RESEARCH STATEMENT

Adarsh Yoga (adarsh.yoga@cs.rutgers.edu)

Performance is a primary concern for software developers. Developers rely on profilers to diagnose performance problems. Traditional profilers usually attribute resource utilization (*e.g.*, time) to program source code and help identify parts of the program where resources are most utilized. While these profilers are useful in highlighting "hot" code, they fall short of providing actionable feedback in the context of parallel programs. The focus of my research is to design performance profilers for software written for today's multi-core systems. In my dissertation, I have proposed a profiling technique that enables programmers to easily identify various performance pathologies in a class of parallel programs, called *structured task-based parallel programs*. In this essay, I will detail my journey so far as a researcher and put forth my future research plans.

1 Dissertation Research - Profiling Task Parallel Programs

My interest in parallel programming led me to explore task based parallel programs for my dissertation research. Task parallel programs provide the programmer the ability to express the parallelism in the program without having to explicitly manage the underlying details of how the parallelism gets distributed to hardware threads. While convinced about the benefits of task parallel programs over general multi-threaded programs, I found that task parallel programs suffered from the same concurrency correctness issues as multi-threaded programs.

With the benefit of prior work, I kick-started my research career by proposing techniques to identify concurrency correctness issues like data races [7] and atomicity violations [4] in task parallel programs. While techniques to detect concurrency bugs in parallel programs are aplenty, the key contribution of our work is the guarantee that a concurrency bug will be detected if one exists in the program for the input, irrespective of whether the bug actually manifests in the specific execution. This strong guarantee was primarily enabled by an abstraction [2] of the task parallel program execution that can be used to check if two accesses may happen in parallel, which forms the basis for any concurrency bug detection technique.

While exploring techniques to identify concurrency bugs, I had a flash of insight. Similar to concurrency bugs, why not leverage the task parallel program abstraction to find performance issues even if the performance issue may not manifest in a given execution.

Performance pathologies. I first started experimenting with numerous task parallel programs to understand what sort of performance pathologies exist. I found that the performance of a task parallel program can be influenced by several factors. On one hand, the program may not be performing sufficient work in parallel causing load imbalance and sub-optimal performance. On the other hand, the program may have excessive parallelism resulting in the runtime overhead overwhelming the execution. Finally, even if a program has optimal parallelism, it can have low performance due to contention for shared resources in hardware (*e.g.*, caches) and software(*e.g.*, locks).

I argue that simply measuring where the program spends time is not sufficient to identify performance problems in task parallel programs. For instance, a source code region may be performing a significant fraction of the total work in the program. But, if it is not executing on the critical path of the program, optimizing the region will not improve the performance of the program. Hence unlike in sequential programs, in task parallel programs there is no direct correlation between code that executes frequently and code that must be optimized to improve performance. Moreover, many performance bottlenecks will manifest only when executed at scale and may not really show up during small-scale testing. So, I tackle the following question in my dissertation research: *is it possible to predict from an execution on a specific machine, the program regions that must be optimized to improve performance when executed on any machine?*

My dissertation research makes the following contributions.

Parallelism profiling. The performance of a task parallel program depends on the amount of parallel work in the program. Hence, we characterize the performance of the program in terms of the parallelism in the program. Parallelism as a metric of interest, is constrained by the critical path of the program and places an upper bound on the speedup of program. It is independent of the number of threads on the execution machine and thereby a property of the program. A program that does not have sufficient parallelism for a given number of threads will not exhibit good performance. Hence, knowing the parallelism of a poorly performing program is useful in understanding why the performance is low.

I created TaskProf, a profiler that computes the parallelism of a task parallel program and attributes the parallelism to the program source code [5]. The primary task in computing the parallelism is to identify the critical path of the program from the program execution. This is particularly challenging in the context of task parallel programs since multiple parallel tasks can get multiplexed on to the same execution thread, and thereby the critical path of the program will not be the same as the critical path of the execution. I tackle this challenge by leveraging the task parallel program abstraction as a performance model that records actual series and parallel relationships between tasks, based on program semantics rather than actual execution. I used TaskProf to profile many task parallel applications for which we were not able to understand the cause for the low performance. TaskProf was highly effective in helping me understand the cause for the low performance and identify what parts of the program are contributing to the low performance.

Identifying program regions to optimize using parallelism. Although TaskProf was useful in diagnosing the cause of low parallelism, it was not very useful in identifying what parts of the program to optimize to improve parallelism. In one instance, I spent hours trying to parallelize a code region highlighted by TaskProf as having low parallelism. But, the optimization did not improve the parallelism or the performance of the program since code region did not perform significant work on the critical path of the program. That was when I had an *aha* moment: what if we could check if optimizing a code region would improve the performance of the program before even designing a concrete optimization.

I designed an analysis in TaskProf to check if hypothetically optimizing a region of code will improve the performance of the program [5]. We call it *what-if analysis*. The key insight is that by characterizing the performance in terms of parallelism, we can perform *what-if analysis* by checking if the parallelism of the program changes when a region of code is designated to be optimized. Again, by leveraging the task parallel program abstraction as a performance model, I mimic the effect of parallelizing the designated code region and re-compute the parallelism. If the parallelism of the program improves, then it means that performance of the program will improve after optimizing designated region.

Using the insight, I took *what-if analysis* a step further. I designed an analysis to automatically identify all the code regions that must be optimized to improve the parallelism of the program to any desired level[6]. Using this analysis, I am able to predict from a single execution, the program regions that must be optimized to improve the parallelism to a level that is needed to achieve scalable performance on any machine. The combination of *what-if analysis* and identifying regions automatically, enabled me to quickly identify what code regions to optimize in numerous task parallel applications and eventually improve their performance.

Identifying runtime overhead and contention for shared resources. Beyond sub-optimal parallelism, a task parallel program can experience other types of performance pathologies like excessive runtime overhead and contention due to shared resources. I show that our performance model can be used to precisely identify the source of such performance pathologies. I have designed an analysis to pinpoint locations where the cost of task creation is significant compared to the computation performed by the task, thereby the execution does not warrant task creation. I have proposed a novel differential analysis technique to identify contention due to shared resources by comparing the performance model of a parallel execution to the serial execution. The intuition here is that since the performance model records work performed by different parts of the program at a fine-granularity, any contention will show up as work inflation in the performance model of the parallel execution over the performance model of the serial execution. Using the differential analysis technique I was able to identify bottlenecks in large-scale scientific applications which typically have sufficient parallelism, but often have sub-optimal performance due to contention in shared resources.

The key enabler of my dissertation research has been the performance model that consists of the abstraction of a task parallel program. While the performance model that I have used is restricted to a class of parallel programs that have constrained parallelism (*e.g.*, task based parallel programs), it always begs the question: are there parallel programs beyond task parallel programs where such a performance model can be constructed? I collaborated with Boushehrinejadmoradi [1] to design a parallelism profiler for OpenMP programs, where we showed that a performance model can be constructed for parallel programs that have slightly less restricted parallelism than the constrained parallelism in task based parallel programs.

2 Future Research Directions

The overarching goal of my research has been to design scalable techniques with strong guarantees to help developers write correct, performant programs. My dissertation research is a first step in showing such techniques are indeed useful in improving program performance for a class of parallel programs. Going forward, I plan to take my research in two directions: (1) design techniques to tune software to make best use of emerging computing systems, and (2) influence the design of hardware and programming interfaces to better match the needs of software. In addition, I plan to extend the ideas from my dissertation research to perform analysis of large scale high performance computing applications.

Tuning software for heterogenous systems. As the demand for computation resources keeps increasing, the trend in computing systems today is towards custom hardware specializations to provide domain specific acceleration. Designing software for such systems having numerous customized chips is bound to get harder. From a software design perspective, I foresee three key challenges. (1) Software systems have to be designed such that the compute units are kept busy at most times during execution. Deciding how to place computation on to the compute units to obtain optimal performance is a hard problem. (2) The programmer will have to reason about interleavings from numerous parallel compute units. Hence ensuring correctness can be challenging. (3) With numerous computation threads executing on different domains running concurrently, contention at various levels can cause performance bottlenecks. To tackle these challenges the programmers will require intimate knowledge of the custom hardware and programming interfaces used to interact with the hardware. I want to design novel analysis techniques to assist programmers tune software for new-age computer systems.

Performance monitoring in heterogenous systems. Future systems could potentially have hundreds of heterogenous compute units. With performance becoming a first order constraint, performance monitoring systems will be important in assisting programmers diagnose performance bottlenecks. So the key question is how should performance monitoring units be enhanced to efficiently monitor such divergent systems. Below are several questions I plan to investigate.

- *ISA extensions.* The performance monitoring unit in a custom compute unit should capture unique characteristics of that compute unit which will help diagnose bottlenecks of that unit. What are the hardware and Instruction Set Architecture (ISA) extensions necessary have capture and communicate the metrics?

- *Programming interfaces.* While each custom unit will have specific metrics which quantify its performance, we will likely need a generic interface to accesses the performance data. How should applications and analysis tools communicate with the performance monitoring units of different compute units?

- *Tracking data communication.* Future applications that runs on these systems will transfer data across different compute units. In such cases, a bottleneck at a particular compute unit can cause application slowdown. We need a lightweight hardware mechanism to track the flow the data across different compute unit.

As an intern at Hewlett Packard Labs, I explored hardware and network protocol extensions to monitor the flow of packets through a high performance computing network [3]. I will draw on this experience to design lightweight performance monitoring systems for heterogenous machines.

Analysis techniques for HPC systems. I am excited about the opportunities my dissertation research has opened up in profiling parallel programs, especially HPC programs. Performance profilers for HPC programs need to perform analyses on the data to provide actionable feedback in the context of parallel programs. My dissertation research showed the value of such an approach in the context of structured task parallel programs. Unlike task parallel programs, HPC programs have arbitrary dependencies between program fragments. The key challenge is in designing a performance model that can reason about arbitrary dependencies in a scalable manner. While there are numerous tools that do a very good job of profiling HPC programs, I will add value by combining program analysis with profiling.

Moving forward, computer systems will become highly specialized and software applications will need to be carefully engineered to obtain the best performance. I look forward to contributing to research that helps programmers get the best out of systems.

References

[1] Nader Boushehrinejadmoradi, Adarsh Yoga, and Santosh Nagarakatte. A parallelism profiler with what-if analyses for openmp programs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, SC '18, pages 16:1–16:14, 2018.

[2] Raghavan Raman, Jisheng Zhao, Vivek Sarkar, Martin Vechev, and Eran 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.

[3] Adarsh Yoga and Milind Chabbi. Path-synchronous performance monitoring in hpc interconnection networks with source-code attribution. In *High Performance Computing Systems. Performance Modeling, Benchmarking, and Simulation*, pages 221–235, 2018.

[4] Adarsh Yoga and Santosh 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.

[5] Adarsh Yoga and Santosh Nagarakatte. A fast causal profiler for task parallel programs. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, pages 15–26, 2017.

[6] Adarsh Yoga and Santosh Nagarakatte. Parallelism-centric what-if and differential analyses. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '19, 2019.

[7] Adarsh Yoga, Santosh Nagarakatte, and Aarti 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.