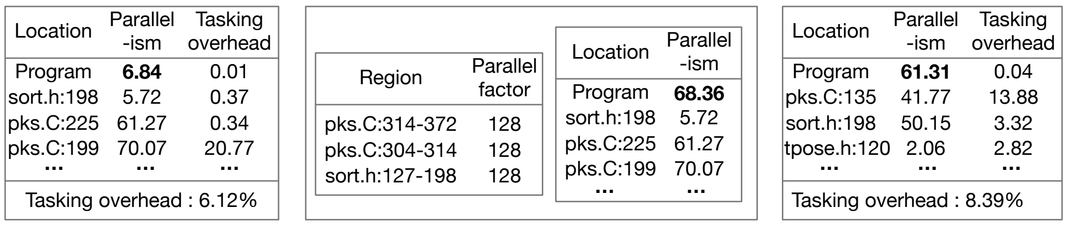
(b) What-if analyses regions and what-if profile

Location	Parallel-ism	Tasking overhead
Program	**52.16**	0.01
glO.h:167	62.16	0.26
spec.h:82	50.53	29.3
sort.h:81	53.87	2.76
...	...	...
Tasking overhead : 6.73%		

(c) Final parallelism profile

Figure 6.3: The initial parallelism profile with tasking overheads, the regions identified using the what-if analyses, and the parallelism profile after parallelizing the regions reported for `minimum spanning forest` application.

analyses identified three regions that can be parallelized to increase parallelism to 54.83 (Figure 6.3(b)) without increasing tasking overheads. On examining the source code for the two regions in `MST.C`, we found that the regions were performing a serial sort. We replaced them with a parallel sort function, which increased the parallelism to 33.12 from 8.18. The third region reported by TaskProf's what-if analyses contained a function that was partitioning edges into multiple blocks. We parallelized the function by recursively spawning tasks and partitioning the edges in parallel. After this optimization, the parallelism of the program increased to 52.16 (Figure 6.3(c)) and the speedup of the program increased to 9.92×.



Location	Parallel-ism	Tasking overhead
Program	**6.84**	0.01
sort.h:198	5.72	0.37
pks.C:225	61.27	0.34
pks.C:199	70.07	20.77
...	...	...
Tasking overhead : 6.12%		

(a) Initial parallelism profile

Region	Parallel factor
pks.C:314-372	128
pks.C:304-314	128
sort.h:127-198	128

Location	Parallel-ism
Program	**68.36**
sort.h:198	5.72
pks.C:225	61.27
pks.C:199	70.07
...	...

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

Location	Parallel-ism	Tasking overhead
Program	**61.31**	0.04
pks.C:135	41.77	13.88
sort.h:198	50.15	3.32
tpose.h:120	2.06	2.82
...	...	...
Tasking overhead : 8.39%		

(c) Final parallelism profile

Figure 6.4: The initial parallelism profile with tasking overheads, the regions identified using the what-if analyses, and the parallelism profile after parallelizing the regions reported for `suffix array` application.

Optimizing `suffixArray`

The `suffix array` program from the PBBS suite takes a set of strings and computes the sorted sequence of all suffixes of the input strings. When we executed the program

Location	Parallel -ism	Tasking overhead
Program	**25.92**	0.01
BFS.C:96	57.43	6.83
gIO.h:197	64.93	0.94
BFS.C:138	49.39	6.74
...	...	...
Tasking overhead : 3.37%		

(a) Initial parallelism profile

Region	Parallel factor
BFS.C:54-138	128
graph.h:140-151	128
graph.h:153-167	128

Location	Parallel -ism
Program	**66.07**
BFS.C:96	57.43
gIO.h:197	64.93
BFS.C:138	49.39
...	...

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

Location	Parallel -ism	Tasking overhead
Program	**41.12**	0.03
BFS.C:96	54.87	5.47
gIO.h:197	64.9	0.72
BFS.C:138	53.32	4.81
...	...	...
Tasking overhead : 3.93%		

(c) Final parallelism profile

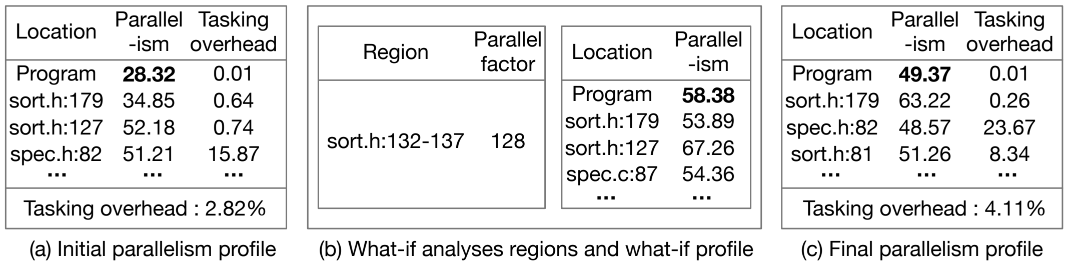
Figure 6.5: The initial parallelism profile with tasking overheads, the regions identified using the what-if analyses, and the parallelism profile after parallelizing the regions reported for `breadth first search` application.

on our 16-core execution machine it had a low initial speedup of 2.09× over the serial execution. The parallelism profile generated by TaskProf indicated a low parallelism of 6.84 (see Figure 6.4(a)). Our what-if analyses identified two regions of the program that can increase the parallelism to 68.36 (Figure 6.4(b)). The first region pointed to a merge function that combined results from the parallel suffix computation using recursive divide and conquer. The second region was performing a transpose function to divide the input string into multiple blocks using recursive divide and conquer. We parallelized both these regions by recursively spawning tasks. After parallelizing them, the parallelism of the program improved to 61.31 (Figure 6.4(c)). The speedup of the program increased from the original 2.09× to 7.38× on a 16-core machine.

Optimizing `breadthFirstSearch`

The breadth first search program in PBBS computes the breadth-first-search tree given a connected undirected graph. The program has a speedup of 6.69× and a parallelism of 25.92 (Figure 6.5(a)). TaskProf's what-if analyses reported three regions that can be hypothetically parallelized to improve the parallelism of the program to 66.07 (Figure 6.5(b)). Among the three regions reported by TaskProf, we were able to design concrete optimizations for two regions.

The breadth first search program performs multiple copy and delete operations on the input graph. TaskProf's what-if analyses highlighted two regions within the copy and delete functions as the regions that will improve the parallelism of the program. In



Location	Parallel -ism	Tasking overhead
Program	**28.32**	0.01
sort.h:179	34.85	0.64
sort.h:127	52.18	0.74
spec.h:82	51.21	15.87
...	...	...
Tasking overhead : 2.82%		

(a) Initial parallelism profile

Region	Parallel factor
sort.h:132-137	128

Location	Parallel -ism
Program	**58.38**
sort.h:179	53.89
sort.h:127	67.26
spec.c:87	54.36
...	...

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

Location	Parallel -ism	Tasking overhead
Program	**49.37**	0.01
sort.h:179	63.22	0.26
spec.h:82	48.57	23.67
sort.h:81	51.26	8.34
...	...	...
Tasking overhead : 4.11%		

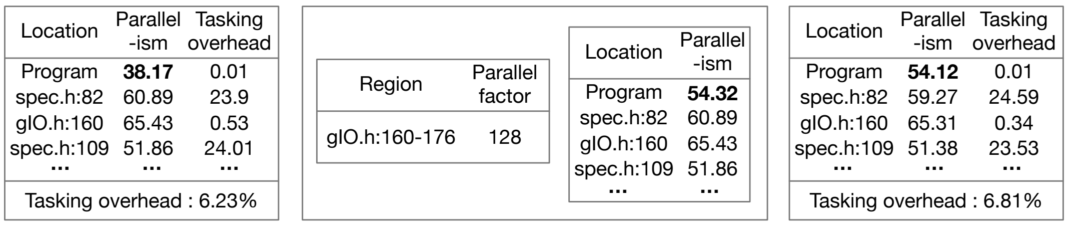
(c) Final parallelism profile

Figure 6.6: The initial parallelism profile with tasking overheads, the regions identified using the what-if analyses, and the parallelism profile after parallelizing the regions reported for `comparison sort` application.

the copy operation, the program creates a new set of vertices and initializes the neighbors of each vertex sequentially in a for loop. We parallelized the for loop that was initializing the neighbors sequentially, using TBB's `parallel_for` function. Similarly, in the delete operation the program was sequentially deallocating all the vertices in the graph. We parallelized the sequential loop in the delete function after determining that the vertices can be deallocated safely in parallel. After the two optimizations, the parallelism of the program increased from 25.92 to 41.12 (Figure 6.5(c)) and the speedup increased from 6.69× to 8.16×.

Optimizing `comparisonSort`

The `comparison sort` program in the PBBS benchmark suite implements a parallel algorithm to sort a sequence of elements given an arbitrary comparison function. The initial speedup of the program is 4.74× over the serial execution. Figure 6.6(a) presents the parallelism profile generated by TaskProf which reports the program has a parallelism of 28.32. TaskProf's what-if analyses identifies a single region that can be parallelized to increase parallelism to 58.38 (Figure 6.6(b)). On examining region reported by TaskProf's what-if analyses, we noticed that the region was sequentially partitioning the elements into separate blocks. We parallelized the region by recursively spawning tasks in parallel, where each task handled the partitioning of a chunk of elements. After this optimization, the parallelism of the program increased to 49.37 (Figure 6.6(c)) and the speedup of the program increased to 6.41×.



Location	Parallel -ism	Tasking overhead
Program	**38.17**	0.01
spec.h:82	60.89	23.9
glO.h:160	65.43	0.53
spec.h:109	51.86	24.01
...	...	...
Tasking overhead : 6.23%		

(a) Initial parallelism profile

Region	Parallel factor
glO.h:160-176	128

Location	Parallel -ism
Program	**54.32**
spec.h:82	60.89
glO.h:160	65.43
spec.h:109	51.86
...	...

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

Location	Parallel -ism	Tasking overhead
Program	**54.12**	0.01
spec.h:82	59.27	24.59
glO.h:160	65.31	0.34
spec.h:109	51.38	23.53
...	...	...
Tasking overhead : 6.81%		

(c) Final parallelism profile

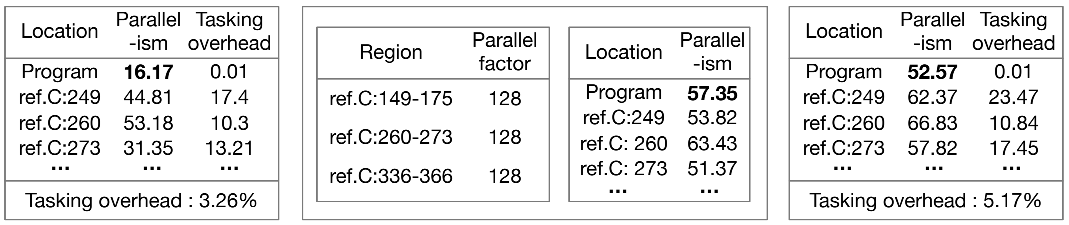
Figure 6.7: The initial parallelism profile with tasking overheads, the regions identified using the what-if analyses, and the parallelism profile after parallelizing the regions reported for `spanning forest` application.

Optimizing `spanningForest`

The spanning forest application in PBBS implements a parallel greedy algorithm to find the spanning forest from an undirected graph. The initial speedup of this program was 7.43× over the serial execution. TaskProf's parallelism profile (Figure 6.7(a)) showed that the program has a parallelism of 38.17. TaskProf's what-if analyses (Figure 6.7(b)) reported a single region in the program that would increase the parallelism to 54.12 if parallelized. The region highlighted by TaskProf was computing the maximum entries of two vector sequentially. Since finding the maximum of a vector is an associative operation, we used Intel TBB's `parallel_reduce` function to divide the vectors in multiple chunks, compute the maximum of each chunk in parallel, and combine the results of each individual chunk to obtain the maximum of the two vectors. This optimization increased the parallelism of the program to 54.12 (Figure 6.7(c)) and the speedup improved to 8.34×. We also separately measured the execution time of the region before and after optimization. Interestingly, the execution time of the specific region improved by 6.45× after our optimization.

Optimizing `DelaunayRefinement`

Th `delaunay refinement` PBBS application takes a set of triangles that form a delaunay triangulation and produces a new triangulation such that no triangle has an angle less than a threshold value. TaskProf's parallelism profile for this program reports a parallelism of 16.17 (see Figure 6.8(a)) and it had a speedup of 2.98×. TaskProf's



Location	Parallel -ism	Tasking overhead
Program	**16.17**	0.01
ref.C:249	44.81	17.4
ref.C:260	53.18	10.3
ref.C:273	31.35	13.21
...	...	...
Tasking overhead : 3.26%		

(a) Initial parallelism profile

Region	Parallel factor
ref.C:149-175	128
ref.C:260-273	128
ref.C:336-366	128

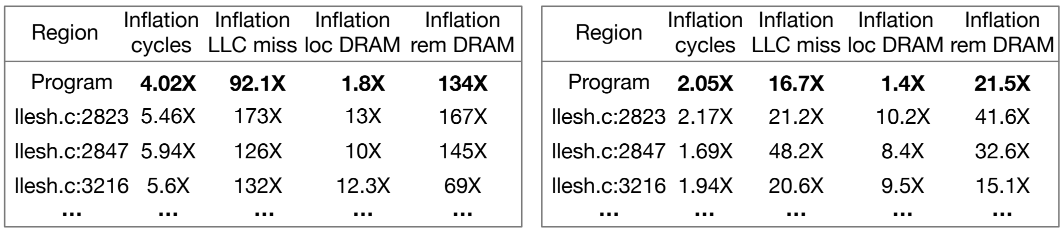
Location	Parallel -ism
Program	**57.35**
ref.C:249	53.82
ref.C: 260	63.43
ref.C: 273	51.37
...	...

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

Location	Parallel -ism	Tasking overhead
Program	**52.57**	0.01
ref.C:249	62.37	23.47
ref.C:260	66.83	10.84
ref.C:273	57.82	17.45
...	...	...
Tasking overhead : 5.17%		

(c) Final parallelism profile

Figure 6.8: The initial parallelism profile with tasking overheads, the regions identified using the what-if analyses, and the parallelism profile after parallelizing the regions reported for `delaunay refinement` application.



Region	Inflation cycles	Inflation LLC miss	Inflation loc DRAM	Inflation rem DRAM
Program	**4.02X**	**92.1X**	**1.8X**	**134X**
llesh.c:2823	5.46X	173X	13X	167X
llesh.c:2847	5.94X	126X	10X	145X
llesh.c:3216	5.6X	132X	12.3X	69X
...	...	...	...	...

(a) Intial differential profile

Region	Inflation cycles	Inflation LLC miss	Inflation loc DRAM	Inflation rem DRAM
Program	**2.05X**	**16.7X**	**1.4X**	**21.5X**
llesh.c:2823	2.17X	21.2X	10.2X	41.6X
llesh.c:2847	1.69X	48.2X	8.4X	32.6X
llesh.c:3216	1.94X	20.6X	9.5X	15.1X
...	...	...	...	...

(b) Differential profile after optimization

Figure 6.9: (a) The differential profile for LULESH showing the inflation in cycles, last level cache misses, local DRAM, and remote DRAM accesses. (b) The profile after reducing the secondary effects at `lulesh.c:2823` and `lulesh.c:2847`.

what-if analyses reported three regions (see Figure 6.8(b)) that would increase the parallelism of the program to 54.32 on parallelization. On carefully inspecting the source code, we found that the identified regions were sequentially processing disjoint sets vertices of the triangles using a for loop. Since these vertices could be safely processed in parallel, we replaced the sequential for loops in all the three regions with TBB's `parallel_for` calls. This optimization improved the parallelism to 52.57 (see Figure 6.8(c)) and the speedup increased from 2.98× to 7.13×.

Optimizing `LULESH`

LULESH [94] is an application developed at Lawrence Livermore National Labs and is widely used to model hydrodynamics in scientific applications. The program had a speedup of 4.23× on our 16-core machine. The parallelism of the program was 42.48. We wanted to understand the reason behind low speedup even when the program has a

Location	Parallel -ism	Tasking overhead
Program	**73.75**	0.01
HJM_SimPath.cpp:135	38.74	82.65
HJM_Securities.cpp:297	74.21	17.34
Tasking overhead : **47.36**% of total work		

(a) Initial parallelism profile

Location	Parallel -ism	Tasking overhead
Program	**49.77**	0.02
HJM_SimPath.cpp:135	18.29	47.2
HJM_Securities.cpp:297	48.39	52.78
Tasking overhead : **6.74**% of total work		

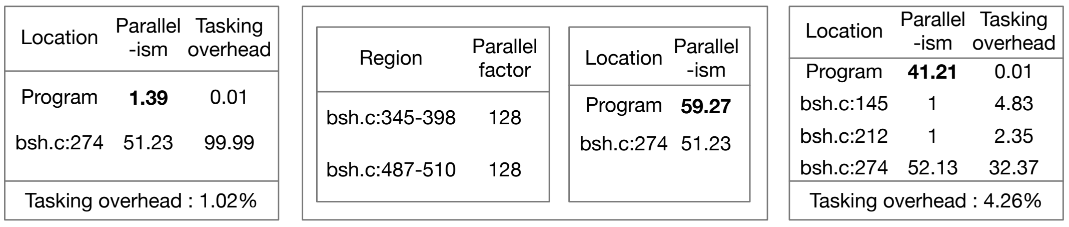
(b) Final parallelism profile

Figure 6.10: (a) The parallelism profile for `swaptions` that highlights high tasking overhead. (b) The profile after reducing the tasking overhead by increasing the grain size at `HJM_SimPath.cpp:135`.

parallelism of 42.48. We profiled the program with TaskProf's differential analysis. Figure 6.9(a) shows the differential profile generated by TaskProf. Overall, the program has 4.02× inflation in cycles when compared to serial execution. The inflation in the critical path is also relatively high at 4.02× (see Table 6.2). The profile also shows significant inflation in last level cache misses and remote memory accesses. The profile highlights two `parallel_for` calls (llesh.c:2823 and llesh.c:2847 in Figure 6.9(a)) for having high inflation in last level cache (LLC) misses and remote DRAM accesses. Further, these `parallel_for` calls were performing almost 30% of the work on the critical path. Since these regions had high inflation in LLC misses, we checked whether the working set of the program was larger than the LLC during parallel execution. We noticed that both the `parallel_for` regions were performing computations on two large arrays. Accessing the arrays in parallel was causing a large number of last level cache misses. We rearranged the computation to reduce the working set size while ensuring that the transformation was correct. Figure 6.9(b) shows the reduction in inflation for all events after this optimization. The optimization improved the speedup of the program to 5.9×.

Optimizing `swaptions`

The `swaptions` program from the Parsec benchmark suite has an initial speedup of 12.21× on our 16-core machine. Although this program has relatively higher speedup, the profiler reported that the program has a tasking overhead of 47.36% (Figure 6.10(a)).



Location	Parallel -ism	Tasking overhead
Program	**1.39**	0.01
bsh.c:274	51.23	99.99
Tasking overhead : 1.02%		

(a) Initial parallelism profile

Region	Parallel factor
bsh.c:345-398	128
bsh.c:487-510	128

Location	Parallel -ism
Program	**59.27**
bsh.c:274	51.23

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

Location	Parallel -ism	Tasking overhead
Program	**41.21**	0.01
bsh.c:145	1	4.83
bsh.c:212	1	2.35
bsh.c:274	52.13	32.37
Tasking overhead : 4.26%		

(c) Final parallelism profile

Figure 6.11: The initial parallelism profile, the regions identified using the what-if analyses, and the parallelism profile after parallelizing the regions reported for `blackscholes` application.

The parallelism profile generated by TaskProf highlights the `parallel_for` call at `HJM_SimPath.cpp:135`, which accounts for more than 80% of the tasking overhead. On examining the code, we found that the cut-off for recursive decomposition was too small. We increased the cut-off with the help of TaskProf until the overall tasking overhead reduced to less than 10%. Figure 6.10(b) presents the parallelism profile after reducing the tasking overhead. The speedup of the program increased to 14.19×. TaskProf's what-if analyses subsequently reported that the program cannot be parallelized any further without increasing the tasking overhead.

Optimizing `blackscholes`

The `blackscholes` application from the PARSEC suite [24] computes the price of a portfolio of options using partial differential equations. The initial speedup in the program was 1.14× over the serial execution. TaskProf's parallelism profile indicates that `blackscholes` has low parallelism of 1.39 (see Figure 6.11(a)). This program has a single `parallel_for` that has reasonable parallelism of 51.23. TaskProf's what-if analyses in Figure 6.11(b) highlights two regions that must be parallelized to reach a maximum parallelism 59.27. Our examination of the code for the two regions revealed that the regions were reading and writing serially. We split the input and output into multiple files and parallelized the input and output operations which increased the parallelism to 41.21 (Figure 6.11(c)) and the speedup increased from 1.14× to 8.15×.

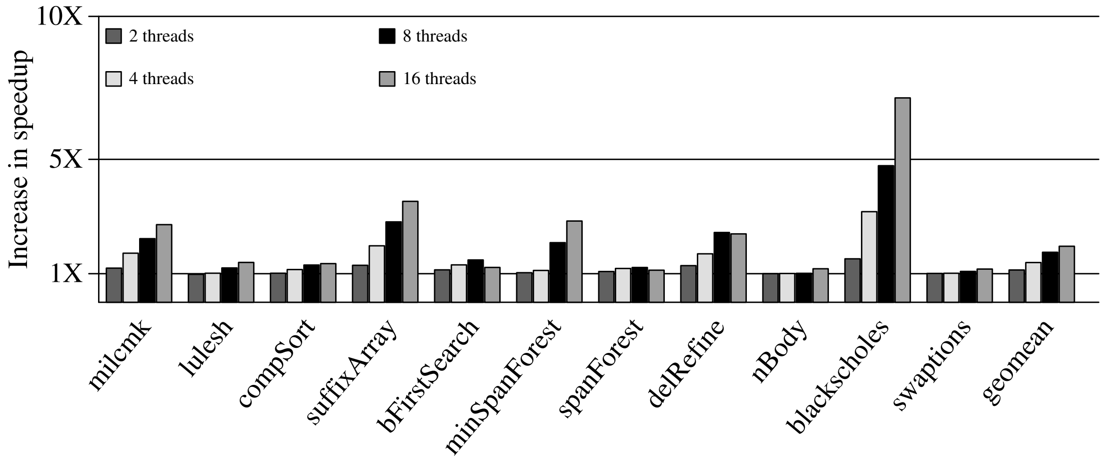


Figure 6.12: For all applications we sped up using TaskProf, graph shows the increase in the speedup after optimization over the original speedup (speedup after optimization/original speedup) for executions with 2, 4, 8, and 16 threads. An increase in speedup greater than 1X implies that optimization sped up the execution.

Speedup Improvements on Varying Thread Counts

While all the programs we optimized improve the speedup in the execution with sixteen threads, we also evaluate if the optimizations improve speedup on varying number of threads. Figure 6.12 shows the increase in the speedup after optimization over the original speedup for all the programs we sped up using TaskProf. For each program, the figure shows four bars representing the increase in speedup on executions with two, four, eight, and sixteen threads. We compute the increase in speedup as the ratio of the speedup after optimization over the original speedup on a execution with a given number of threads. Hence an increase in speedup that is greater than $1\times$ indicates that the optimization improved the speedup for the execution.

Figure 6.12 shows that the optimizations do not slowdown the programs on any number of threads, since the increase in speedup for all the programs is at least $1\times$. Although, in executions with lesser number of threads (two or four threads), most of the programs show a nominal increase in speedup. This is because the programs before optimization have sufficient parallelism to achieve scalable speedup on execution with lesser number of threads, and introducing the optimizations does not increase the speedup significantly. For programs like `suffix array`, `delaunay refinement`, and

Table 6.4: The parallelism computed and the number of regions identified by TaskProf while executing each application using 4 threads.

Application	Parallelism	No. of regions
MILCmk	51.27	3
LULESH	49.14	2
compSort	28.26	2
integerSort	36.34	2
remDups	46.29	1
dictionary	40.79	1
suffixArray	6.63	3
BFS	27.21	3
maxIndSet	26.19	2
maxMatching	47.24	1
minSpanForest	7.97	3
spanForest	38.71	1
convexHull	68.26	0
nearestNeigh	16.35	4
delTriang	58.45	1
delRefine	17.28	2
rayCast	51.68	1
nBody	128.24	0
blackscholes	1.43	2
bodytrack	31.45	2
fluidanimate	64.16	1
streamcluster	74.26	5
swaptions	74.13	0

`blackscholes`, which have relatively lower parallelism the execution with lesser number threads (two and four threads) shows a significant increase in speedup.

Among all the programs that we sped up, six programs show a near constant increase in speedup for executions with increasing number of threads. For all of these programs, the speedup of the original program scaled similar to the speedup of the program after optimization. In contrast, for programs like `suffix array` and `blackscholes` the increase in speedup scales as the number of threads are increased. This is because the speedup in the programs before optimization does not scale with the number of threads. In addition to increasing the speedup in these programs, the optimizations we implemented with insights from TaskProf enabled the programs to attain scalable speedup.

6.3.2 Identifying Scalability Bottlenecks with Lower Thread Counts

TaskProf's parallelism computation and what-if analyses provide the programmer the ability to profile the program on a given number of processors and identify scalability bottlenecks that can occur when the program is executed on a larger number of processors. To evaluate the effectiveness of TaskProf in identifying scalability bottlenecks, we profiled all the twenty three applications on the same sixteen-core machine but restricted the execution to four cores.

Table 6.4 presents the parallelism measured and the number of regions identified by TaskProf when the applications are executed with four threads. The table shows that the parallelism computed by TaskProf when executed with four threads is similar to the parallelism computed with sixteen threads for most of the applications. The parallelism of two applications, `MILCmk` and `LULESH`, is slightly higher for the execution with four threads than the execution with sixteen threads (see Table 6.2 for parallelism on sixteen threads). The two applications have higher parallelism in the execution with four threads because the four-threaded execution is not experiencing the secondary effects that are present in the execution with sixteen threads. The secondary effects in the execution with sixteen threads are causing an inflation in the total work and a higher inflation in the critical path work, which results in the parallelism being lower than the four-thread execution.

We ran TaskProf's what-if analyses on all the applications using four threads with the anticipated parallelism set to 128 (same as the execution with sixteen threads). In all the applications except `nBody`, TaskProf's what-if analyses identified the same number of regions in the four-threaded execution as the execution with sixteen threads (see Table 6.4). The parallelism of `nBody` in the four-thread execution increases slightly over the parallelism in the sixteen-thread execution, which results in the parallelism being greater than the anticipated parallelism of 128. Since, the parallelism measured in the four-thread execution is already greater than the anticipated parallelism, TaskProf's what-if analyses does not report any regions.

Effectiveness Summary

The case studies demonstrate that TaskProf is effective in identifying the parts of the program that are the performance bottlenecks in task parallel programs. We show that TaskProf identifies scalability bottlenecks by profiling the programs on smaller numbers of processors and identifying the bottlenecks that occur in an execution with a larger number of processors. TaskProf provided us simple, yet, crucial insights into the performance of the programs that were profiled and directly guided us to the code optimization that we eventually implemented. Overall, TaskProf enabled us to quantify the logical parallelism of programs, find sources of task runtime overhead, and identify parts of a program that matter for improving the parallelism and addressing secondary effects.

6.4 Evaluation with Other Profilers

To highlight the effectiveness of TaskProf, we also evaluated all applications with three other profilers: Coz [51], Intel Advisor [47], and Intel VTune Amplifier [48].

6.4.1 Evaluation with Coz

Coz is a profiler for multi-threaded programs that quantifies the possible speedup of the program from optimizing a line of code in the program. Coz highlights the lines of code that speed up the program as the parts of the program that matter for optimization. Coz provides three modes for profiling applications: (1) end-to-end sampling, (2) latency profiling, and (3) throughput profiling. In the end-to-end profiling mode, Coz profiles the entire program execution and reports the lines of code in the program that have the most impact on the speedup of the entire program. In the throughput and the latency profiling modes, the programmer is required to annotate the input program with *progress points* that represent the primary functionality of the program (for *e.g.*, the primary for loop in a program). Coz treats the progress points as proxies for the whole program speedup. In the throughput profiling mode, Coz reports the lines of code that impact the speedup based on the change in the rate of visits (throughput) to a progress point.

Table 6.5: Summary of the results from profiling all applications with Coz in end-to-end, throughput, and latency profiling modes. For each profiling mode, table shows the number of profiling runs, the number of lines highlighted, and the speedup estimated by Coz. Negative speedup indicates that Coz estimated a slowdown from optimizing the corresponding lines. For applications that did not report any lines to optimize after 90 runs, the entry for the number of lines highlighted is set to 0.

Application	End-to-end profiling mode			Throughput profiling mode			Latency profiling mode		
	No. of runs	No. of lines	Speedup	No. of runs	No. of lines	Speedup	No. of runs	No. of lines	Speedup
MILCmk	20	3	4%	30	1	3%	60	2	-8%
LULESH	20	1	-2%	30	5	40%	30	3	-50%
compSort	30	1	5%	60	1	1%	30	1	1%
integerSort	60	2	5%	70	1	12%	40	2	-4%
remDups	30	1	8%	30	1	25%	30	1	-80%
dictionary	20	2	3%	30	1	10%	30	1	-4%
suffixArray	40	2	-5%	30	2	20%	30	2	-20%
BFS	30	1	4%	50	1	10%	90	1	-60%
maxIndSet	40	2	6%	30	1	5%	30	1	2%
maxMatching	20	2	3%	40	3	11%	30	2	-15%
minSpanForest	30	1	4%	30	1	5%	70	3	-8%
spanForest	90	0	0	90	0	0	50	2	-50%
convexHull	60	2	14%	90	0	0	90	0	0
nearestNeigh	40	2	6%	30	4	13%	30	2	-20%
delTriang	30	2	1%	20	2	4%	20	1	2%
delRefine	60	2	8%	30	1	60%	60	1	1%
rayCast	30	2	4%	20	2	15%	20	3	-10%
nBody	60	2	1%	60	3	30%	60	4	20%
blackscholes	30	1	-1%	30	5	50%	40	4	3%
bodytrack	20	3	4%	40	2	7%	50	2	-1%
fluidanimate	30	4	5%	60	5	10%	60	3	-5%
streamcluster	30	2	7%	50	3	20%	60	2	1%
swaptions	20	1	30%	20	4	65%	30	2	-50%

In the latency profiling mode, Coz reports the lines of code that impact the speedup based on the change in the latency between progress points.

Table 6.5 presents the results from profiling all the twenty three applications with Coz. For each profiling mode, it shows the number of profiling runs taken by Coz to produce a profile, the number of lines of code highlighted in the profile, and the maximum speedup or slowdown estimated by Coz from optimizing the highlighted lines. To generate a profile with Coz in any profiling mode, we had to execute the input program with Coz at least 20 times and in many cases up to 90 times. We were unable to generate a profile for the `spanning forest` (in end-to-end and throughput modes) and `convex hull` (in throughput and latency modes) applications even after 90 executions with Coz.

In the end-to-end profiling mode, Coz highlights 1-4 lines of code in all applications as the parts of the program that either increase the speedup or cause a slowdown on optimization. In most applications where Coz identifies an increase in the speedup, it estimates an increase in the speedup between 1-8%. On examining the lines of code highlighted by Coz, we found that the changes required to optimize the lines would modify the program's output. For instance, in the `convex hull` application Coz highlights a parallel for loop that initializes a large array to zero based on a condition (if statement). But, optimizing the parallel for loop with a call to `memset` was not feasible since elements in the array that were initialized to zero were not consecutive. Furthermore, in most of the applications the lines highlighted by Coz did not belong to the regions highlighted by TaskProf's what-if analyses. In `swaptions`, Coz identifies a line of code in the parallel for loop that TaskProf highlighted as having high task runtime overhead (see Figure 6.10(a)). The goal of Coz is to identify lines of code that matter in increasing the speedup. In contrast to Coz, TaskProf provides additional feedback about the factors affecting the speedup like task runtime overhead, which are useful in designing concrete optimizations.

Subsequently, we profiled the applications with Coz in both the latency and the throughput profiling modes by placing progress points in the regions where the programs are performing their primary computation. The latency profiles generated by Coz showed a slowdown for most of the applications with the `remove duplicates` program showing the maximum slowdown of 80%. For six applications, the latency profiles showed a nominal increase in the speedup in the range of 1-3%. In the throughput profiling mode, Coz reports an increase in speedup for all of the applications. In five applications, `BFS`, `suffix array`, `delaunay refinement`, `blackscholes`, and `swaptions`, the lines highlighted by Coz lie within the regions identified by TaskProf's what-if analyses. However, the estimated speedup reported by Coz was lesser than the actual speedup we achieved after parallelizing the regions. For instance, Coz reported a speedup of 20% for the `suffix array` program. However, we improved the speedup by 253% when we concretely parallelized the region reported by TaskProf.

Table 6.6: Summary of the results from evaluating all applications with Intel Advisor. For each application, the table presents the number of regions highlighted by Intel Advisor, the speedup predicted by Intel Advisor on parallelizing the regions highlighted by it, and if parallelizing the regions highlighted by Intel Advisor improved the actual speedup.

Application	No. of regions	Predicted speedup	Did parallelization increase speedup?
MILCmk	5	38%	✗
LULESH	1	53%	✗
compSort	2	26%	✓
integerSort	2	32%	✗
remDups	3	22%	✗
dictionary	3	18%	✗
suffixArray	5	64%	✗
BFS	4	33%	✗
maxIndSet	2	26%	✗
maxMatching	2	28%	✗
minSpanForest	4	39%	✗
spanForest	1	64%	✗
convexHull	2	34%	✗
nearestNeigh	3	19%	✗
delTriang	5	62%	✗
delRefine	4	54%	✗
rayCast	2	32%	✗
nBody	5	65%	✓
blackscholes	1	48%	✗
bodytrack	2	23%	✗
fluidanimate	3	42%	✗
streamcluster	5	35%	✗
swaptions	3	81%	✗

In summary, the goal of Coz is to measure the effect of optimizing a line of code on the overall speedup of the program. In optimizing a line, Coz does not consider other effects on the performance like adding parallelism, reducing task runtime overheads, and addressing secondary effects. Further, Coz estimates the speedup from optimizing lines individually. In our experiments with TaskProf, we found that optimizing regions of code in isolation may not increase the parallelism and the speedup of the program. The programmer will have to optimize multiple regions together to increase the speedup.

6.4.2 Evaluation with Intel Advisor

Intel Advisor is an analysis tool for tuning the performance of both multi-threaded programs and task parallel programs. It consists of a survey analysis that reports the regions in the program that consume the highest amount of time. Based on the regions reported by the initial survey analysis, programmers can select certain regions of code and assess the impact of parallelizing the regions on the speedup of the program. The programmer can use annotations to specify the selected regions to Intel Advisor. Intel Advisor reports the estimated speedup from parallelizing the annotated region on systems with various processor counts. Further, Intel Advisor checks the suitability of the region of code for optimization to address load imbalance, lock contention, or runtime overheads.

We first ran Intel Advisor's survey analysis on all the applications to identify the regions that consume the highest amount of time. Table 6.6 shows the number of regions reported by Intel Advisor for each application. In all the applications, Intel Advisor reported serial and parallel for loops as the most time consuming parts of the program. To identify the loops that matter for parallelization, we annotated the loops and executed the programs with Intel Advisor to check if the annotated regions improve the speedup on parallelization. Intel Advisor reported a speedup in range of 18-81% on parallelizing the loops reported by the initial survey analysis (see Table 6.6). On examining the for loops we found that the serial for loops in all the applications, except `comparison sort`, were not parallelizable due to existing dependencies across loop iterations. Except the parallel for loop reported in `nBody`, further parallelizing all the other parallel for loops did not increase the actual speedup of the programs. The loops reported in `nBody` and `comparison sort` by Intel Advisor were the same regions that we identified using TaskProf's what-if analyses. All other loops reported by Intel Advisor were not the regions highlighted by TaskProf's what-if analyses.

Subsequently, we annotated the regions identified by TaskProf's what-if analyses to obtain an estimate of the improvement in speedup of the program on parallelizing the regions identified by TaskProf. Of the nine applications that had parallelism bottlenecks, Intel Advisor reports a speedup in the range 28%-65% for five applications (applications

`BFS`, `suffixArray`, `spanForest`, `compSort`, and `nBody`). If the annotations are within loops, Intel Advisor predicts a reasonable speedup (*e.g.*, `nBody` application). For the remaining four applications Intel Advisor reports a nominal speedup in the range of 1%-6% (`MILCmk`, `LULESH`, `delRefine`, and `blackscholes`). The annotated regions in these programs were serial sections of the program between the parallel loops, which we eventually parallelized by recursively spawning parallel tasks. Overall, we found that Intel Advisor's speedup prediction is useful if the annotated regions are within loops and are also hot-spots in the program. If the annotated regions involved serial sections which can be parallelized by other approaches like recursive fork-join, Intel Advisor can underestimate the speedup from optimization.

6.4.3 Evaluation with Intel VTune

Intel VTune Amplifier is a comprehensive commercial profiler that identifies various sources of performance inefficiencies in parallel programs. VTune uses sampling to profile the input program and identify the hot-spots in the program, which are the regions in the program where the program execution spends the most time. In addition to identifying the hot-spots, VTune indicates if the input program is experiencing low parallelism, high task scheduling overhead, and secondary effects.

Table 6.7 shows the results from profiling all the applications with VTune. It specifies the number of regions reported by VTune, and indicates if VTune flagged the program as having low parallelism, high task scheduling overhead and high secondary effects. When VTune profiles an application, it identifies hot-spots in the entire application including in any runtime libraries that are used by the application. We found that the hot-spots reported by VTune profiles in all applications include regions that belong to runtime libraries like the Intel TBB library, and `libc`. Reporting regions other than application program is not useful and can often mislead the programmer.

Among the twenty three applications, VTune identifies that thirteen applications have low parallelism (see Table 6.7). However, the parallelism measured by VTune is based on the CPU utilization during a specific execution and does not measure the logical parallelism expressed in the program. To address performance issues due to low

Table 6.7: Performance bottlenecks and hotspots reported by Intel VTune. For each application, table uses a checkmark (✓) to indicate if Intel VTune reports low parallelism, high task scheduling overhead, or high secondary effects.

Application	Hotspots	Low parallelism	High scheduling overhead	High secondary effects
MILCmk	5		✓	✓
LULESH	4	✓		✓
compSort	3	✓	✓	
integerSort	4	✓		
remDups	2			✓
dictionary	3			✓
suffixArray	3	✓	✓	
BFS	2	✓		✓
maxIndSet	3	✓		
maxMatching	3	✓		
minSpanForest	4	✓	✓	
spanForest	2	✓		
convexHull	3			✓
nearestNeigh	5	✓		
delTriang	2			✓
delRefine	3	✓		
rayCast	4			✓
nBody	1			✓
blackscholes	1	✓		
bodytrack	2	✓		
fluidanimate	1			✓
streamcluster	3			✓
swaptions	1		✓	

parallelism, VTune identifies the hotspots in the program. However, optimizing the hot-spots, which were primarily `parallel_for` parts of the code did not improve the speedup for the programs.

Of the three applications that TaskProf reported as having high task runtime overhead, VTune identified the scheduling overhead in two out of the three applications. It did not identify the scheduling overhead in nBody. Intel VTune reports that eleven out of the twenty three applications are experiencing some kind of secondary effects (see Table 6.7). While VTune highlighted various causes for the secondary effects in the programs, it did not provide specific code regions that were experiencing the secondary effects. Overall, VTune is a useful tool to identify hot-spots, secondary effects, and scheduling overheads. However, it does not report whether optimizing these hot-spots matter in improving parallelism or speedup of the program. Further, it does not provide feedback about the specific regions in the program that are experiencing secondary effects.

6.5 Efficiency in Profiling

In this section, we evaluate the performance and efficiency of TaskProf using the set of twenty three applications listed in Table 6.1. To evaluate the efficiency of TaskProf, we measure the execution time overhead and memory overhead in profiling all the applications with TaskProf. We measure the overheads in the parallel as well as the serial executions since TaskProf executes the input programs serially and in parallel to perform what-if and differential analyses. For all the timing and resident memory measurements, we execute the program five times and consider the average of the five executions.

In evaluating the efficiency of TaskProf, we measure the overheads from both the offline and the on-the-fly profiling techniques. In the offline profiling mode, we measure the overhead from constructing the performance model, measuring work using hardware performance counters, and writing the performance model to a file. In the online mode, in addition to constructing the performance model and measuring the work using hardware performance counters, we also perform the parallelism and what-if analyses computations. Our goal is to ascertain the extra overhead added by the on-the-fly computation over merely measuring the work in the offline profiling mode.

6.5.1 Execution Time Overhead

Figure 6.13 shows the execution time overhead of profiling all the applications in parallel over the baseline parallel execution of the applications without any profiling. We measured the overhead while executing the program in parallel using 16 threads on our evaluation machine. For each application, the figure shows two bars. One, showing the overhead of the offline profiling mode, and the other showing the overhead of the

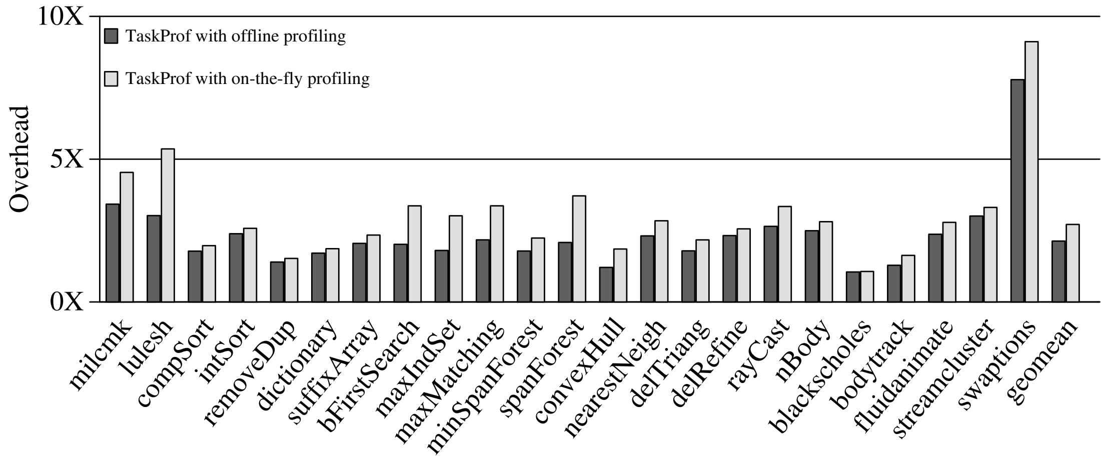


Figure 6.13: Execution time performance overhead of TaskProf's offline and on-the-fly profiling compared to a baseline execution without any profiling. Both profiling (offline and on-the-fly) and baseline executions use 16 threads. As each bar represents the overhead, smaller bars are better.

on-the-fly profiling mode. Smaller bars are better as they represent lower execution time overheads. The mean overhead of TaskProf's offline and on-the-fly profiling modes are 2.13× and 2.71×, respectively. As expected, the overhead of the on-the-fly profiling mode is greater than the offline profiling mode, since the on-the-fly mode performs computation to measure the parallelism and perform what-if and differential analyses while profiling. However, with only a nominal increase in overhead, the on-the-fly analysis enables TaskProf to profile large applications, which would not have been possible with the offline mode which requires the entire performance model to be written to disk.

Although the mean execution time overhead of all the applications is lesser than 3×, two applications (`LULESH`, and `swaptions`) have overhead greater than 5×. The performance model for these applications contains a relatively large number of nodes (greater than a 100 million nodes in contrast with a 100 thousand nodes in the other applications). However, the overhead in these applications is not from constructing the relatively large performance model. Rather, it is from starting and stopping the hardware performance counter measurement relatively frequently at the starting and ending every async and step node in the performance model.

The largest component of the execution time overhead in the applications is from using the `perf_event` API to measure the work in every step node and async node. In

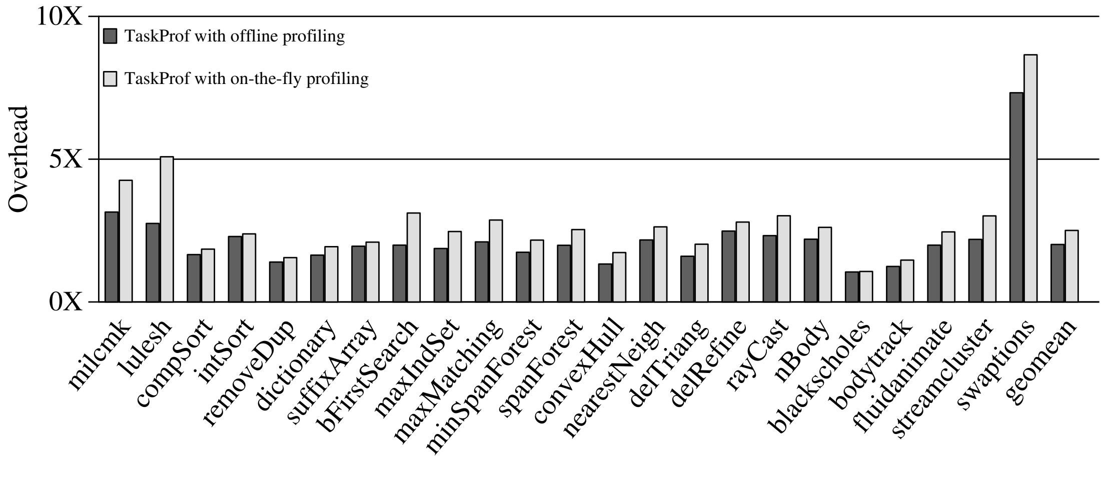


Figure 6.14: Execution time performance overhead of the serial execution of TaskProf's offline and on-the-fly profiling compared to the serial baseline execution without any profiling.

fact, we measured the overhead with just measuring the work in the step nodes and without measuring the task creation work in the async nodes. The average execution time overhead by measuring just the work in the step nodes reduced from 2.71× to 1.56×. Hence, we provide a command-line option to measure just the work in the step nodes if the programmer is just interested in knowing the parallelism of a program.

Figure 6.14 shows the execution time overhead of profiling the applications serially over the baseline serial execution of the applications without any profiling. To measure the execution time overhead of the serial execution, we perform the same amount computation and spawn the same number of tasks as the parallel execution, but execute the program using a single thread. The geometric mean of the overhead of the offline and on-the-fly serial executions are 2.01× and 2.5×, respectively. The mean execution time overhead of the serial profile execution is slightly lesser than the mean execution time of the parallel profile execution. This is because of two factors. (1) Our implementation uses a lock to synchronize the transfer of a performance model node from the thread that is creating the node to the thread that is actually executing the task corresponding to the created node. The threads can be different because of randomized mapping of tasks to threads by the runtime scheduler. (2) Our implementation of the on-the-fly profiler uses atomic variables to maintain the quantities on each node of the performance model

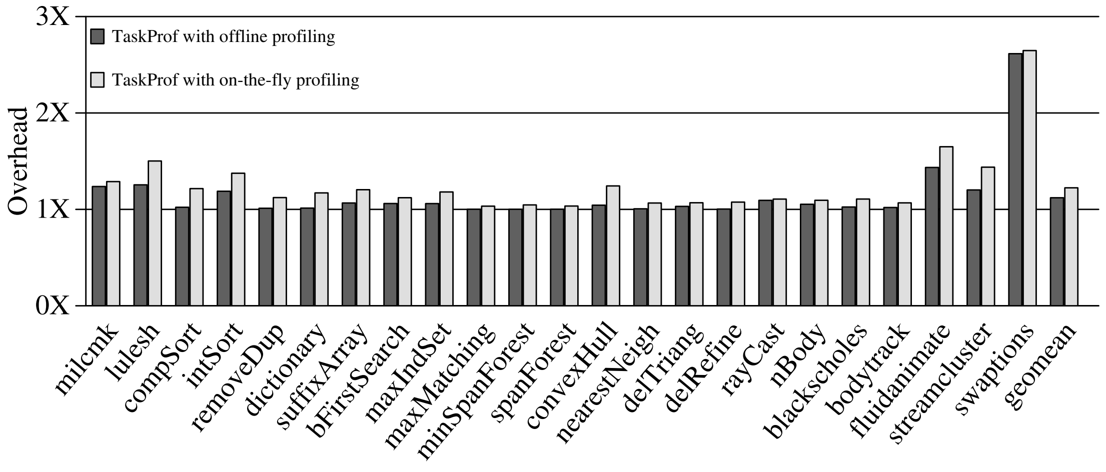


Figure 6.15: Resident memory overhead of TaskProf's offline and on-the-fly profiling compared to a baseline execution without any profiling. Both profiling (offline and on-the-fly) and baseline executions use 16 threads.

required to compute the parallelism. Since there would be no contention on the lock or the atomic variables in the serial profiling executions, the overhead of serial profiling is lesser than the overhead of profiling in parallel. However, the largest the component of the overhead in the serial profile execution is from the system calls to start and stop hardware performance counters.

6.5.2 Memory Overhead

To measure the memory overhead of a program execution, we use the `proc` utility available on Unix-based systems to read the resident memory set size. Figure 6.15 shows the memory overhead of profiling the applications in parallel over the baseline parallel execution without profiling. For each application, the figure shows two bars which represent the memory overheads in the offline and the on-the-fly profiling modes. The mean memory overhead of the offline profiler is 1.12× and that of the on-the-fly profiler is 1.22×. The memory overhead of the on-the-fly profiler is slightly higher than the overhead of the offline profiler since the on-the-fly profiler maintains data with each node in the performance model to perform the parallelism computation.

Among all the applications, only the `swaptions` application has a relatively high overhead of 2.6×. While the performance model for `swaptions` has a large number of

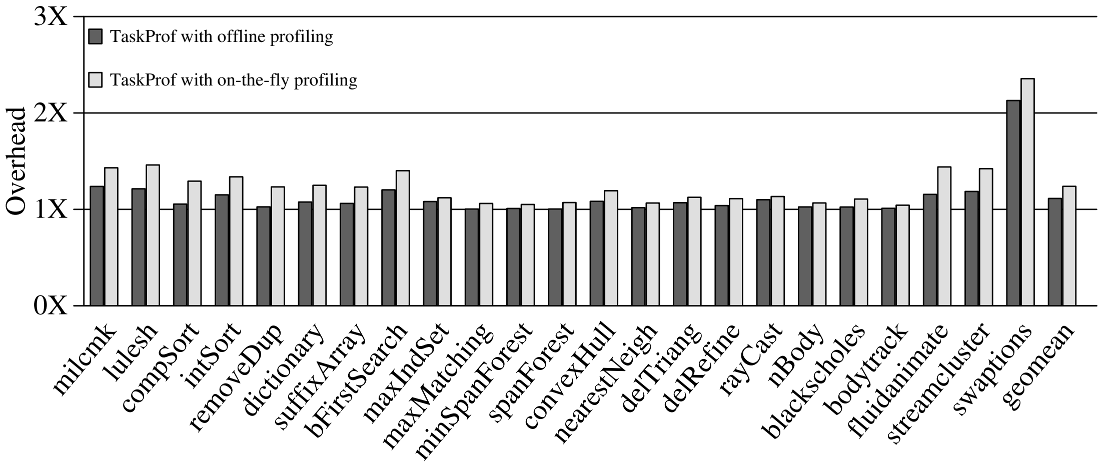


Figure 6.16: Memory overhead of the serial execution of TaskProf's offline and on-the-fly profiling compared to the serial baseline execution without any profiling.

nodes (144 million), a majority of these nodes are created in two `parallel_for` calls. Hence, a large fraction of the nodes in the performance model is resident in memory together, which is causing the high memory overhead. Other applications like `LULESH` and `MILCmk` have a low memory overhead despite having a large number of nodes in their respective performance model. This is because the nodes in the performance models are spread over multiple `parallel_for` calls and only a small fraction is resident in memory together.

Figure 6.16 shows the memory overhead of profiling the applications serially over the baseline serial execution of the applications without any profiling. The geometric mean of the memory overhead of the offline and on-the-fly serial executions are 1.11× and 1.22×, respectively. The low memory perturbation of the serial profile execution in both the modes makes it an ideal approximation for the oracle execution.

6.6 Usability Study with Programmers

We conducted a user study to evaluate the usability of TaskProf's what-if analyses with average programmers. The user study had thirteen participants: twelve graduate students and one senior undergraduate student. Among all the participants, two students had 4+ years of experience in parallel programming, five students had some prior experience, four students had passing knowledge, and two students had no prior experience with parallel programming. The total duration of the user study was four hours. To ensure that every student had some knowledge in parallel programming, we provided a 2-hour tutorial on task parallelism, and on writing and debugging task parallel programs using Intel TBB. We gave multiple examples to demonstrate parallelism bottlenecks and described various ways in which parallelism bottlenecks can manifest in task parallel programs.

After the tutorial, the participants were given a total of four applications and were asked to identify parallelism bottlenecks without using TaskProf in a one hour time period. Among them, three applications — minSpanForest, convexHull, and blackscholes — were from Table 6.1 and one application was a simple treesum program that spawned recursive tasks to compute the sum of all the values in the nodes of the tree. We chose these applications as they had varying levels of difficulty in diagnosing parallelism bottlenecks. We asked the participants to identify the static region of code in the programs causing low parallelism and record the time they spent in analyzing each program. The participants were simply required to identify the bottlenecks and were not required to design any optimization to speed up the programs. Some participants used `gprof` [72] and others used fine-grained wall clock based timing for assistance. At the end of the time period, twelve of them did not correctly identify parallelism bottlenecks in any of the four applications. One participant, who had 4+ years of experience in parallel programming, identified the bottleneck in one (minSpanForest) application.

Subsequently after the first part, we gave a brief tutorial of TaskProf's what-if analyses on a simple example program. The participants were then asked to identify bottlenecks in the four applications using what-if analyses within an hour. All the participants used the annotation-based approach to what-if analyses where TaskProf would estimate the improvement in parallelism of the annotated regions. Our user study uses a repeated-measures experiment, which can introduce order effects. Students had an opportunity to study the code and attempt to optimize it during the first phase before they were given TaskProf.

Using TaskProf, seven participants found the parallelism bottleneck in all the four applications, one participant found the bottleneck in three of them, four participants found the bottleneck in two of them, and one participant did not find the bottleneck in any application. Among the participants who identified at least one bottleneck for any application, it took them 12 minutes on average per application to identify the bottleneck using TaskProf. The participants indicated that once they became familiar with the tool by identifying a bottleneck in one application, subsequent tasks were repetitive. A common feedback comment from the participants was whether it would be possible to extend TaskProf to identify regions in a program that must be parallelized. This led us to design the adviser feature of TaskProf where TaskProf uses what-if analyses to automatically identify regions that must be parallelized. In summary, our user study suggests that programmers can quickly identify parallelism bottlenecks using TaskProf.

Chapter 7

Related Work

Performance is a primary concern for parallel program developers. However, given the myriad of factors that influence the performance of parallel programs, it is often challenging to extract the best possible performance. Consequently, over the years, numerous performance analysis tools have been proposed to assist developers in diagnosing performance issues in parallel programs. These tools range from generic performance profilers that calculate various metrics from a program execution to tools that identify specific kinds of performance issues in parallel programs. While there has been extensive work on performance analysis tools, there have also been some approaches proposed to automatically mitigate performance issues at execution time by designing intelligent runtime systems. This chapter describes the relevant prior work in identifying and addressing performance issues in parallel programs.

We begin this chapter by presenting an overview of all the widely used performance profilers for parallel programs in Section 7.1. We then present the related work on profilers for identifying serialization and scalability issues, performance estimation tools, critical path analysis tools and tools for identifying secondary effects in Sections 7.2-7.4. Finally, we conclude the chapter by discussing related work on other approaches to address performance issues (Section 7.5) in parallel programs.

7.1 Generic Performance Profilers

Generic performance profilers monitor the execution of a program and highlight code regions where the program performs the most work. Generic profilers typically use sampling or instrumentation to measure various metrics during an execution and attribute the metrics back to the program source code. There are a number of profilers that have been proposed based on generic profiling [2, 3, 10, 23, 48, 56, 68, 72, 91, 150, 151]. These profilers mostly differ on the programming models supported, the technique used for measurement (sampling or instrumentation), or the kinds of performance insights that are provided to the programmer.

One of the earliest generic profilers to be proposed was Gprof [72], which identifies regions where a program spends its time. Gprof uses both sampling and instrumentation to measure the execution time of various functions in a program. It then ranks the functions based on their execution time to highlight the parts of a program that consume the most amount of time. Extensions to Gprof proposed ways to attribute timing measurements to calling contexts and not just syntactic program components [14, 19].

Gprof was a seminal tool that enabled programmers to quickly identify where a program spends time. Following Gprof, many profilers [10, 48, 69, 91, 150, 151, 158, 168] have been proposed that measure various metrics using hardware performance counters and attribute them to the program source code. Profilers like HPCToolkit [10], Intel VTune Amplifier [48], OpenSpeedShop [150], and PGPROF [158] use sampling to measure hardware performance counter events and attribute the samples to the source code. In contrast to sampling based profilers that have low measurement overhead, profilers like Periscope [69], TAU [151], Scalasca [168], and Vampir [91] use heavy-weight instrumentation to obtain precise metrics. However to reduce measurement overhead, these profilers provide a method to annotate parts of the program that need to be profiled.

All of these profilers are useful in understanding the performance of parallel programs. They highlight the most resource consuming parts of the program and can point to hardware related issues such as cache misses, cycles per instruction, latency memory and floating point operations, branch mis-predictions, *etc.* Profilers like Scalasca [168], Paraver [3], and Intel VTune Amplifier [48], also provide a thread-based view of the execution which can be useful to pinpoint code regions where the threads are waiting and thereby identify performance issues due to load imbalance.

However, by measuring resource utilization at various parts of a program, these profilers may not be able to identify the parts of a program that matter for improving the performance. Further the thread-based view provides an insight into a specific execution of the program. It is useful when performance issues like load imbalance manifest in the specific execution. None of these profilers are able to identify scalability bottlenecks that may occur in a different execution with a larger thread count. Hence, these profilers at best can point to the symptom of a performance problem. In contrast, by computing the logical parallelism and performing what-if analyses TaskProf can assess the scalability of programs and identify the parts of a program that matter for increasing the parallelism of the program.

7.2 Identifying Serialization and Scalability Bottlenecks

There is a large body of work on analyzing parallel programs for serialization and scalability performance bottlenecks [31, 54, 57, 58, 61, 66, 84, 89, 130, 145]. They involve techniques that quantify idleness [12, 155, 156], propose visualizations that highlight scalability [58, 61, 135] and measure logical parallelism [92, 146].

Tallent *et al.* [155, 156] proposed techniques to quantify insufficient parallelism and parallelization overhead in the context of both thread-based parallel programs and task parallel programs. These techniques use a novel concept called blame shifting to identify causes of insufficient parallelism and parallelization overhead. The key idea of blame shifting is to attribute the waiting time of idle threads to other parallel threads that are executing concurrently. The insight is that the thread that is executing has excessive serial work which is causing load imbalance and serialization in the computation. Liu *et al.* [109] extended the blame shifting technique in the context of fork-join programs written in OpenMP. Chabbi *et al.* [38] use the blame shifting approach to identify serialization issues in parallel programs executed on heterogenous architectures.

There have been multiple efforts on identifying performance problems due to serialization and load imbalance in large scale distributed parallel programs. Bohme *et al.* [31] propose an analysis in the Scalasca profiler suite to identify wait states in parallel MPI processes which indicate load imbalance or serialization in the computation. While the core idea of wait states is similar to blame shifting, Bohme *et al.* [31] address challenges that arise from wait states due to communication bottlenecks in multi-process, multi-cpu executions. DeRose *et al.* [54] design multiple metrics that can highlight load imbalance issues with a goal of identifying load imbalance with low overhead in large scale applications while executing the program in production.

While all of these approaches focus on serialization in threads and processes, LIME [130] explores serialization and load imbalance stemming from syntactic constructs in a program, like control flow structures. They propose a statistical method to count dynamic decisions at control points and use a divergence in the decision count of two concurrent control points to flag load imbalance issues in a program.

All of the above techniques, including techniques that identify wait states or root causes of serialization using blame shifting are not sufficient to identify all the parts of a program that matter for addressing load imbalance or serialization issues. At best these techniques are the first steps in addressing serialization problems. Even if the programmer optimizes the code causing the serialization, the wait states can shift to a parallel thread and thereby not improve the overall performance of the program. Hence, as observed with our what-if analyses technique we need to iteratively identify all the parts of a program that matter for addressing serialization and load imbalance problems.

Analyzing the scalability of parallel programs has been widely studied. Speedup stacks [61] and bottle graphs [58] propose intuitive visualization techniques to diagnose scalability bottlenecks. Speedup stacks presents a method to visualize the speedup of a program in the form of a stack. The stack is then divided on various scaling delimiters to identify the the specific delimiter that is causing low speedup. Bottle graphs presents a similar visual technique to identify the threads in a program that are causing the scalability bottlenecks. Both speedup stacks and bottle graphs represent orthogonal approaches to visualize scalability problems. While bottle graphs presents a method for identifying threads that cause scalability problems, speedup stacks focuses on identifying the cause for the loss of scalability. Roth *et al.* [145] propose a technique to pinpoint parallel performance factors like delay in software, hardware or work distribution that introduce overhead leading to scaling losses. However, all of these techniques focus on isolating the cause of scalability losses in applications. They do not identify either the parts of the program that are causing the scalability loss or the parts of the program that matter for improving the scalability of the program.

CilkProf [146] and CilkView [77] are profilers that use logical parallelism to identify scalability bottlenecks in task parallel programs. While CilkProf instruments the program at the bytecode level using the compiler, CilkView performs dynamic binary instrumentation. Our on-the-fly algorithm resembles CilkProf's algorithm. Both CilkProf and CilkView require the program to be executed serially, wherein at every spawn statement the newly spawned task and its descendants have to complete before the parent task can continue execution beyond the spawn statement. While such serial depth-first execution is feasible in task parallel models that have serial semantics (*e.g.*, Cilk, Habanero Java), it is not possible in library based task parallel models, like Intel TBB. In contrast, our approach computes the parallelism while executing the program in parallel and is applicable to any task parallel model. Further, CilkProf and CilkView do not identify the parts of a program where the overhead of creating tasks is high with respect to the work performed by the tasks.

In contrast to approaches that have explored increasing parallelism to improve scalability, there are many approaches that have explored techniques to identify sources of excessive task runtime overhead [75, 86, 142, 143]. These approaches on diagnosing sources of runtime overheads have explored profiling techniques to identify optimal task granularity [142, 143] and compiler based techniques to suggest optimal cut-off points for recursive task parallel programs [75, 86, 181]. Improving parallelism or addressing parallelization overheads in isolation is not sufficient to improve the speedup of a program. The programmer has to carefully balance parallelism and the overhead for orchestrating the parallelism to obtain optimal performance.

7.3 Tools to Identify Optimization Opportunities

There have been many tools that have been built for identifying parallelization and optimization opportunities in parallel programs. The goal of such tools is similar to what-if analyses, *i.e.*, to identify parts of the program where the programmer must focus optimization efforts to improve the performance. These tools can be broadly divided into three categories - (1) tools that estimate the improvement in performance on optimization [47, 51, 144], (2) tools that use critical path analysis to identify code that will benefit from optimization [11, 29, 35, 43, 79, 80, 82, 120, 134, 159, 164], and (3) tools that focus on identifying candidates for parallelization [16, 67, 76, 88, 100, 163, 180].

7.3.1 Performance Estimation Tools

Coz [51] is a profiler for multi-threaded programs that estimates the speedup of a program on optimizing a line of code. Coz measures the virtual speedup of a progress point (which indicates the completion of piece of computation) by slowing down all other concurrently executing threads and by building an analytical model to estimate the speedup. It runs periodic experiments at execution time that virtually speed up a single randomly selected program fragment. Virtual speedups produce the same effect as real speedups by uniformly slowing down code executing concurrently with the fragment, causing the fragment to run relatively faster. Based on the virtual speedup experiments on various lines of code, Coz identifies lines of code that speed up the program as the ones that matter for optimization. Coz has a clever idea and our what-if analyses is inspired by it. In our experiments with Coz, we found that it reported pessimistic estimates of speedup with task parallel programs. The improvements suggested by Coz are specific to an execution with a given number of threads. Specifically, it does not indicate whether the program will have scalable speedups when executed on a machine with a different core count.

Similar to Coz, Intel Advisor [47] provides an estimate of the speedup of a program on parallelizing a programmer-annotated piece of code. It also provides information on the scalability of the program by estimating the speedup on machines with larger processor counts. Intel Advisor uses the hotspot analysis to guide the programmer on regions to annotate. However, hotspot analysis does not highlight regions that matter for improving the performance, which makes identifying regions to annotate challenging.

Profilers have explored performance modeling to predict performance on a large number of cores [36, 140] and identify critical paths in parallel programs [11, 148]. The

Roofline model is a visual performance model to identify the upper bound of a kernel's performance on a given machine by measuring metrics such as floating point performance, off-chip and on-chip memory traffic [85, 114, 166]. Yu *et al.* [179] combine performance modeling with data mining to identify general performance problems. However, these approaches do not provide feedback on concrete program regions to focus on.

7.3.2 Critical Path Analysis Tools

There have been numerous prior approaches that have explored identifying optimization opportunities by finding code regions that execute on the critical path. While some approaches propose a post-mortem critical path analysis on traces [35, 82, 119, 120, 171, 172], other approaches propose online analysis [79, 80, 134] to identify code executing on the critical path. IPS [119, 172], and IPS-2 [120, 171] are trace-based approaches that propose techniques to apportion the work on the critical path of an execution to the corresponding functions in the context of both shared and distributed memory parallelism. Slack [82] proposes a metric to estimate the improvement in execution time from optimizing a function that executes on the critical path. Bohme *et al.* [35] extend the approach to find wait states in traces to quantify activity on the critical path in MPMD programs. They identify the critical path as the longest path in the trace without any wait states. Tools that perform online computation of the critical path construct variants of the program activity graph for shared memory [79, 134] and message passing programs [80]. Identifying code regions that execute on the critical path can provide useful information to the programmer on optimization opportunities that can impact a program's performance. However, these approaches do not provide information on how the critical path changes when code regions are optimized.

To identify optimization opportunities when the critical path changes, Hollingsworth and Miller proposed a metric called logical zeroing [81] which quantifies the reduction in the length of the critical path for each function before the critical path changes. Similarly, Alexander *et al.* [11] proposed a technique to identify a finite set of the k-longest execution paths, called near-critical paths, along with the critical path in a program. Since these approaches track only a limited set of paths, they will not identify all the regions that matter for increasing the performance of a program. Further, all of these approaches consider the critical path based on a program's activity in a given execution. They do not consider the critical path based on parallelism constructs expressed in the program.

7.3.3 Tools to Identify Parallelization Candidates

Tools for identifying optimization opportunities have also explored identifying code that can potentially be parallelized [16, 67, 76, 88, 100, 163, 180]. Such tools typically execute the program and track dynamic dependencies between code fragments like, for instance, loop iterations. These tools are different from automatic parallelization approaches [162, 167] which identify parallelization opportunities statically.

Kremlin [67] is a tool for identifying regions of code that can be parallelized in serial programs. It uses a technique similar to critical path analysis to compute the parallelism of code regions in a serial program. Kremlin then ranks the code regions based on their parallelism to identify regions that can potentially be parallelized. Kismet [88] builds on Kremlin to estimate actual speedups for specific machines on which the serial program is executed. DiscoPoP [100] is similar in spirit to Kremlin and can find parallelization opportunities in code regions beyond loops. There have been similar proposals to find parallelization opportunities in sequential Java programs [76] and programs for heterogenous systems [71]. While all of these tools identify code regions that can potentially be parallelized, they do not provide information on regions that will improve the performance of the program on parallelization. Further, these tools cannot identify parallelization opportunities in programs that have already been partly parallelized. In contrast, TaskProf can identify performance issues in parallel programs and identify regions of code that can benefit from parallelization.

7.4 Tools for Identifying Secondary Effects

Techniques to identify secondary effects in parallel program execution have been widely studied. They include tools to detect cache contention [34, 39, 60, 87, 103, 104, 112, 127, 165, 182], identify data locality issues [105–107, 110], detect lock contention problems

[41, 53, 157, 178], and find bottlenecks due to NUMA [95, 108, 115]. Unlike TaskProf, these tools are tailored to detect specific kinds of secondary effects. Further, they either use binary instrumentation or sampling. TaskProf does not explicitly sample while performing measurements (PMU units employ precise event sampling).

Prior profilers have explored using differential analysis to find scalability bottlenecks in parallel programs [45, 83, 111, 116, 126, 149]. eGprof [149] extends Gprof to identify scalability bottlenecks by comparing the profiles generated from multiple executions. Similar to eGprof, HPCToolkit [45] provides the ability to compare performance metrics from different executions to identify scalability bottlenecks. ScaAnalyzer [111] is a tool for identifying memory scalability bottlenecks in parallel programs. It samples memory accesses and attributes the latency of the accesses to the corresponding data object. It then compares executions with different threads and isolates data objects that have high increase in latency as the source of memory scalability bottlenecks. These tools focus on using sampling to identify data objects that cause scalability bottlenecks. Since accesses are sampled, these techniques can miss data objects that do not occur frequently but have high access latencies. In contrast, TaskProf collects precise counts of performance metrics which enables it to identify memory scalability bottlenecks in regions with very few accesses but high latencies.

There are tools that measure work inflation to identify if the program is experiencing secondary effects [8, 131, 145]. But these tools neither pinpoint parts of the program with secondary effects nor its relation to program parallelism. In contrast, our differential analyses can highlight regions that are experiencing any type of secondary effect and probable reasons for them by examining inflation in various metrics of interest.

7.5 Performance-Aware Runtimes

One approach to addressing the performance challenges in task parallel programs is to engineer runtimes that can mitigate the performance problems during program execution. The general idea here is to design intelligent runtimes that are "aware" of how performance issues can occur and steer the execution to avoid such issues. The advantage of such an approach is that the programmer would be relieved from diagnosing performance issues and designing code optimization strategies to address the issues.

There has been significant research on designing intelligent runtimes to mitigate many of the performance issues. Acar *et al.* [6] proposed a technique to automatically control the amount of parallel computation in task parallel programs. The technique proposes a novel concept called oracle-guided scheduling, where the asymptotic cost of a piece of code is used to estimate the execution cycles required to run the code. Based on this estimation, the runtime decides to either execute the piece of code serially or promote the code to other threads and execute it in parallel. Along similar lines, there are runtime techniques that explore limiting the amount of parallel computation to an optimum amount during execution. These techniques either coarsen parallel computation from multiple tasks [4, 7, 9, 59, 86] or explicitly create tasks only on demand when threads become idle [70, 78, 121, 160]. Apart from techniques that deal with controlling the amount of parallel computation, there have been other techniques that focus on mitigating secondary effects. These include techniques to mitigate NUMA effects [25–27, 55], reduce private [5, 62] and shared cache misses [28], increase locality [74, 101, 113, 124, 153, 169], and reduce latency [125].

The disadvantage of such runtime based approaches is that they are best-effort. Many approaches base their decisions solely on certain heuristics or dynamically collected data, like recursive task creation depth and dynamic execution load, which can adversely affect the performance of a program. Further, these approaches have to constantly monitor the execution, which can add significant overhead. In addition, the frequent poling for execution cycles could perturb the execution and introduce unintended secondary effects that affect the performance. Such secondary effects are particularly frustrating for the programmers to diagnose, as they must determine that secondary effects originated due to the runtime. For this reason, most programmers prefer to use profilers over heavyweight abstractions to address performance challenges.

Chapter 8

Conclusion

This dissertation makes a case for using logical parallelism to identify parts of a program that matter in improving the parallelism and the speedup of the program. We first summarize this dissertation's key technical contributions and then present directions for future work.

8.1 Dissertation Summary

Traditional profilers for parallel programs highlight performance issues by identifying the parts of a program where the program's execution spends the most time (or other metrics like execution cycles). However, they fall short of providing actionable information about the parts of the program that matter in improving the performance of the program. In this dissertation, we propose a set of techniques to identify the parts of the program that matter in increasing the parallelism and the speedup of the program.

To identify the parts of the program that matter in improving the parallelism, we first propose a technique to measure the parallelism of a program. The parallelism of a program defines the speedup of the program in the limit. Computing the parallelism requires us to determine the total work performed in the program and the work on the critical path in the program. Since parallelism quantifies the speedup in the limit, the key challenge is in computing the work on the critical path by considering the parts of the program that execute in series based on the series relations expressed in the program. We address the challenge by designing a performance model that records the logical series-parallel relations expressed in the program and the work performed in various fine-grain code fragments in the program. We describe a technique to construct the performance model from an execution of the program by tracking the logical series-parallel relations as task runtime constructs are encountered and by measuring fine-grain work using hardware performance counters. Leveraging the series-parallel relations and fine-grain work in the performance model, we compute the total work performed in the program and the work on the critical path to measure the parallelism of the program.

The performance model for long running applications can be significantly large making it infeasible to store the performance model in memory and perform the parallelism computation over the entire performance model. One of the key contributions of this dissertation is an on-the-fly technique that performs the parallelism computation by maintaining only parts of the performance model corresponding to the active tasks.

In addition to computing the parallelism, we propose a technique that measures the task runtime overhead in the program to determine if the parallel work in the program is too fine-grain to be worth executing in parallel. Computing the logical parallelism and task runtime overhead enables the programmer to assess if a program has adequate parallel work to exhibit scalable speedup on machines with any number of processors. Further, we design a method to attribute the parallelism and the task runtime overhead from the performance model to various static code locations in the program. This enables the programmer to quickly identify the specific parts of the program that have low parallelism and high task runtime overhead.

While our parallelism computation technique highlights parts of the program that have low parallelism, it is still challenging to identify the parts of the program that the programmer should parallelize to improve the parallelism. Parallelizing a part of the program that has low parallelism may not improve the overall parallelism of the program if the part of the program is not performing work on the critical path. Even if the part of the program is performing significant work on the critical path, parallelizing it can shift the critical path to another path and not improve the overall parallelism of the program.

To address these challenges, we propose what-if analyses, a technique that identifies parts of the program that matter in increasing the parallelism of the program. Our insight is that the performance model for computing parallelism contains information about the various code fragments in the program and the series-parallel relations between the code fragments. Using the performance model, we can mimic the effect of parallelizing a code fragment and estimate the parallelism on hypothetically parallelizing the code fragment. The key essence of what-if analyses is that it enables the programmer to determine if a part of the program will benefit from parallelization even before designing a concrete parallelization strategy. We leverage the performance model and our parallelism estimation technique to extend what-if analyses to automatically identify a set of regions that matter in increasing the parallelism of the program. To make what-if analyses practical, we consider the task runtime overhead in the program and perform what-if analyses only if the work after parallelization amortizes the overhead from the runtime to create parallel tasks.

Even if a program has adequate parallelism, it may not exhibit scalable speedup due to the presence of secondary effects of parallel execution that manifest due to contention in hardware or system resources. We propose a technique to identify the parts of the program that matter in increasing the speedup of the program by highlighting code regions that are experiencing secondary effects. It is our insight that a program that is experiencing secondary effects will perform greater amount of work in the parallel execution than the amount of work in an oracle execution that is not experiencing secondary effects. Leveraging this insight, we propose a differential analysis technique that performs a fine-grain comparison of the performance models of the parallel and oracle executions to highlight code regions that are showing an inflation in the work in parallel performance model over the oracle performance model. To highlight code regions that matter in addressing secondary effects, we use the performance model to highlight code regions that are having inflation in the critical path work and are performing a significant fraction of the total work and the critical path work. We also show that by performing differential analysis over a range of hardware performance counter events, we can identify the hardware or system resources which are possibly the source of the secondary effects.

Using the techniques proposed in this dissertation, we have developed TaskProf, a profiler and an adviser for task parallel programs. The insights provided by TaskProf enabled us to identify performance bottlenecks and design optimizations to improve the speedup of a range of applications. The user study we conducted shows that TaskProf enables parallel program developers to quickly identify regions to focus on for improving the performance. Overall, this dissertation demonstrates that techniques to identify parts of the program that matter in improving the performance are useful and necessary for effective performance analysis of parallel programs.

8.2 Directions for Future Work

This dissertation makes a case for analyses that identify the parts of a parallel program that matter in improving the performance of the program. Building on the ideas presented in this dissertation, there are several potential avenues that can be explored to further enhance the performance analysis of parallel programs.

Estimating Maximum Speedup for a Given Number of Processors

In this dissertation, we measure the logical parallelism to determine if a program has sufficient parallel computation to exhibit scalable speedup. For a program to have scalable speedup on a given execution machine, the logical parallelism of the program must significantly exceed the number of processors on the machine. However, it is challenging for the programmer to determine the precise parallelism that is necessary for scalable speedup on a given number of processors. Schardl *et al.* [146] determined through experimentation that the parallelism should be at least 10$\times$ the number of processors on the execution machine. In our experiments, there are programs that have significantly high parallelism but do not exhibit scalable speedup on the execution machine. At the same time, there are programs that have relatively lower parallelism but exhibit scalable speedup.

One possible approach to determine the required parallelism is to estimate the maximum speedup that can be expected from the program on a given number of processors. Estimating the maximum speedup on a given number of processors will provide the programmer the ability to tune the performance on a given machine. If a program does not have scalable speedup, then the programmer can determine if the actual speedup is due to low parallelism in the program or other factors that can affect the speedup (for *e.g.*, secondary effects). Prior tools like Kismet [88], explored estimating the speedup from parallelizing sequential programs using a range of parameters. However, estimating the maximum speedup for a given number of processors can be challenging in the context of task parallel programs, since we would need some means to explore all the ways in which parallel tasks can actually get mapped to threads. A new abstraction that can estimate the maximum speedup would probably be necessary.

Parallelism Profiling for Other Parallel Programming Models

In this dissertation, we focused our attention on measuring parallelism and designing what-if and differential analyses in the context of task parallel programs. For wider adoption by the parallel programming community, an attractive avenue to explore would be to extend these techniques to other parallel programming models, like thread-based shared memory models (pthreads [129]), message-passing models (MPI [44]) and hybrid programming models (CUDA [128], OpenCL [154], *etc.*). The key challenge in extending the techniques proposed in this dissertation to other frameworks is in handling arbitrary dependencies that can exist between code fragments which would otherwise execute in parallel. Most task parallel frameworks constrain the program execution to disallow such arbitrary dependencies although, recent versions of Cilk [46, 52] and OpenMP tasks [133] provide support for such arbitrary dependencies. Hence, designing techniques to measure parallelism and perform what-if and differential analyses in the presence of arbitrary dependencies would be beneficial for all parallel programming models.

Synthesizing Concrete Optimizations

Improving the performance of programs requires the programmer to first identify the parts of the program that have to be optimized. Once the programmer identifies the parts of the program that need to be optimized, the programmer has to design concrete optimizations that will improve the performance. The what-if and differential analyses techniques that we have proposed in this dissertation assist the programmer in identifying program regions that have to be optimized. But, the programmer still has to manually design concrete optimizations. A potential avenue to explore would be to automatically synthesize concrete optimizations for the program regions highlighted by what-if and differential analyses. The key challenge in synthesizing such optimizations is to determine dependencies between code fragments that may execute in parallel in the synthesized code. Further, evaluating the performance implications of various optimizations for the same code fragment will be interesting to explore.

Bibliography

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

[2] Paraformance, 2018. https://www.paraformance.com/.

[3] Paraver, 2018. https://tools.bsc.es/.

[4] 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, pages 214–228, 2019.

[5] U. A. Acar, G. E. Blelloch, and R. D. Blumofe. The data locality of work stealing. In *Proceedings of the Twelfth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA, pages 1–12, 2000.

[6] 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, pages 769–782, 2018.

[7] U. A. Acar, A. Charguéraud, and M. Rainey. Oracle scheduling: Controlling granularity in implicitly parallel languages. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11, pages 499–518, 2011.

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

[9] U. A. ACAR, A. CHARGUÉRAUD, and M. RAINEY. Oracle-guided scheduling for controlling granularity in implicitly parallel languages. *Journal of Functional Programming*, 26:e23, 2016.

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

[11] 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, pages 308–317, 1994.

[12] 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, pages 739–753, 2010.

[13] 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), pages 483–485, 1967.

[14] G. Ammons, T. Ball, and J. R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation*, PLDI '97, pages 85–96, 1997.

[15] N. S. Arora, R. D. Blumofe, and C. G. Plaxton. Thread scheduling for multiprogrammed multiprocessors. In *Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '98, pages 119–129, 1998.

[16] R. Atre, A. Jannesari, and F. Wolf. The basic building blocks of parallel tasks. In *Proceedings of the 2015 International Workshop on Code Optimisation for Multi and Many Cores*, COSMIC '15, pages 3:1–3:11, 2015.

[17] E. Ayguadé, N. Copty, A. Duran, J. Hoeflinger, Y. Lin, F. Massaioli, X. Teruel, P. Unnikrishnan, and G. Zhang. The design of openmp tasks. *IEEE Trans. Parallel Distrib. Syst.*, pages 404–418.

[18] R. Balasubramonian and N. Jouppi. *Multi-Core Cache Hierarchies.* Morgan & Claypool Publishers, 1st edition, 2011.

[19] T. Ball and J. R. Larus. Efficient path profiling. In *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 29, pages 46–57, 1996.

[20] R. Barik, Z. Budimlic, V. Cavè, S. Chatterjee, Y. Guo, D. Peixotto, R. Raman, J. Shirako, S. Tasirlar, Y. Yan, Y. Zhao, and V. Sarkar. The habanero multicore software research project. In *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 735–736.

[21] 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.

[22] M. A. Bender and M. O. Rabin. Scheduling cilk multithreaded parallel programs on processors of different speeds. In *Proceedings of the Twelfth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '00, pages 13–21, 2000.

[23] S. Benedict, V. Petkov, and M. Gerndt. Periscope: An online-based distributed performance analysis tool. In *Tools for High Performance Computing 2009*, pages 1–16, 2010.

[24] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The parsec benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT, pages 72–81, 2008.

[25] S. Blagodurov and A. Fedorova. Towards the contention aware scheduling in hpc cluster environment. *Journal of Physics. Conference Series (Online).*

[26] S. Blagodurov, S. Zhuravlev, and A. Fedorova. Contention-aware scheduling on multicore systems. *ACM Trans. Comput. Syst.*, pages 8:1–8:45.

[27] S. Blagodurov, S. Zhuravlev, A. Fedorova, and A. Kamali. A case for numa-aware contention management on multicore systems. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, pages 557–558, 2010.

[28] G. E. Blelloch, J. T. Fineman, P. B. Gibbons, and H. V. Simhadri. Scheduling irregular parallel computations on hierarchical caches. In *Proceedings of the Twenty-third Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '11, pages 355–366, 2011.

[29] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP, pages 207–216, 1995.

[30] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, pages 720–748, 1999.

[31] D. Bohme, M. Geimer, F. Wolf, and L. Arnold. Identifying the root causes of wait states in large-scale parallel applications. In *2010 39th International Conference on Parallel Processing*, pages 90–100, 2010.

[32] W. J. Bolosky and M. L. Scott. False sharing and its effect on shared memory performance. In *USENIX Systems on USENIX Experiences with Distributed and Multiprocessor Systems - Volume 4*, Sedms'93, pages 3–3.

[33] 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, pages 16:1–16:14, 2018.

[34] 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, pages 1565–1574, 2013.

[35] 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.

[36] 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, pages 45:1–45:12, 2013.

[37] 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.

[38] M. Chabbi, K. Murthy, M. Fagan, and J. Mellor-Crummey. Effective sampling-driven performance tools for gpu-accelerated supercomputers. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 43:1–43:12, 2013.

[39] 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, pages 152–167, 2018.

[40] 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.

[41] 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.

[42] J. Chen and R. M. Clapp. Critical-path candidates: scalable performance modeling for mpi workloads. In *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 1–10, 2015.

[43] J. Chen and R. M. Clapp. Critical-path candidates: scalable performance modeling for mpi workloads. In *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 1–10, 2015.

[44] L. Clarke, I. Glendinning, and R. Hempel. The mpi message passing interface standard. In K. M. Decker and R. M. Rehmann, editors, *Programming Environments for Massively Parallel Distributed Systems*, pages 213–218, 1994.

[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] I. Corporation. Intel cilk plus, 2017. https://www.cilkplus.org.

[47] I. Corporation. Intel advisor, 2019. https://software.intel.com/en-us/advisor.

[48] I. Corporation. Intel vtune amplifier, 2019. https://software.intel.com/en-us/intel-vtune-amplifier-xe.

[49] I. Corporation. Official intel(r) threading building blocks (intel tbb) github repository., 2019. https://github.com/01org/tbb.

[50] Cray Inc. Using Cray Performance Measurement and Analysis Tools, 2015.

[51] 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.

[52] J. S. Danaher, I.-T. A. Lee, and C. E. Leiserson. Programming with exceptions in jcilk. *Sci. Comput. Program.*, pages 147–171, 2006.

[53] 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.

[54] 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.

[55] J. Deters, J. Wu, Y. Xu, and I. A. Lee. A numa-aware provably-efficient task-parallel platform based on the work-first principle. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*, pages 59–70, 2018.

[56] 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.

[57] 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.

[58] 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.

[59] A. Duran, J. Corbalán, and E. Ayguadé. An adaptive cut-off for task parallelism. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, SC '08, pages 36:1–36:11, 2008.

[60] 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, pages 251–265, 2016.

[61] 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.

[62] A. Fedorova, M. I. Seltzer, M. D. Smith, and C. Small. Casc : A cache-aware scheduling algorithm for multithreaded chip multiprocessors. 2005.

[63] 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.

[64] 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.

[65] 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.

[66] T. Gamblin, B. R. de Supinski, M. Schulz, R. Fowler, and D. A. Reed. Scalable load-balance measurement for spmd codes. In *SC '08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, pages 1–12, 2008.

[67] 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.

[68] M. Geimer, F. Wolf, B. J. N. Wylie, E. Ábrahám, D. Becker, and B. Mohr. The scalasca performance toolset architecture. *Concurr. Comput. : Pract. Exper.*, pages 702–719.

[69] M. Gerndt and M. Ott. Automatic performance analysis with periscope. *Concurr. Comput. : Pract. Exper.*

[70] S. C. Goldstein, K. E. Schauser, and D. E. Culler. Lazy threads. *J. Parallel Distrib. Comput.*, pages 5–20, 1996.

[71] R. Govindarajan and J. Anantpur. Runtime dependence computation and execution of loops on heterogeneous systems. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, CGO '13, pages 1–10.

[72] 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.

[73] 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.

[74] Y. Guo. *A Scalable Locality-aware Adaptive Work-stealing Scheduler for Multi-core Task Parallelism.* PhD thesis, Houston, TX, USA, 2010.

[75] S. Gupta, R. Shrivastava, and V. K. Nandivada. Optimizing recursive task parallel programs. In *Proceedings of the International Conference on Supercomputing*, ICS, pages 11:1–11:11, 2017.

[76] C. Hammacher, K. Streit, S. Hack, and A. Zeller. Profiling java programs for parallelism. In *Proceedings of the 2009 ICSE Workshop on Multicore Software Engineering*, IWMSE '09, pages 49–55, 2009.

[77] 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.

[78] 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, pages 55–64, 2009.

[79] J. K. Hollingsworth. An online computation of critical path profiling. In *Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools*, SPDT '96, pages 11–20, 1996.

[80] J. K. Hollingsworth. Critical path profiling of message passing and shared-memory programs. *IEEE Transactions on Parallel and Distributed Systems*, pages 1029–1040, 1998.

[81] J. K. Hollingsworth and B. P. Miller. Parallel program performance metrics: a comparison and validation. In *Supercomputing '92:Proceedings of the 1992 ACM/IEEE Conference on Supercomputing*, pages 4–13, 1992.

[82] J. K. Hollingsworth and B. P. Miller. Slack: A new performance metric for parallel programs. Technical report, University of Wisconsin-Madison, 1994.

[83] 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.

[84] 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.

[85] A. Ilic, F. Pratas, and L. Sousa. Cache-aware roofline model: Upgrading the loft. *IEEE Computer Architecture Letters*, pages 21–24, 2014.

[86] 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, pages 139–150, 2016.

[87] 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, pages 30:1–30:9, 2013.

[88] 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.

[89] 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.

[90] 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.

[91] 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.

[92] M. Kumar. Measuring parallelism in computation-intensive scientific/engineering applications. *IEEE Trans. Comput.*, pages 1088–1098.

[93] R. Kumar, D. M. Tullsen, P. Ranganathan, N. P. Jouppi, and K. I. Farkas. Single-isa heterogeneous multi-core architectures for multithreaded workload performance. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, ISCA, 2004.

[94] L. L. N. Labs. Livermore unstructured lagrangian explicit shock hydrodynamics (lulesh), 2018. https://computation.llnl.gov/projects/co-design/lulesh.

[95] 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, pages 53–64, 2012.

[96] D. Lea. A java fork/join framework. In *Proceedings of the ACM 2000 Conference on Java Grande*, JAVA, pages 36–43, 2000.

[97] D. Lea. *Concurrent Programming in Java(TM): Design Principles and Patterns (3rd Edition) (Java (Addison-Wesley))*. Addison-Wesley Professional, 2006.

[98] D. Leijen, W. Schulte, and S. Burckhardt. The design of a task parallel library. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA, pages 227–242, 2009.

[99] C. E. Leiserson. The cilk++ concurrency platform. In *Proceedings of the 46th Annual Design Automation Conference*, DAC, pages 522–527, 2009.

[100] Z. Li, A. Jannesari, and F. Wolf. Discovery of potential parallelism in sequential programs. In *Proceedings of the 2013 42Nd International Conference on Parallel Processing*, ICPP '13, pages 1004–1013, 2013.

[101] J. Lifflander, S. Krishnamoorthy, and L. V. Kale. Steal tree: Low-overhead tracing of work stealing schedulers. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 507–518, 2013.

[102] A. Limited. big.little technology: The future of mobile. 2013.

[103] 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, pages 3–18, 2011.

[104] 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, pages 3–14, 2014.

[105] X. Liu and J. Mellor-Crummey. Pinpointing data locality problems using data-centric analysis. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO, pages 171–180, 2011.

[106] X. Liu and J. Mellor-Crummey. A data-centric profiler for parallel programs. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC, pages 28:1–28:12, 2013.

[107] 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.

[108] 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.

[109] 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.

[110] 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.

[111] 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.

[112] 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.

[113] J. Maglalang, S. Krishnamoorthy, and K. Agrawal. Locality-aware dynamic task graph scheduling. In *2017 46th International Conference on Parallel Processing (ICPP)*, pages 70–80, 2017.

[114] D. Marques, H. Duarte, A. Ilic, L. Sousa, R. Belenov, P. Thierry, and Z. A. Matveev. Performance analysis with cache-aware roofline model in intel advisor. In *2017 International Conference on High Performance Computing Simulation (HPCS)*, pages 898–907, 2017.

[115] 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.

[116] P. E. McKenney. Differential profiling. *Software - Practice & Experience*, pages 219–234, 1999.

[117] 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.

[118] Microsoft Windows Dev Center. Processes and threads, 2018. https://docs.microsoft.com/en-us/windows/win32/procthread/processes-and-threads.

[119] B. Miller and C.-Q. Yang. Ips: An interactive and automatic performance measurement tool for parallel and distributed programs. pages 482–489, 1987.

[120] 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*, pages 206–217, 1990.

[121] 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*, pages 264–280, 1991.

[122] G. E. Moore. Cramming more components onto integrated circuits. *Electronics*, 1965.

[123] G. E. Moore. Progress in digital integrated electronics. *International Electron Devices Meeting*, 21:11–13, 1975.

[124] A. Muddukrishna, P. A. Jonsson, and M. Brorsson. Locality-aware task scheduling and data distribution for openmp programs on numa systems and manycore processors. *Sci. Program.*, pages 5:5–5:5.

[125] S. K. Muller and U. A. Acar. Latency-hiding work stealing: Scheduling interacting parallel computations with work stealing. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '16, pages 71–82, 2016.

[126] 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, pages 353–366, 2012.

[127] 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, pages 141–154, 2013.

[128] NVIDIA Corporation. NVIDIA CUDA C programming guide, 2019. Version 10.1.

[129] I. of Electrical and E. Engineers. Ieee standard for information technology–portable operating system interface (posix(r)) base specifications. 2017.

[130] 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.

[131] 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, pages 65:1–65:12, 2012.

[132] OpenMP Architecture Review Board. Openmp 4.5 complete specification, Nov. 2015.

[133] OpenMP Architecture Review Board. Openmp 5.0 preview 2 specification, Nov. 2017.

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

[135] 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, 2017.

[136] R. Raman. *Dynamic Data Race Detection for Structured Parallelism.* PhD thesis, Rice University, 2012.

[137] 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.

[138] 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.

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

[140] 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.

[141] H. Riesel. *Prime Numbers and Computer Methods for Factorization.* Progress in mathematics. Springer, 1994.

[142] 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, pages 27–37, 2018.

[143] 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.

[144] C. Rosas, J. Giménez, and J. Labarta. Scalability prediction for fundamental performance factors. *Supercomput. Front. Innov.: Int. J.*, pages 4–19, 2014.

[145] 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, pages 59–68, 2012.

[146] 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.

[147] F. Schmitt, R. Dietrich, and G. Juckeland. Scalable critical-path analysis and optimization guidance for hybrid mpi-cuda applications. *The International Journal of High Performance Computing Applications*, 31(6):485–498, 2017.

[148] M. Schulz. Extracting critical path graphs from mpi applications. In *2005 IEEE International Conference on Cluster Computing*, pages 1–10, 2005.

[149] 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.

[150] 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.

[151] S. S. Shende and A. D. Malony. The tau parallel performance system. *International Journal of High Performance Computing Applications*, pages 287–311, 2006.

[152] 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.

[153] S. Srikanthan, S. Dwarkadas, and K. Shen. Data sharing or resource contention: Toward performance transparency on multicore systems. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '15, pages 529–540.

[154] J. E. Stone, D. Gohara, and G. Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *Computing in Science Engineering*, pages 66–73, 2010.

[155] 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.

[156] 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.

[157] 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.

[158] The Portland Group. Pgi profiler user's guide - pgi compilers, 2017.

[159] D. M. Tullsen and B. Calder. Computing along the critical path. Technical report, University of California, San Diego, 1998.

[160] A. Tzannes, G. C. Caragea, U. Vishkin, and R. Barua. Lazy scheduling: A runtime adaptive scheduler for declarative parallelism. *ACM Trans. Program. Lang. Syst.*, pages 10:1–10:51.

[161] R. Utterback, K. Agrawal, J. T. Fineman, and I.-T. A. Lee. Provably good and practically efficient parallel race detection for fork-join programs. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA, 2016.

[162] H. Vandierendonck, S. Rul, and K. De Bosschere. The paralax infrastructure: Automatic parallelization with a helping hand. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, pages 389–400, 2010.

[163] C. von Praun, R. Bordawekar, and C. Cascaval. Modeling optimistic concurrency using quantitative dependence analysis. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '08, pages 185–196, 2008.

[164] L. Wang, Y. Deng, R. Gong, W. Shi, Z. Zhao, and Q. Dou. A parallel algorithm for instruction dependence graph analysis based on multithreading. In *2018 IEEE Intl Conf on Parallel Distributed Processing with Applications, Ubiquitous Computing Communications, Big Data Cloud Computing, Social Computing Networking, Sustainable Computing Communications (ISPA/IUCC/BDCloud/SocialCom/SustainCom)*, pages 716–721, 2018.

[165] 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, pages 332–347, 2018.

[166] S. Williams, A. Waterman, and D. Patterson. Roofline: An insightful visual performance model for multicore architectures. *Communications of the ACM*, pages 65–76, 2009.

[167] R. P. Wilson, R. S. French, C. S. Wilson, S. P. Amarasinghe, J. M. Anderson, S. W. K. Tjiang, S.-W. Liao, C.-W. Tseng, M. W. Hall, M. S. Lam, and J. L. Hennessy. Suif: An infrastructure for research on parallelizing and optimizing compilers. *SIGPLAN Notices*, pages 31–37.

[168] 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.

[169] Y. Yan, J. Zhao, Y. Guo, and V. Sarkar. Hierarchical place trees: A portable abstraction for task parallelism and data movement. In *Proceedings of the 22Nd International Conference on Languages and Compilers for Parallel Computing*, LCPC'09, pages 172–187, 2010.

[170] 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.

[171] C. . Yang and B. P. Miller. Performance measurement for parallel and distributed programs: a structured and automatic approach. *IEEE Transactions on Software Engineering*, pages 1615–1629, 1989.

[172] C.-Q. Yang. *A Structured and Automatic Approach to the Performance Measurement of Parallel and Distributed Programs.* PhD thesis, 1987. AAI8723365.

[173] A. Yoga and S. Nagarakatte. Atomicity violation checker for task parallel programs. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, CGO, pages 239–249, 2016.

[174] 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, pages 15–26, 2017.

[175] 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, pages 485–501, 2019.

[176] A. Yoga and S. Nagarakatte. Taskprof2, 2019. https://github.com/rutgers-apl/TaskProf2.git.

[177] 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.

[178] 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.

[179] X. Yu, S. Han, D. Zhang, and T. Xie. Comprehending performance from real-world execution traces: A device-driver case. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, pages 193–206, 2014.

[180] X. Zhang, S. Jagannathan, and A. Navabi. Alchemist: A transparent dependence distance profiling infrastructure. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 47–58, 2009.

[181] J. Zhao, J. Shirako, V. K. Nandivada, and V. Sarkar. Reducing task creation and termination overhead in explicitly parallel programs. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT, pages 169–180, 2010.

[182] 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, pages 27–38, 2011.