論文の内容の要旨
Abstract

論文題目　Analyzing Performance Differences of Task Parallel Runtime Systems based on Scheduling Delays (スケジューリング遅延に基づいたタスク並列ランタイムシステムの性能差の解析)

氏名　フィン ゴクアン (Huynh Ngoc An)

The number of processor cores integrated in a computer keeps increasing with *multicore* and *many-core* architectures. Programming these architectures becomes more difficult using traditional parallel programming models such as MPI and POSIX Threads where programmers need to manually manage each thread for each processor core. Modern *task parallel* programming models have developed to deploy sophisticated *runtime systems* which can handle those low-level thread-managing details so that the programmers can focus on higher-level aspects of the software development (e.g., algorithms, creativities).

In task parallel programming models, programmers are presented with a unified interface of *tasks*: a programmer does not need to be aware of threads, but just needs to extract logical parallelism in the program by creating tasks as easily as calling a function. These tasks will be mapped to available threads (processor cores) at runtime so as to turn these logical parallelism to actual parallelism exploiting hardware parallel resources as much as possible. In order to provide that unified task interface, a runtime system includes a *task scheduler* as an essential component which is responsible for scheduling a large number of logical tasks created in the program onto available threads dynamically at runtime. The scheduler usually launches a certain number of worker threads (or *workers* for short) according to the available processor cores in the underlying hardware system and assigns tasks to them promptly as the tasks get created.

As the runtime system handles most of the important mechanisms in a parallel execution automatically, the performance of a task parallel program depends greatly on the runtime system that runs it. The same program run by different runtime systems may expose very *different performances*. Clarifying causes behind these performance differences between systems is important for the development of the task parallel paradigm.

We have developed an analysis to quantify performance differences of task parallel runtime systems based on their *scheduling delays*. The analysis breaks down the cumulative execution time of a parallel execution of a task parallel program into four components. Cumulative execution time of a parallel execution is the multiplication of the execution's elapsed time and the number of workers participating in the execution (cumul. exe. time = elasped_time × workers). This cumulative execution time is divided into *work*, *delay*, *no-work-sched*, and *no-work-app* components. Work is the time the workers spend on executing the program code. Delay is the time a worker was not executing the program code, despite there was at least one ready task existing in the system to feed that worker at the time. Delays happen because the system fails to do its scheduling job fast enough in matching together the free worker and the ready task. On the other hand, no-work (sum of no-work-sched and no-work-app) is also the time a worker was not executing the program code, but there was *no* ready task existing in the system to feed

that worker at the time. No-work seems to be legitimate because there was no work for the free worker to do; the situation can't be blamed on the runtime system, but is caused by the application not creating enough parallelism. However, no-work is not totally caused by the application's lack of work; only a part of no-work is caused by the application, and the other part is *actually* caused by the runtime system (scheduler). To see this, suppose a program that has only one parent task that spawns all other child tasks. A delay in advancing that parent task will not only cause a delay on the worker executing the parent task, but also cause longer no-work intervals on the other workers trying to steal tasks. Therefore, it is necessary to divide no-work into two sub-components of no-work-sched, which is caused by the scheduler, and no-work-app, which is caused by the application.

We have done this no-work sub-division by adopting a heuristic that uses the notion of *ready path*. Ready path is one of the critical paths on the task graph of the task parallel program; along the ready path there is always a task running or ready. The ready path length can be classified into three parts of *work* (during which a task was running), *scheduler delay* (during which a task was ready and there was at least one free worker), and *busy delay* (during which a task was ready but there was no free worker). As all workers were busy during busy delay intervals, the no-work component of the cumulative execution time happens during either work or scheduler delay intervals of the ready path; and we consider the part of no-work happening during work intervals as no-work-app and the other part of no-work happening during scheduler delay intervals as no-work-sched.

These four components of the cumulative execution time breakdown play the role of general metrics that do not only help give users an *overall impression* about the performance of the execution, but also effectively *contrast* the differences in performance between different executions, and *signal* possible causes of performance drawbacks. A large work in a parallel execution (compared with the work of the serial execution of the same program) indicates the inflation in work (*work stretch*) due to more cache misses, longer remote memory accesses, more thread contention, etc. which are routinely incurred in a parallel execution on a highly parallel architecture. Large delay and no-work-sched suggest more inefficiencies happening in the runtime scheduler; and large no-work-app suggests there is a lack of parallelism in the application.

In order to implement this scheduling delay-based analysis, we need to capture a trace, which records every start and stop time of any task, and dependencies between those tasks, from an execution of a task parallel program. We model the trace as a directed acyclic graph (*computation DAG*) with execution intervals of tasks as nodes, and dependencies between tasks as edges. The tracing part of our tool (DAG Recorder) instruments time-measuring code around any task parallel primitive (i.e., task-creating primitives, and tasks-waiting primitives), constructs the DAG in memory as the execution progresses, and flattens the DAG out to file when the execution ends. Capturing the whole DAG of a fine-grained task parallel program will result in a huge trace and large overheads. In order to mitigate this problem, we make the tracer collapse "uninteresting" parts of the DAG into single nodes, while maintaining aggregate performance information, on the fly during the execution. "Uninteresting" parts refer to parts (sub-graphs) of the DAG that were executed entirely by only one worker. By replacing single-worker-executed sub-graphs with single nodes, the size of the trace now scales with work-stealing operations across workers rather than with the number of task creations.

The trace is then examined by the post-mortem analysis part of our tool (DAGViz) to calculate the breakdown of work, delay, no-work-sched, and no-work-app. Not only this breakdown, DAGViz is also a useful and practical visualization tool which visualizes the trace with many kinds of visualizations to provide users with many persepctives to inspect the performance. DAGViz's visualizations allow users to interactively explore the trace, and arbitrarily zoom in any spot in the trace to get to understand the performance of the execution. DAGViz provides mainly four kinds of visualizations: (1) basic DAG visualization with nodes having basic shapes (triangles, rectangles, rounds) based on their kinds (create, wait, collective, end); (2) timing-based DAG visualizations with nodes having lengths based on the duration of their execution intervals; (3) timelines visualizations in which nodes are rearranged into rows of workers; and (4) parallelism profiles which depict the running parallelism (number of running tasks) and ready parallelism (number of ready tasks) of the execution over time.

Because the DAG has a *hierarchical structure* (there is only a single root node at the highest level

which gets expanded gradually into the full DAG), the visualizations are also implemented in a hierarchical manner, allowing users to expand and collapse the DAG freely either level-by-level or node-by-node to view at a high-level angle or a detailed angle. The expansions and collapses of DAG are performed with *animations* of gradually expanding/collapsing nodes so that the perception of the DAG in the user's mind holds seamlessly during transitions between visualizations of different levels of details.

Our tool has been implemented with five task parallel programming models (Cilk Plus, MassiveThreads, OpenMP, Qthreads, and TBB). A set of generic task parallel primitives (task creation, task synchronization) that wrap respective primitives in specific models are used to simplify the writing effort. The benchmark is written only one time using the generic primitives, then gets compiled to multiple executables based on different models by switching compilation options that dictate the translation of the generic primitives to spefic ones of the specified model. During this translation, time-measuring code is also automatically instrumented around a task parallel primitive in order to capture time points when a worker transits from program code to scheduler code and vice versa.

We have evaluated our proposed analysis and tool with 10 applications in BOTS benchmark suite and 11 applications in TP-PARSEC benchmark suite. The BOTS benchmarks were originally written in OpenMP's task parallel model, so we have just replaced these task parallel primitives with our generic ones so that our DAG tracer works and allows us to evaluate BOTS with five supported runtime systems. TP-PARSEC (task parallel PARSEC) benchmark suite is a new benchmark suite made by us based on the PARSEC benchmark suite. The original PARSEC benchmark suite was written in three traditional parallel programming models of POSIX Threads, OpenMP's parallel for loop, and TBB's parallel for loop. We have re-implemented PARSEC benchmarks with task parallelism based on our generic primitives and published it as a new benchmark suite - TP-PARSEC.

By applying our analysis and tool to various runtime systems and benchmarks, we have discovered many useful and interesting inefficiencies in the implementations of some runtime systems.