Critical Path Analysis for Characterizing Parallel Runtime Systems*

An Huynh[1,a)] Kenjiro Taura[1,b)]

Abstract: Parallel programming models are increasingly relying on runtime systems to map logical parallelism onto available physical cores at runtime. In order to characterize runtime systems in doing this dynamic scheduling job, we analyze how they progress computation along the critical path of serially dependent tasks along which there is always a task ready or running. Hence, the entire execution time can be classified into three parts: work, during which a task was running; busy delay, during which no task on the path was running but all cores were busy; and scheduler delay, during which no task on the path was running and there was at least one available (idle) core. A large scheduler delay is likely to be an artifact of an implementation of the runtime system. Furthermore, we classify scheduler delays by the types of dependencies (e.g., end of a task, creation of a task, continuation of parent tasks) which can highlight differences in the design and implementation of the runtime systems.

Keywords: task parallel; critical path; runtime system; computation DAG; profiler

1. Introduction

Computer systems have been becoming increasingly parallel with more nodes, more cores, and more threads. The memory hierarchy has also been developed deeper and more complex. They all have made parallel programming techniques difficult and hard to use. In order to achieve both targets of programmability and good performance, system developers need to first present easy-to-use, generic APIs to users, and on the other hand, provide good supporting tools such as compilers and runtime systems that can extract the best performance out of the program algorithms and target hardware systems while avoiding involving users into intricate mechanisms in parallel processing as much as possible.

As the underlying hardware systems are developing large and diversely, many parallel programming models have made use of a more capable runtime system which can do the scheduling of logically parallel tasks dynamically at runtime. This approach is promising because it is more likely to figure out the best scheduling strategy for the target system and environment configuration at runtime. Because the hardware settings taking part in a program execution is not known until runtime, dynamic scheduling is considered more scalable and portable compared with a static one determined at compile time. These more capable runtime systems generally provide a task parallel scheduler that schedules and delivers logical tasks to available processing elements automatically. Only one common interface - task parallel primitives are exposed to the users, and they have only one mission that is to create many enough tasks which can be executed in parallel. All low-level mechanisms are handled by the runtime systems (and hopefully compilers, etc.). Other easier-to-use, commonly seen interfaces like parallel for loop, parallel for loop with reduce can be built upon task parallelism.

It can be said that runtime systems are becoming more responsible, and account more for the performance loss. It matters much to understand them and improve them. However, they have one drawback which is that every mechanism at runtime is invisible from user perspective, making it difficult for them to understand performance and reason when it is bad. There are various choices in doing the scheduling of parallel tasks, and each system has made its own choices in its design and implementation. Clearly clarifying subtle performance differences and trade-offs between them is important for improving task parallel models. As an effort towards solving their drawbacks and improving task parallel runtimes, we have been working on the development of a performance tool that is applicable to all task parallel programming models that can pinpoint performance differences between them.

There are many existing task parallel runtime systems such as widely-used OpenMP [1], Cilk Plus [2], Intel Threading Building Blocks (Intel TBB) [3], and research-based Qthreads [4], and MassiveThreads [5] [6]. Beside higher-level interfaces to parallel processing, they all expose a basic task parallel interface which essentially consists of two primitives,

[1] University of Tokyo, Japan
[a)] huynh@eidos.ic.i.u-tokyo.ac.jp
[b)] tau@eidos.ic.i.u-tokyo.ac.jp
[*1] This manuscript appears in the unrefereed Japanese Summer United Workshops on Parallel, Distributed and Cooperative Processing (SWoPP) and is not a published paper.

one to spawn a task, and another to synchronize tasks. We have built a basic performance model according to this basic interface, it is called computation DAG, and it is applicable to all systems that support task-spawning and task-joining operations. The computation DAG is unique for a program, and is consistent among different executions by different task parallel systems. Because of that computation DAG enables us to compare different systems' performances.

We have developed a critical path analysis method based on the computation DAG model that helps us characterize the runtime systems. A critical path is a serial sequence of tasks from the start of the execution to the end during which there is always a task running or ready. That critical path's length can be decomposed into three components of work, busy delay and scheduler delay. The scheduler delay is further decomposed into four sub-components of end, create, create continue (create cont.) and wait continue (wait cont.) based on four kinds of edges in the computation DAG. Moreover, the total execution time of a program (elapsed time × #cores) can be broken down into three categories of work, delay (scheduler delay), and nowork (core idleness). These decomposition analyses have exposed differences in the executions of different systems, helping us understand more their behaviors and get to know where to put effort on to improve further performance of the benchmarks and task schedulers.

2. Task Parallel Runtime Systems

Cilk Plus has a pure work-stealing task scheduler [7]. It uses two new keywords for creating a task (*cilk_spawn*) and joining a task (*cilk_sync*). Qthreads and MassiveThreads are both libraries, providing operations for user-level lightweight threads (so-called tasks). They expose a POSIX Threads-like interface: one function call to create a task and another function call to synchronize a task of choice.

OpenMP and Intel TBB are widely-used implementations. Task parallel programming model has been introduced from OpenMP 3.0. Since then, many other OpenMP interfaces have been changed to be built upon task parallelism, for example, parallel-for primitive has been based on tasks from OpenMP 4.0, data-flow programming model has been built on tasks from OpenMP 4.5. It can be said that the task parallel model and task runtime scheduler are becoming the central component of OpenMP.

Intel TBB has a large set of parallel programming interfaces which are all built around a central task parallel runtime system. We can write parallel programs using easy-to-use high-level API, or we can also specify low-level details such as where a task should be executed or how it should be synchronized with Intel TBB.

3. Computation DAG Performance Model

A computation DAG is a model (also a trace file format) that represents an execution of a task parallel program. Each node of a DAG represents a serial computation

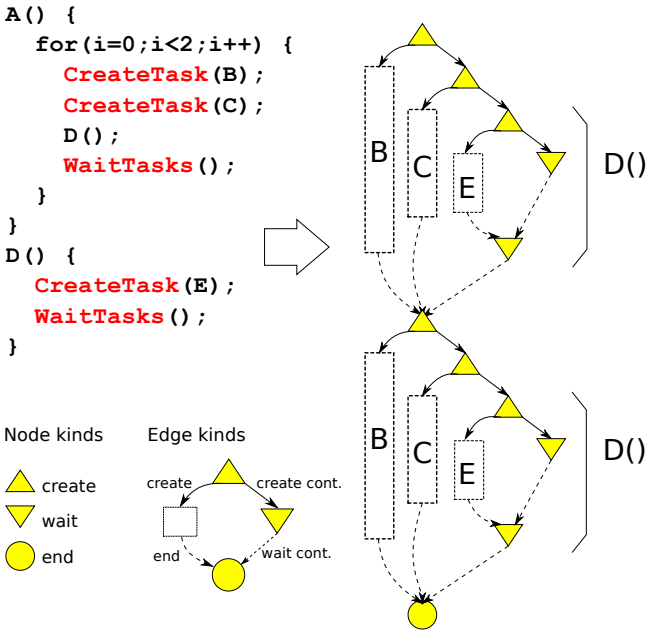


Fig. 1: An example task parallel program and its DAG. The DAG consists of two similar consecutive sub-graphs which correspond to two iterations of the loop. There are three kinds of leaf nodes, and four kinds of edges.

involving no task-related operations inside. An edge represents a dependence between two nodes. The DAG structure is not flatly stretched out, but hierarchically grouped tasks and sections. Nodes are grouped hierarchically such that a collective node (task or section) contains in it a sub-graph of other collective nodes and leaf nodes which contain no sub-graph. Thus, a DAG always starts with a single initial task node representing the whole execution. That root node can then get expanded step by step into sub-graphs of increasing depths and finally become the full graph of only leaf nodes. An example computation DAG is shown in **Fig. 1**.

Computation DAG captures task creation and joining, along with performance information such as timing, CPU cores, hardware events. Computation DAG separates the program code's execution (work) from the runtime scheduling mechanisms happening inside task parallel primitives (delay). This approach allows us to analyze the common phenomenon of work stretch when the program is run on many cores compared with its serial run on one core. The computation DAG is a dynamic instance of the program's static logical task structure, which is augmented with actual performance information (e.g., time, cache misses). This persistent task structure allows us to compare different runs of the same program by different runtime systems so that we can understand differences between them and pinpoint root causes behind those performance variations.

How a computation DAG is captured is described in our previous work [8]. This technique is not only applicable to task parallel systems which automatically schedule implicitly-dependent tasks, but also able to be applied to systems like PaRSEC which executes explicitly dependent DAG-based tasks.

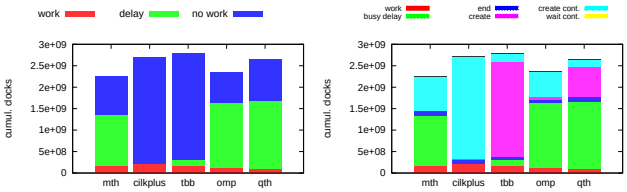
4. Critical Path Analysis

Our analysis takes as an input the computation DAG representing an execution of a task parallel program. Each node of a DAG represents a serial computation involving no task-related operations inside. An edge represents a dependence between two nodes. A critical path is a serial chain of dependent tasks from the start of the entire computation to the end. Of many such paths, our analysis is focusing on the particular path along which there is always a task ready or running. It is easy to see there is always such a path; from the end of the computation, we trace the dependence graph backwards, choosing the last finished one when a node has multiple predecessors.

Our tool analyzes how the computation progressed along this path. To this end, we classify the entire execution time into the following three parts: **work**, during which a task was running and thus is making a progress along the path; **busy delay**, during which no task on the path was running but all cores are busy working on other tasks (not on the path); and **scheduler delay**, during which no task on the path was running and there is at least one available (idle) core. Busy delay is a period in which the computation does not progress. Yet, there is little or nothing to blame the runtime system about it, as such delays are caused just by a decision to work on other tasks at that point. A large busy delay does not indicate an issue of the runtime system but just an ample parallelism in the execution. On the other hand, having a large **scheduler delay** is likely to be an artifact of an implementation of the runtime system; as there is at least one core available to pick up the ready task of that point, a better runtime system could have picked up it sooner. To further characterize implementations, we classify scheduler delays by the types of the dependencies. If that delay follows the completion of a task, it is called **end** delay. If that delay follows the creation of a task and precedes its first execution, it is called **create** delay. If that delay follows the creation of a task and precedes the parent task's continuation, it is called **create cont.** delay. If that delay is for the parent to wait for the completion of its children after issuing a synchronization instruction, it is called **wait cont.** delay.

A runtime system might have a large overhead for particular operations (task creation, work stealing, task joining, and so on) that makes scheduler delays constantly large for a specific type of scheduler delays; another runtime system might have a restriction in migrating tasks among cores so some ready tasks might not be able to run on the cores idle at that point; another system might deliberately delay the execution of a particular task in favor of other criterion, such as a better locality. We envision scheduler delay will highlight how such factors affected the execution time of the program.

The results of the critical path decompositions of SparseLU program executed by five systems are shown in **Fig. 2** which consists of two sub-figures which are corresponding to two subsequent decompositions. Because of differences in scheduling policy (to prioritize child tasks or to prioritize continuation of the parent task), MassiveThreads and Cilk Plus systems have the create cont. delay dominant; on the other hand, Intel TBB and Qthreads systems have the create delay dominant. While OpenMP sometimes has create cont. dominant, sometimes oppositely has create dominant. This is because OpenMP prioritizes parent task only until the number of ready tasks on queue reaches a predefined threshold after which it may switch to prioritize child tasks.



(a) critical path breakdown (b) scheduler delay breakdown

Fig. 2: SparseLU: critical path breakdown

App	stack	cut off	other args
Alignment	2^{20}	-	`-f prot.100.aa`
FFT	2^{15}	-	`-n` 2^{24}
Fib	2^{15}	manual	`-n 47 -x 19`
Floorplan	2^{17}	manual	`-f input.20 -x 7`
Nqueens	2^{14}	manual	`-n 14 -x 7`
Sort	2^{15}	manual	`-n` 2^{27} `-a 512 -y 512`
Sparse LU	2^{14}	-	`-n 120 -m 40`
Strassen	2^{14}	manual	`-n 4096 -x 7 -y 32`

Table 1: BOTS benchmark arguments

We have implemented this critical path analysis as an additional analysis pass in our computation DAG-based performance profiler/visualizer - DAGViz [8].

5. Case Studies

We have run our analysis with 8 benchmarks from Barcelona OpenMP benchmark suite (BOTS) [9] together with 5 different runtime systems of MassiveThreads, Cilk Plus, Intel TBB, OpenMP, and Qthreads. The arguments passing to each benchmark are summarized in **Table 1** including stack sizes, manual cut-off values, input files or problem sizes. The experiment machine is a 2.30GHz 36-core Haswell server with two 18-core Xeon E5-2699 v3 chips. We analyze the executions on full 36 cores with 36 threads, a serial version of each benchmark which runs on 1 core and replaces task spawns by normal function calls is taken as the base performance.

5.1 SparseLU

SparseLU is an LU matrix factorizing computation for sparse matrices. With problem size of $n = 120$, it has 120 phases at each of which a large amount of tasks (up to 3600 tasks) are created by a for loop. Each task does leaf computation which does not recursively spawn any further child tasks. Because only one task is created at an iteration of the loop, the computation's parallelism increments along with the progress of the loop.

In work-first runtime systems like MassiveThreads and Cilk Plus which switch to execute the child task whenever a new task is spawned, the parent task which executes task-spawning loops is left back on ready queue after every loop iteration. When a free worker steals the parent task and execute the loop's next iteration, the execution's actual parallelism increments one unit. The program's computation progress depends heavily on how fast the worker threads can do work-stealing. This situation is reflected on the scheduler delay of type *create cont.* for MassiveThreads and Cilk Plus (**Fig. 3**). Moreover, because Cilk Plus has higher work-stealing costs, it takes more time to find and migrate the ready parent task from its previous core to a free core, and its scheduler delay is much longer than MassiveThreads'. The penalty of longer scheduler delay that Cilk Plus bears is its larger nowork factor in its total execution time as free cores waits longer for available parallelism to increase.

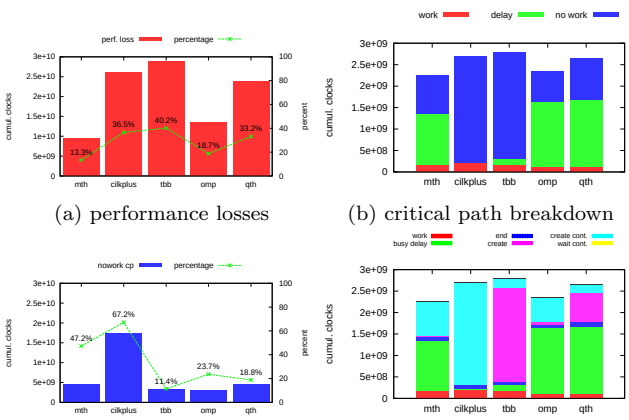
In help-first runtime systems like Intel TBB and Qthreads which keep running the parent task after a child task is created, the program's available parallelism grows fast because the loop is executed continuously on one worker without interruption and it keeps spawning child tasks into ready queue. As multiple child tasks can be stolen and distributed to multiple workers faster, these help-first systems do not suffer from larger nowork factor. Because of the nature of help-first, Intel TBB, OpenMP and Qthreads are supposed to have large scheduler delay of type create. But only Intel TBB and Qthreads do, OpenMP reversely has large amount of create cont. delay. This is because OpenMP automatically stops the execution of the parent task when the number of ready tasks waiting on run queue reaches a predefined number (around six times of the number of cores), which can be observed in **Fig. 6**.

Qthreads scheduler's one noticeable point is that it does not schedule child tasks for execution until they are being waited. As we can see from **Fig. 5**, the executions of child tasks are shifted far to the right and started at almost the same time. This characteristic explains the reason why Qthreads has so much total delay factor in its execution time breakdown. However, in compensation of this scheduler delay Qthreads somehow achieves better locality so that their tasks can execute faster than those of Intel TBB do, resulting in quite low work factor (without work stretch) compared with Intel TBB.

5.2 Sort

Sort benchmark sorts a random permutation of 2^27 32-bit numbers with a parallel variation of mergesort. The algorithm divides the input array into four quarters, sorts them separately and then merges together to form the sorted array. The division is conducted recursively until the array gets small enough for serial sorting by quicksort.

As observed from timeline visualizations of Sort's executions by MassiveThreads and Intel TBB in **Fig. 8**. In the latter half of the executions (merging phase) there is a lack of parallelism happening due to a fairly large amount of long-



(a) performance losses (b) critical path breakdown

(c) nowork caused by scheduler delay (d) scheduler delay breakdown

Fig. 3: SparseLU: breakdown of total execution times of all threads (run by mth, clkp, tbb, omp & qth)

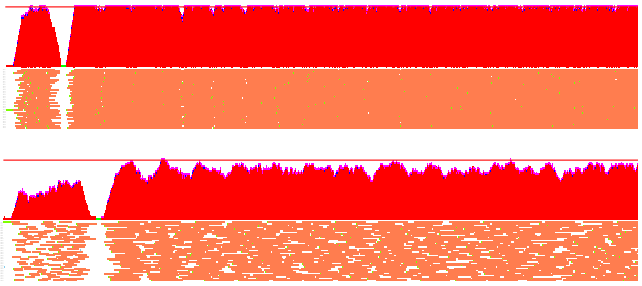


Fig. 4: SparseLU's parallelism profiles by mth vs. cilkplus

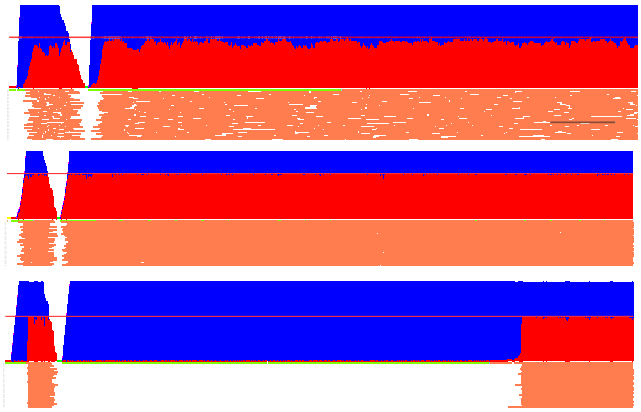


Fig. 5: SparseLU's parallelism profiles by tbb vs. omp vs. qth

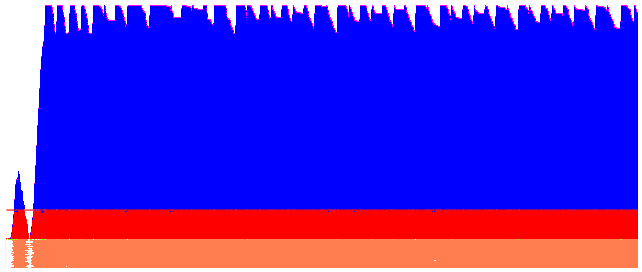
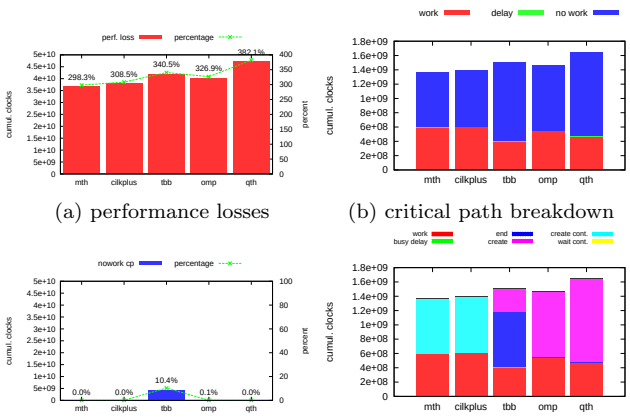


Fig. 6: SparseLU's parallelism profiles by omp

running tasks (long boxes). In the algorithm, the second phase of recursive parallel merge will turn to simple sequen-



(a) performance losses (b) critical path breakdown

(c) nowork caused by scheduler delay on critical path (d) scheduler delay breakdown

Fig. 7: Sort: breakdown of total execution times of all threads (run by mth, clkp, tbb, omp & qth)

tial memory copy whenever the smaller array in the two arrays of the merge is empty. This condition (the smaller array is empty) does not always guarantee that the larger array is sufficiently small; but contrarily, the larger array might be very large, making the sequential memory copy operation costly. This trivial condition itself causes the lack of available parallelism accompanied with many long-running tasks at the stage near the end of the execution. Only by seeing the timelines we can somehow realize that Intel TBB causes a larger amount of core idleness than MassiveThreads does during this insufficient parallelism period. Our critical path analysis method can quantify exactly Intel TBB's surplus amount of nowork as shown in **Fig. 7c**.

In Intel TBB, tasks are tied to cores where they were first executed. When two child tasks merging four sub-arrays have already finished, their parent task can only be resumed on its tied core. However, if the core happens to be busy executing a long-running task at that time, the resume of the parent task needs to wait, and this wait can be very long, causing long scheduler delay on the critical path. That scheduler delay is of type end, fit with the result shown in **Fig. 7d** that Intel TBB has a large scheduler delay of type end. On the contrary, tasks are untied in MassiveThreads so the parent task can be resumed immediately on the core that has just finished the last child task.

When the nodes that reside on the critical path are highlighted with green color on DAG as seen in Fig. 8, MassiveThreads' critical path tends to be long and continuous. On the other hand, Intel TBB's critical path contains many end nodes which are often scattered far from each other due to the wait of the tied cores to finish its long nodes.

5.3 Alignment

Alignment program aligns all 100 protein sequences from an input file against every each other sequence. The alignment are scored and the best score for each pair is provided as a result. The parallelization pattern is simple, it consists of a single for loop which has around $100 \times 100 = 10,000$ it-

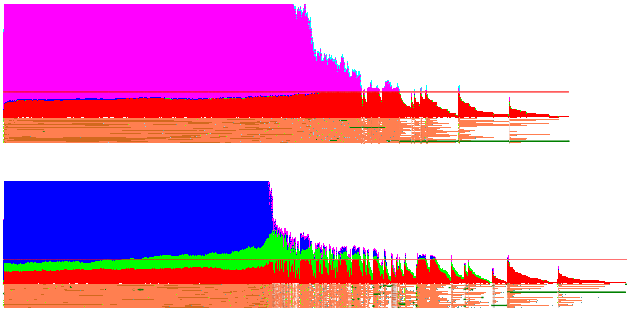


Fig. 8: Sort's parallelism profiles by MassiveThreads & Intel TBB

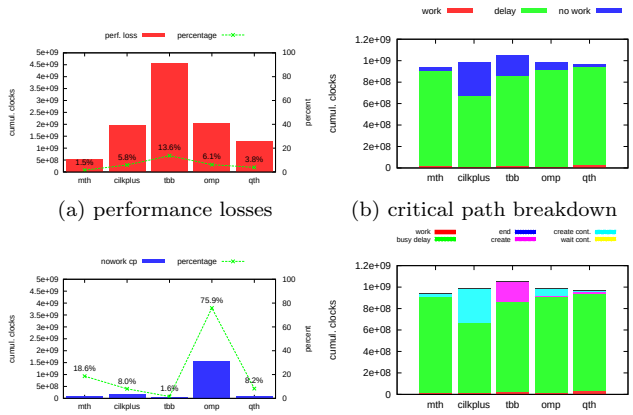
erations, at each iteration a serial child task which does not spawn any more task recursively and computes the alignment score of a sequence pair is created. Although there is a degree of load imbalance in each task, the number of tasks is abundant so there should not be any lack of parallelism in the program.

However, our critical path analysis has detected the reverse happening with OpenMP. As shown in **Fig. 9c**, the amount of nowork caused by scheduler delay on the critical path accounted for up to 76% the performance loss of OpenMP's alignment execution. This nowork factor indicates a lack of parallelism during the execution because of the scheduler's delay in delivering the ready task to a large number of free cores. A quick look into OpenMP version's parallelism profile in **Fig. 10** gives us a sense of the presence of a quite long idle period in the middle of the execution. This nowork period is caused by scheduler delay, which means that there is a ready task during the period but it is not scheduled for execution fast enough to proceed the computation's progress. Another look inside the execution's DAG visualization (**Fig. 11** tells us that the delayed ready task is the root parent task which executes the main serial for loop. This parent task has been tied to the core 34 and cannot be migrated to another core. Therefore, when this core 34 happens to execute a long running child task while all created child tasks have already been executed, the program gets out of available parallelism making cores other than 34 idle.

5.4 FFT

For FFT benchmark, our automatic detection of nowork factor caused by the scheduler delay on critical path has indicated it contributes a little in the total performance loss (4.6%) in the Qthreads version (less than 1% for other versions) (**Fig. 12**). In our experiments, FFT computes the one-dimensional Fast Fourier Transform of a vector of 2^24 complex values. It is using a divide-and-conquer algorithm which creates many tasks recursively for smaller problem sizes.

As we have known (from the SparseLU case study), Qthreads delays executions of all child tasks until the parent task has issued a join instruction. Although this characteristic causes a severe performance bottleneck in parallel pat-



(a) performance losses (b) critical path breakdown

(c) nowork caused by scheduler delay on critical path (d) scheduler delay breakdown

Fig. 9: Alignment: breakdown of total execution times of all threads (run by mth, clkp, tbb, omp & qth)

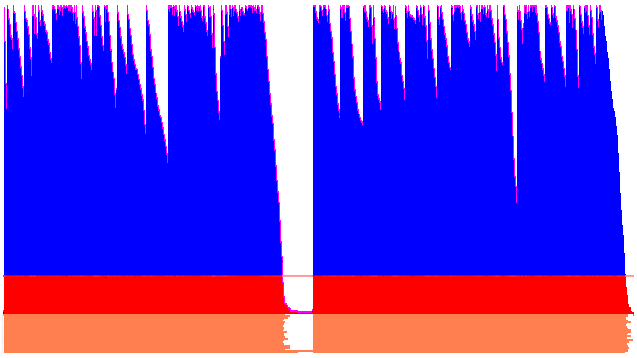


Fig. 10: Alignment's parallelism profile by OpenMP: nowork factor is significant.

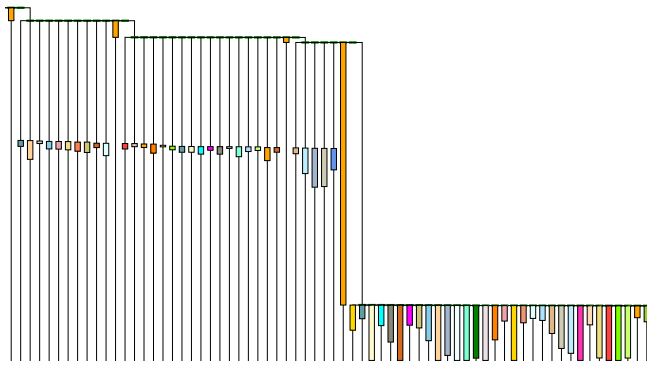
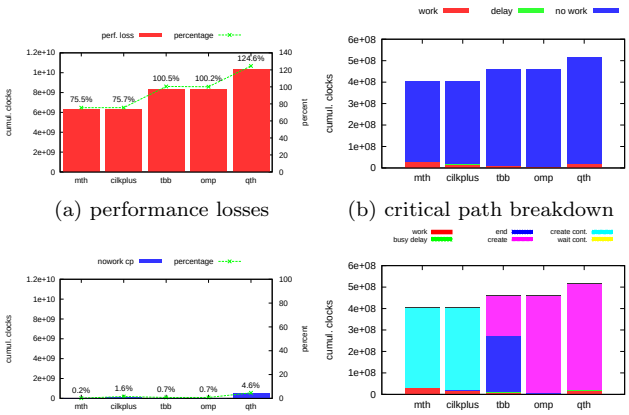


Fig. 11: Alignment's DAG by OpenMP: the long task causes a long (create cont.) scheduler delay.

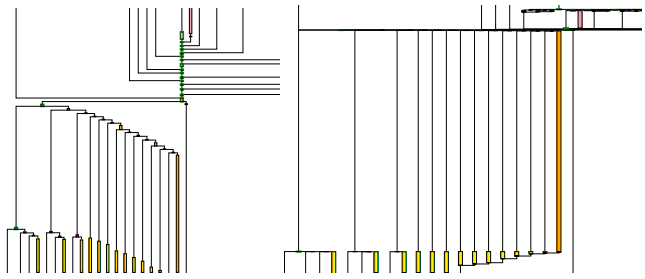
terns in which all available parallelism is created by a serial for loop, in recursive parallelism it is usually not the case as many available logical execution paths existing at the same time have compensated for the scheduler's delay. That explains its relatively low accountability of 4.6% in this case. It is noticeable just because it is much higher than other versions. And our tool DAGViz can automatically bring us directly to that area of high nowork, long scheduler delay in the DAG which is shown in **Fig. 13**.



(a) performance losses (b) critical path breakdown

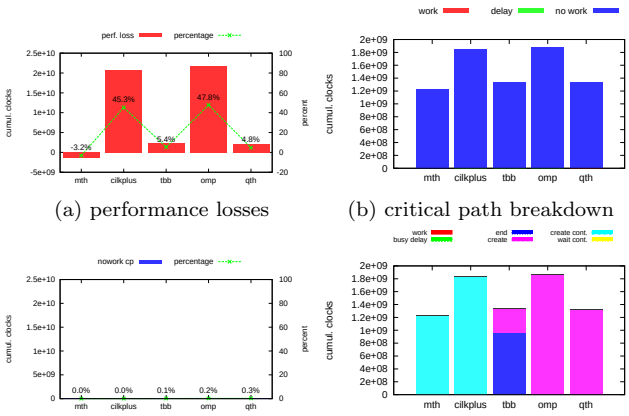
(c) nowork caused by scheduler delay on critical path (d) scheduler delay breakdown

Fig. 12: FFT: breakdown of total execution times of all threads (run by mth, clkp, tbb, omp & qth)



(a) power scale (b) linear scale

Fig. 13: FFT's DAG by Qthreads: delayed scheduling of child tasks has caused a lot of idleness in free cores which is concretized as "nowork" and "delay" factors.



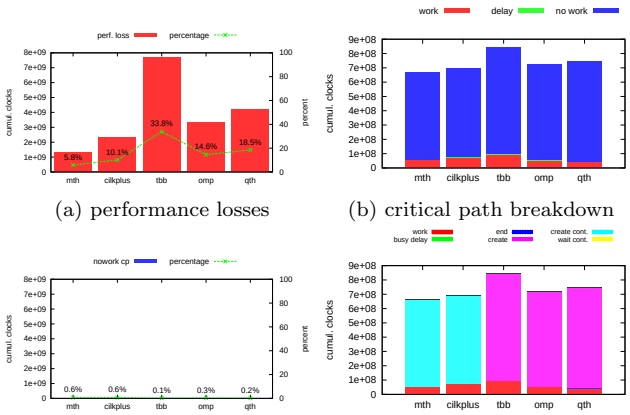
(a) performance losses (b) critical path breakdown

(c) nowork caused by scheduler delay on critical path (d) scheduler delay breakdown

Fig. 14: Fib: breakdown of total execution times of all threads (run by mth, clkp, tbb, omp & qth)

5.5 Others

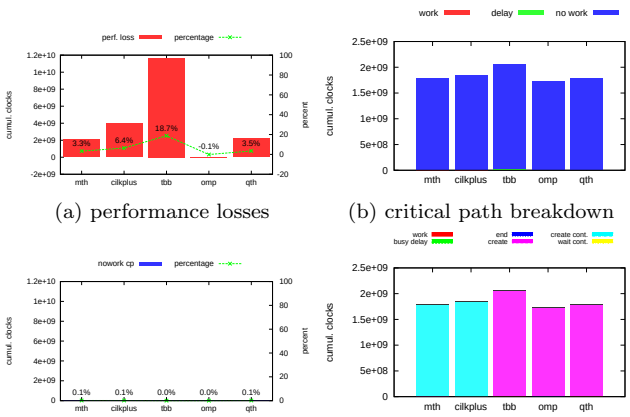
Fibonacci **Fig. 14**, Floorplan **Fig. 15**, NQueens **Fig. 16**, and Strassen **Fig. 17** seem not to be bottlenecked by the nowork factor caused by scheduler delay. Their performances are affected by other factors which will be addressed in our other papers.



(a) performance losses (b) critical path breakdown

(c) nowork caused by scheduler delay on critical path (d) scheduler delay breakdown

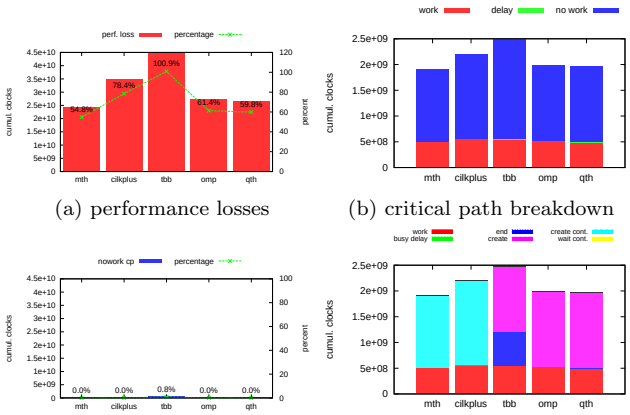
Fig. 15: Floorplan: breakdown of total execution times of all threads (run by mth, clkp, tbb, omp & qth)



(a) performance losses (b) critical path breakdown

(c) nowork caused by scheduler delay on critical path (d) scheduler delay breakdown

Fig. 16: NQueens: breakdown of total execution times of all threads (run by mth, clkp, tbb, omp & qth)



(a) performance losses (b) critical path breakdown

(c) nowork caused by scheduler delay on critical path (d) scheduler delay breakdown

Fig. 17: Strassen: breakdown of total execution times of all threads (run by mth, clkp, tbb, omp & qth)

6. Related Work

Tallent et al. [10] have broken down execution time into three components of work, overhead, and idleness, which is similar to our work - delay - nowork decomposition. They have used these overhead and idleness measures to identify which code regions to increase parallelism and which code regions to decrease parallelism.

Olivier et al. [11] have taken a step further that they identified work inflation (the increase in work when changing from serial execution to multithreaded execution) as a critical parallel performance bottleneck which sometimes surpasses parallel overhead and idleness. Because work inflation occurs due to the increased overhead in getting data from remote module in NUMA architecture. They have developed a locality-aware scheduler which places tasks near to their data, and demonstrated that this scheduler had mitigated the work inflation of two benchmarks *health* and *heat*.

DAG has been used broadly in literature to model a static parallel program. However, we leverage it to model a dynamic parallel execution. Our measurement is instrumentation-based so it might seem to have large overhead. But we have implemented a mechanism to collapse DAG dynamically on-the-fly to keep both time overhead and memory overhead under reasonable limit. In actual experiments with BOTS, our tool's overhead is less than 10%.

7. Conclusion

We aim to build a performance tool to discover automatically performance differences between systems. We have developed an additional analysis focusing on critical path, and made our visualization tool to highlight the critical path on DAG visualizations.

By breaking down critical path length into work, safe delay & problematic delay which is further decomposed into different sub-components of end, create, create cont. & wait cont. delays, we have successfully contrasted five different runtime systems of interest and emphasized reasons why one system is performing better than another on a particular application.

References

[1] OpenMP Architecture Review Board: OpenMP Application Program Interface, Technical Report July, OpenMP Architecture Review Board (2011).
[2] Leiserson, C. E.: The Cilk++ concurrency platform, Proceedings of the 46th Annual Design Automation Conference DAC '09, ACM Press (2009).
[3] Pheatt, C.: Intel(R) Threading Building Blocks, J. Comput. Sci. Coll., Vol. 23, No. 4, pp. 298–298 (2008).
[4] Wheeler, K. B., Murphy, R. C. and Thain, D.: Qthreads: An API for programming with millions of lightweight threads, 2008 IEEE IPDPS, IEEE, pp. 1–8 (2008).
[5] Nakashima, J., Nakatani, S. and Taura, K.: Design and implementation of a customizable work stealing scheduler, Proceedings of the 3rd International Workshop on Runtime and Operating Systems for Supercomputers - ROSS '13, ACM Press (2013).
[6] Nakashima, J. and Taura, K.: MassiveThreads: A Thread Library for High Productivity Languages, Festschrift of Symposium on Concurrent Objects and Beyond: From Theory to High-Performance Computing (to appear as a volume of Lecture Notes in Computer Science) (2012).
[7] Blumofe, R. D. and Leiserson, C. E.: Scheduling multithreaded computations by work stealing, Journal of the ACM, Vol. 46, No. 5, pp. 720–748 (online), DOI: 10.1145/324133.324234 (1999).

[8] Huynh, A., Thain, D., Pericàs, M. and Taura, K.: DAGViz: A DAG Visualization Tool for Analyzing Task-parallel Program Traces, Proceedings of the 2nd Workshop on Visual Performance Analysis, VPA '15, ACM, pp. 3:1–3:8 (2015).
[9] Duran, A., Teruel, X., Ferrer, R., Martorell, X. and Ayguade, E.: Barcelona OpenMP Tasks Suite: A Set of Benchmarks Targeting the Exploitation of Task Parallelism in OpenMP, 2009 International Conference on Parallel Processing, IEEE, pp. 124–131 (2009).
[10] Tallent, N. R. and Mellor-Crummey, J. M.: Effective Performance Measurement and Analysis of Multithreaded Applications, Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '09, ACM, pp. 229–240 (2009).
[11] Olivier, S. L., de Supinski, B. R., Schulz, M. and Prins, J. F.: Characterizing and Mitigating Work Time Inflation in Task Parallel Programs, SC '12, IEEE Computer Society Press, pp. 65:1–65:12 (2012).