博士論文

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)

平成 29 年 12 月 8 日提出

指導教員　　田浦 健次朗　教授

東京大学
THE UNIVERSITY OF TOKYO

フィン ゴクアン (Huynh Ngoc An)

Abstract

The number of processor cores integrated in a computer keeps increasing with *multicore* and *manycore* 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.

Contents

List of Figures

List of Tables

Chapter 1

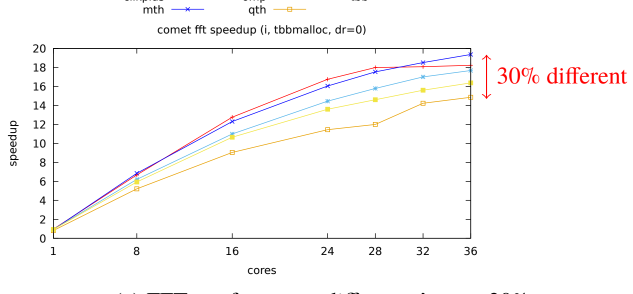
Introduction

1.1 Motivation

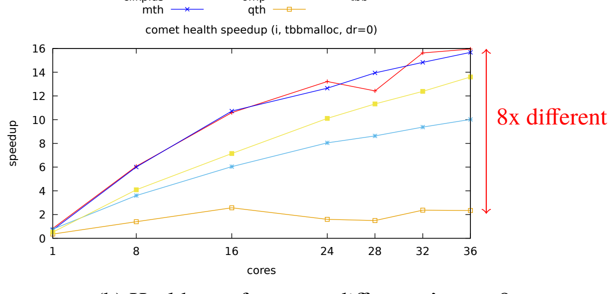
Modern task parallel programming models are equipped with sophisticated runtime systems which are responsible of scheduling a large number of tasks onto available hardware resources at runtime. As the hardware parallelism keeps increasing with more nodes, more cores, more threads, it would be difficult for programmers to write efficient parallel programs with traditional parallel programming models like MPI, POSIX Threads. In these traditional parallel programming models, programmers need to manually manage threads, break down the work accordingly with available threads, and schedule these work to threads during runtime. These low-level details in managing a parallel execution are intricate, and practically distract programmers from other higher-level aspects in the application development.

A runtime system in the middle that abstracts away the underlying threads and processor cores, while providing a unified interface of logical tasks to upper layers is a common approach. Programmers just need to extract logical parallelism from the program's algorithm by creating tasks. The runtime system will take care of mapping these tasks to available threads at runtime so that these logical parallelism becomes actual parallelism as much as possible.

The relief of burdens on programmers are accompanied by a larger reliance on the runtime systems. As a runtime system takes care of more things in the parallel execution mechanism (e.g., thread management, scheduling, load balancing), the execution's performance depends largely on it. Different runtime systems can perform largely different because of their differences in the design, scheduling algorithms and implementation quality. Fig. 8.22 show two examples of speedup differences. In a mild case, five runtime systems (Cilk Plus, MassiveThreads, OpenMP, Qthreads, TBB) differ only 30% when executing the FFT benchmark (Fig. 1.1a). In a harsh case, their differences were up to 8x in Health benchmark (Fig. 1.1b).



(a) FFT: performance difference' up to 30%



(b) Health: performance difference' up to 8x

Figure 1.1: Performance differences between different runtime systems running the same program.

Clarifying causes behind these performance differences is important for improving task parallel programming models. Almost all systems have adopted the well-known work-stealing scheduling policy in their schedulers, their differences in the implementations are now presumed to be sources of performance variation.

Moreover, the runtime systems sometimes perform prominently well in some applications, and sometimes noticeably poorly in some other applications. It can be said that performance differences are not the results of the scheduling policy of the runtime system only, but more of the results of the interplay of the scheduler's policy and the application's behaviors. Therefore, analyzing the runtime systems with a large set of applications is necessary.

1.2 Contributions

We have developed a kind of analysis for contrasting performance between difference runtime systems. The analysis is based on scheduling delays which are artifacts imposed by the runtime schedulers on the execution of a task parallel program. We have also implemented a tool to realize that kind of analysis, and applied it in a large number of benchmark programs (10 benchmarks in BOTS suite, 11 benchmarks in TP-PARSEC suite). The PARSEC benchmark suite is originally written with traditional parallel programming models like POSIX Threads, OpenMP and TBB's loop-parallel primitives. So before we could use them with our task parallelism-based analysis and tool, we had first translated PARSEC to task parallelism, hence the name TP-PARSEC which is task parallel PARSEC.

1.3 Organization of the rest of this thesis

In Chapter 2, Chapter 3, and Chapter 4, the background of this work is discussed. Chapter 2 describes the common scheduling strategy that almost all systems use nowadays - work stealing. Chapter 3 describes the five main task parallel programming models that are studied in this work. Chapter 4 describes common and well-known existing performance analysis and visualization tools. Chapter 5, Chapter 6, and Chapter 7 are the main meat of this work. Chapter 5 describes our proposed scheduling delay-based analysis for differentiating performance between different runtime systems. Chapter 6 describes the toolset we have implemented in order to realize the analysis. The toolset includes a tracer named DAG Recorder and a visualization GUI tool named DAGViz. Chapter 7 describes how we have translated the original PARSEC benchmarks to task parallelism-based TP-PARSEC benchmarks. Chapter 8 is the evaluation of the two benchmark suites (BOTS and TP-PARSEC). We have evaluated both benchmark suites on two machines, one is equipped with 36 Xeon cores (Comet), and the other is a 68-core Xeon Phi (KNL) machine (Denebola). Chapter 8 only discusses some selective points in the experimental results of BOTS and TP-PARSEC on a the Xeon machine. A full set of resulting graphs of BOTS and TP-PARSEC on both Comet and Denebola are included in the Appendices.

Chapter 2

Work-Stealing Scheduling Strategy

Work stealing has been widespread since its debut, and deployed widely in many parallel systems such as Cilk [13], Cilk++ [42], Cilk Plus [1] [61], Java fork/join framework [39], .NET Task Parallel Library [41], Qthreads multithreaded library [71], MassiveThreads multithreaded library [52], OpenMP Tasks [7], Intel TBB [57].

2.1 Brief Description

A parallel runtime system typically launches a fixed number of threads at the start of the program's execution. Each thread is bound to a separate single processor core, and usually referred to as a *worker thread* or simply a *worker*. The runtime system's scheduler is responsible for load-balancing the tasks which are created by the program to these available worker threads.

The primary aspect of this load balancing job is to answer the question how to move jobs from a busy worker to an idle worker. There were two possible schemes to deal with this problem: *work sharing* and *work stealing*. In work sharing, when a thread has created tasks, it takes initiative and attempts to migrate some of them to other threads in hopes of distributing the work to underutilized threads. On the contrary, in work stealing the idle threads, not the busy ones, would take initiative and go stealing tasks from other threads. By putting the load balancing overhead to idle threads, work stealing incurs less interruption to the busy threads who are working on the main computation, resulting in shorter critical path, better performance. Besides, instead of migrating tasks in hopes of an assumed balanced distribution even at the times when all workers have been fed up and busy working on their own works, work stealing invokes task migration only when a thread is starving for work. That is to say work stealing incurs less task migration than work sharing.

The idea of work stealing can date back to 1980s, but it was Blumofe et al. who proved theoretically the efficiency of the randomized work stealing scheme in their work in 1994 [12]. "Randomized" here refers to the way how a starving thread chooses a victim thread to steal from, which is at random and usually with uniform distribution. They have also applied their work stealing technique in practice with the Cilk language [13] and its runtime system, showing that work stealing is also practically efficient. Therefore, work stealing has been a load balancing method of choice, and adopted widely in commercial and open-source implementations of task schedulers.

Generally, in the execution of a task parallel program, each worker thread maintains a work queue of its own containing tasks that are waiting for execution. When a worker has no more task in its queue to execute, it goes to another worker randomly to steal a task from that worker's queue. At the beginning of a program execution, there is only one task, which is the main program, in the queue of the master worker thread. When a worker creates a new task, it has two choices to proceed. One is to pause the current task to switch to executing the new task. The other is the opposite; the worker pushes the new task into its work queue and continues executing the current one. These approaches are usually referred to as *work-first* and *help-first* respectively [48]. *Work-first*'s execution order is similar to that of a serial execution, so it tends to maintain the data locality that exists in the serial execution [3]. *help-first* tends to expose higher parallelism when many tasks are created by a serial loop. OpenMP, TBB and Qthreads adopt *help-first* policy in their schedulers. Cilk Plus and MassiveThreads adopt *work-first*.

2.2 Chronological Description

The idea of work stealing can date back to 1980s with Burton et al.'s research in 1981 on an execution model for parallel functional programs on a connected network of computers (processors) [15], and Halstead's implementation of the Multilisp language in 1984 [26] [27]. Burton et al. referred to the scheduling scheme as *work diffusion*, and Halstead referred to his scheme as *unfair scheduling policy*. And later on, in 1990 Mohr, Kranz, and Halstead improved the scheduling algorithm and renamed it as *lazy task creation* [48] [47] (which was first implemented in the Mul-T language [37]). Then in 1994, Blumofe et al. [12] [14] have provided a theoretical proof of the efficiency of work stealing, and officially named the scheduling scheme as *work stealing* which is also the name widely used nowadays. They also first implemented the scheme in Cilk language [13]. In 1998, Frigo et al. [24] revised the implementation of runtime scheduling of Cilk system with its 5th version (Cilk-5) by integrating and immersing deeper the *work-first* principle.

2.2.1 Work diffusion (1981)

The work of Burton et al. [15] was a part of the ZAPP (Zero Assignment Parallel Processor) project at the University of East Anglia (England) which aimed to demonstrate the feasibility of achieving speedup on large numbers of computing elements. The parallel *process tree* (task graph) virtually made up at runtime by parallel programs (especially ones with divide-and-conquer style) is distributed to available processors via immediate neighbours (connected pairs of processors). Topologically adjacent processors who are underutilized are allowed to *steal* processes from each other, thus creating the scheduling basis of *work diffusion*. Each channel connecting a pair of processors supports two-way communication, and repeatedly exchanges packets of information of fixed size. These information packets may include information like available processes, results produced by terminated processes, requests by one processor for a process on another processor, transfers of requested processes, etc. so that each processor will have *recent information* about the state of its neighbouring processors so that it can perform necessary interactions.

The process tree consists of nodes (processes) which are either *active*, *pending*, or *blocked*. A processor may have multiple *active* processes running at a point in time. *Pending* processes which are ready but not run yet may be stolen by neighbours who have run out of available processes. A running process may either *terminate* (and return result to its parent) or become *blocked* (to wait for the result of another running process). A blocked process can only become active again on its current processor, i.e., a blocked process cannot be stolen. This constraint is to ensure that each process is always on the same processor of its parent or an adjacent processor (i.e., stolen only once).

For load control, although they acknowledge the general heuristic that encourages a *breadth-first expansion* of the process tree when the network is underutilized, and a *depth-first expansion* at other times, acquiring or maintaining a global state of all processors on the network is expensive and unlikely feasible. Therefore, in practice a local basis can be used: if a processor is fully or overutilized, it can just discourage parallelism. This scheme of allowing neighbours to steal tasks when underutilized provides a basis for work diffusion with an exponential diffusion rate.

2.2.2 Unfair scheduling policy (1984)

Halstead (MIT, USA) proposed a new language *Multilisp* based on the Scheme dialect of the Lisp language families, which natively supports constructs for expressing concurrency [26] [27]. Using these constructs, programmers can explicitly specify concurrency which might not be found by automated analysis at compile time or run time. Multilisp's implementation techniques (task scheduling and garbage collection) were first discussed in the conference paper in 1984 [26], then the language's design, implementation, and background were discussed in more details in a longer journal paper in the consecutive year 1985 [27].

Multilisp supports two additional constructs for expressing concurrency: `pcall` (parallel call), and `future` (i.e., a placeholder for a value to be determined in the future). `pcall` construct allows concurrency between the evaluations of two or more arguments to a function. For example, the expression

$$(\texttt{pcall} + A\ B)$$

in which A and B are two operands to the addition operation is equivalent to the sequential expression

$$(+\ A\ B)$$

except that the first expression has A and B evaluated concurrently before passing their results to the addition operation.

`future` construct enables an additional form of concurrency: the concurrency between the computation of a value and the use of that value. `future` allows a computation to proceed past the calculation of a value without waiting for that calculation to complete, and if the value is never used, the computation will never pause to wait for the calculation of that value to finish. The `future` construct is actually more fundamental, e.g., a `pcall` can be implemented in terms of `future` like following:

$$(\texttt{pcall}\ + A\ B) \quad \text{is equivalent to} \quad (+\ (\texttt{future}\ A)\ (\texttt{future}\ B))$$

`pcall` construct is supported mainly because there may well be situations in which a programmer feels confident that two expressions A and B can safely be evaluated in parallel with each other, but feels less sure about the safety of evaluating A concurrently with the arbitrary subsequent computation of the program. Hence, `pcall` is a more conservative approach to introducing parallelism compared with `future`.

Task creation is made through these `pcall` and `future` constructs. For example, the expression $(\texttt{pcall} + A\ B)$ will create two tasks evaluating A and B; $(\texttt{future}A)$ will create one task evaluating A. Multilisp's task scheduling policy aims for two primary goals: to preserve the locality and to avoid creating excessively tasks. Ideally, tasks should be created until the parallel machine is fully utilized (i.e., *saturation*) and then the execution within each task should become sequential. An *unfair* scheduling policy is used to produce this behavior.

It is normal that a processor has more than one active tasks which share the processing power in a round-robin manner. But Multilisp tries to avoid increasing the number of simultaneously active tasks per processor. Two tasks created from the expression $(\texttt{pcall} + A\ B)$ are treated unfairly, the processor devotes all its resources to only one task and puts the other to an associated LIFO pending queue, instead of making them both active. If the system is saturated, a pending task will remain pending until the processor finishes all preceding tasks and switches to it, as would occur in the sequential execution. Thus a task will eventually be executed by the same processor that created it (unless some other processor who runs out of tasks steals it), which helps preserve locality of memory references. This unfair scheduling mechanism also helps prevent an explosion of parallelism that is possible if A and B recursively invoke `pcall`. The same mechanism is applied with `future`: the newly created task is placed on the pending queue, while the parent task is kept active.

As discussed, each task has two possible states: *active* and *pending*. When finishing all active tasks, the processor looks at its queue of pending tasks to find one to activate. If the queue is empty, it looks in the queues of other processors to find a task to *steal*. When a processor steals a task, it would take the oldest task which is likely to be the root of a larger tree of computation. Moving large quanta of computation between processors like this helps enhance the locality.

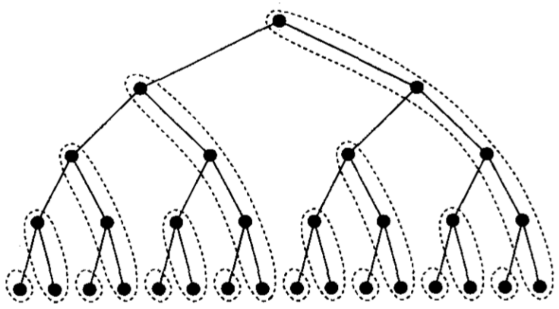
2.2.3 Lazy task creation (1990)

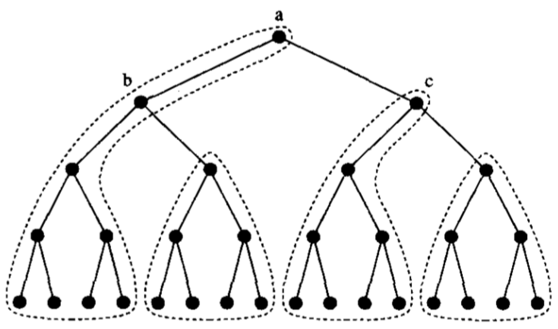
Halstead, Kranz, and Mohr have improved the unfair scheduling policy in Multilisp several years after the Multilisp work, out of their work on another parallel version of Scheme *Mul-T*. They named the scheduling algorithm as *lazy task creation* which is implemented in Mul-T language system. The design and implementation of the Mul-T language was published in 1989 (Kranz, Halstead, and Mohr [37]); the lazy task creation scheduling algorithm was published the next year 1990 (Mohr, Kranz, and Halstead [48]), and republished on a journal in 1991 [47].

Mul-T [37] is also based on the `future` construct for generating parallel tasks. The expression (`future` X) creates a task for evaluating the expression X, and also creates an object known as a *future* to eventually hold the value of X. Until X is finished evaluating, the future is considered as *unresolved*, or *undetermined*. By returning the future as a placeholder the program can proceed without waiting for the evaluation of X to finish, hence a unit of parallelism is forked. With this programming style, a programmer can add a small number of `future` constructs to make the program parallel, but there is an efficiency issue about task granularity. There are usually too many fine-grained tasks created by `future`, especially in divide-and-conquer programs (i.e., too much parallelism for a parallel machine to exploit efficiently). They have dealt with the problem by improving the implementation of `future`, instead of relying on a parallelizing compiler to detect it (which is nearly impossible in many cases) or requiring programmers to manually specify it case by case (which degrades the language's programmability).

Mohr et al. [48] [47] improved the implementation of the `future` construct and the task scheduler based on an observation that executing the task specified by a `future` construct in parallel with the parent is *permissible but not required*. The expression (`future` X) does not have to always create a separate task X, instead it can effectively *inline* X as a subroutine, eliminating the task creation, task scheduling, placeholder creation overheads and reducing the number of ready tasks kept on the queue. With the expression (K (`future` X)), it is also correct for the parent task to compute X first (i.e., inlining X), then compute K, ignoring the `future`.

The ideal task creation is one that expands the task tree breadth-first by creating separate tasks at every future until all processors are busy, then switches to depth-first expansion by inlining `future`s to avoid excessive task creations. This ideal scheduling is referred to as *BUSD (breadth-first until saturation, then depth-first)*. Fig. 2.1 depicts visually the difference in task granularity of the conventional eager task creation which spawns task at every `future` with an ideal BUSD task creation which spawns just enough tasks for the available number of processors in the underlying system (4 in this case).


(a) eager task creation [47]



(b) BUSD task creation [47]

Figure 2.1: The conventional eager task creation creates unnecessarily too many tasks which are fine-grained, while an ideal task creation which follows the BUSD manner will create just enough tasks for a 4-processor system by spawning at first 3 occurrences of `future` (a, b, and c) and inlining at subsequent occurrences.

However, the real challenge of realizing the BUSD is how to get to know when tasks are enough to stop spawning and start inlining. Mohr et al. have raised and compared two scheduling algorithms for approximating that ideal BUSD behavior: *load-based inlining* and *lazy task creation*, and concluded the superiority of lazy task creation over load-based inlining in their work [48] [47]. Load-based inlining is a simple strategy that says "if the system is not loaded, make a separate task to evaluate X; otherwise inline X, evaluating it in the current task". A threshold T is predefined for identifying whether a processor is loaded or not, and a processor will inline all `future`s encountered when the number of tasks on the processor's queue has become greater than the threshold T. For example, if $T = 0$, all `future`s are inlined and no parallel tasks are created; if $T = 1$, the existence of a single queued task will be enough to suppress task creation.

Lazy task creation is a strategy to "start evaluating X in the current task, but save enough information so that its continuation K can be moved to a separate task if another processor becomes idle". Lazy future is essentially a revocable inlining mechanism: when a future is encountered, its task is provisionally inlined, but enough information is retained in order to reverse the inlining decision at a later time if needed.

Load-based inlining has some obvious disadvantages compared with lazy task creation like: (1) the programmer must decide the threshold T; (2) only the local load of the current processor is considered, ignoring the global situation; (3) inlinings are *irrevocable* so their purpose fails in programs with *bursty task creation* pattern (i.e., opportunities to create tasks are distributed unevenly across the program).

The authors have described two methods for implementing the lazy task creation: a *stack-based* implemenetation on the Encore Multimax multiprocessor machine, and a *linked-frame-based* implementation on the ALEWIFE multiprocessor machine. But they both involve a basic operation of splitting an existing stack, required when stealing a continuation.

Fig. 2.2 depicts the story of a lazy task queue [47]. Besides the normal stack holding function frames, a task is also associated with a queue storing lazy tasks which are pointers to the continuations of the `future` constructs (a). When the task encounters a `future`, a new continuation frame K_t representing all remaining computation is appended to the stack; and at the same time a new pointer to the continuation frame is added to the tail of the lazy task queue (b). In case that no other processor has stolen the continuation when the task finishes the `future`, it will return to the continuation frame and pop it off the stack (c). In case that an idle processor steals a continuation, the thief will steal from the head of the queue and change the stack to appear as though *an eager `future` had been created* at the time of that continuation (d). A placeholder for the future value is also created to connect the still ongoing task with the stolen continuation.

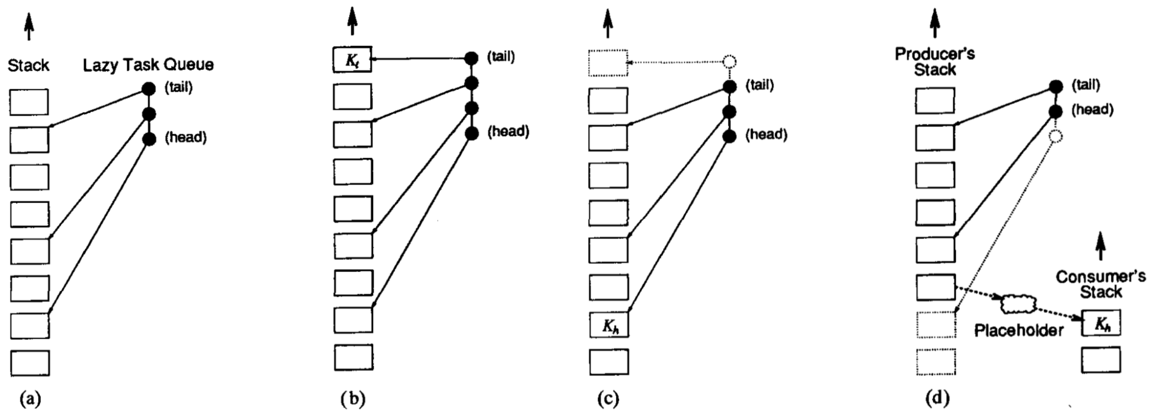
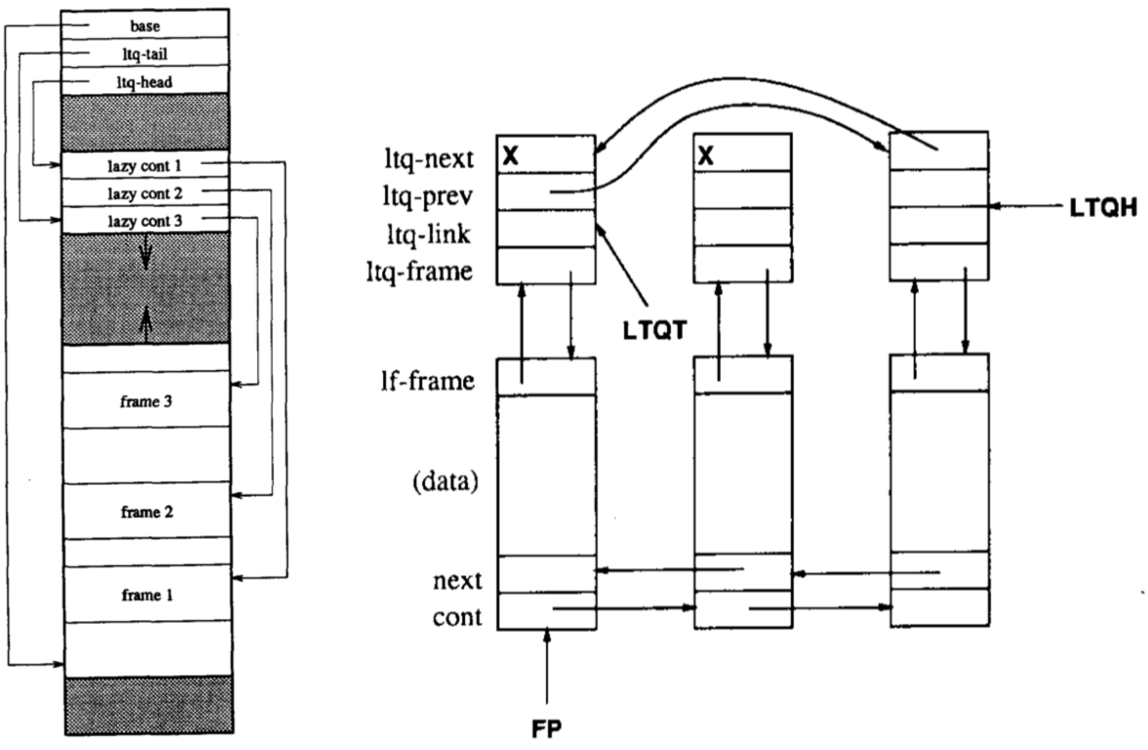


Figure 2.2: An additional lazy task queue structure for storing pointers to lazy tasks (i.e., continuations of `futures`) aside the usual stack structure: (a) stack and lazy task queue grow upward; (b) a `future` causes a continuation to be queued; (c) the continuation gets dequeued when the `future` returns; (d) a continuation at head of the queue gets stolen [47].

An implementation of a lazy task queue must take care of two kinds of race conditions: (1) two thieves race to steal the same continuation; (2) the victim tries to return to the same continuation that a thief is trying to steal.

Stack-based implementation

In this implementation, a stack is represented conventionally in a contiguous section of the heap. The lazy task queue is kept in the top part of the stack and grows downwards, while the stack grows upwards (Fig. 2.3a). A stealing operation will require *copying* a part of the stack from its bottom up to the continuation frame to be stolen. This copying is the most costly part of this stack-based implementation.



(a) stack-based (b) linked-frame-based

Figure 2.3: Two kinds of implementations of the lazy task queue [47].

Linked-frame-based implementation

In this implementation, a stack is represented as a doubly linked list of stack frames in order to minimize copying in the stealing operation (Fig. 2.3b). Each frame has a link to the previous frame (cont), a link to the next frame (next), and another link to the *frame stub* structure (lf-frame) which constitutes the lazy task queue.

2.2.4 Work stealing (1994)

Blumofe et al. [12] [14] has theoretically proven the efficiency of work stealing by using the novel metrics of *work* (T_1) and *critical path length* (T_∞) (sometimes also called *span*). Work stealing has two flavors of “continuation stealing” and “child stealing” which refer to the choice of actions a worker does at a task creation. In “continuation stealing”, when creating a new task the worker switches to that task, leaving the current task on its deque so that the continuation of the task may be stolen by another worker. This stealing style is also referred to as “child-first” or “work-first”, and systems that employ this style are, for example, Cilk, Intel Cilk Plus, MassiveThreads. On the other hand, in the “child stealing” strategy, the worker continues executing the current task, leaving the newly created child task on its deque for stealing. This stylel is also referred to as “parent-first” or “help-first”; it is easier to implement as a library, without compiler support. Some systems that employ “help-first” are Intel TBB, .NET Task Parallel Library, OpenMP Tasks, Qthreads.

Randomized work stealing has a pitfall of underming locality of tasks in accessing data, because tasks are stolen and migrated to *random* workers, away from their accessed data. Some localized variants of work stealing, in which a thief attempts to steal back its own work, have been studied in literature [3] [67].

2.2.5 Work-first (1998)

In [48], combination of lazy task creation and oldest-first stealing is proposed in order to get good task granularity at runtime. There are two kinds of locks to guard against two kinds of race conditions: (1) a lock for each task deque to prevent two thieves or more from racing to steal the same victim; (2) a lock for each task on a task deque to prevent the race where the local worker tries to get back the same task that a thief is trying to steal. The thief chooses a victim by a round-robin search of other workers' task deque.

In Cilk-5 [24], the principle of *work-first* has been integrated more completely and deeper into Cilk. Specifically, the work-first principle is reflected in their proposed two novel strategies: (1) *two-clone* compilation, and (2) a Dijkstra-like mutual-exclusion protocol for implementing the ready deque (*THE* protocol).

Work-first is the principle of minimizing overheads that contribute to the work, even at the expense of overheads that contribute to the critical path. Simply speaking, it is to move the overheads out of work and onto the critical path.

Cilk-5: integrated work-first more completely

- **two-clone** compilation
- **THE protocol** for implementing ready deque (Dijkstra-like mutex protocol)

The work-first principle follows three assumptions:

1 work-stealing scheduler according to the theoretical analysis presented in [12] [14]

2 ample parallel slackness exists (i.e., average parallelism exceeds the number of processors by a sufficient margin)

3 "every Cilk program has a C elision against which its one-processor performance can be measured"

The work-first principle pervades the Cilk-5 implementation. The work-stealing scheduler guaranteed that with high probability, only $O(PT_\infty)$ steal (migration) attempts occur (i.e., $O(T_\infty)$ on average per processor), all costs for which are borne on the critical path.

2.3 Theoretical Proof of Efficiency

theoretical analysis based on an abstract model that ignores real-life details such as memory-hierarchy effects; two fundamental lower bounds of run time:

- $T_P \geq T_1/P$
- $T_P \geq T_\infty$

assuming an ideal parallel computer, Cilk's randomized work-stealing scheduler executes in expected time:

- $T_P = T_1/P + O(T_\infty)$
- $T_P \leq T_1/P + c_\infty T_\infty$
  (the critical path overhead c_∞ is the smallest constant that satisfies the inequality)

some definitions:

- average parallelism: $\overline{P} = T_1/T_\infty$ (i.e., maximum possible speedup)

- parallel slackness = $\overline{P}/P$

assumption of ample parallel slackness: $\overline{P}/P \gg c_\infty$ (i.e., the number of processor P is much smaller than the average parallelism $\overline{P}$), hence,

(using align environment)

$$\begin{aligned} & \overline{P}/P \gg c_\infty \\ \Leftrightarrow \quad & (T_1/T_\infty)/P \gg c_\infty \\ \Leftrightarrow \quad & T_1/P \gg c_\infty T_\infty \end{aligned}$$

(using array environment)

$$\begin{array}{rrl} & \overline{P}/P & \gg c_\infty \\ \Leftrightarrow & (T_1/T_\infty)/P & \gg c_\infty \\ \Leftrightarrow & T_1/P & \gg c_\infty T_\infty \end{array}$$

hence,

$$\left.\begin{array}{ccccccc} T_1/P & \leq & T_P & \leq & T_1/P & + & c_\infty T_\infty \\ & & & & T_1/P & \gg & c_\infty T_\infty \end{array}\right\} \Rightarrow T_P \approx T_1/P$$

assuming T_S is the running time of the C elision of the Cilk program, the work overhead is defined as $c_1 = T_1/T_S$, hence $T_P \approx T_1/P \approx c_1 T_S/P$.

Work-first principle: to minimize c_1, even at the expense of a larger c_∞, because c_1 has a more direct impact on performance.

2.4 Practical Implementation

Work queue is an essential and critical component of a runtime scheduler. The more it is optimized the better performance is gained. Specifically, its implementation needs to be lock-free and optimized to reduce possible concurrent contention scenarios to the minimum. Arora, Blumofe and Plaxton [5] describes a basic scheme of work queue which is used in original Cilk runtime (ABP work stealing). The work queue is organized as a double-ended queue (deque) having two ends, top and bottom. The local worker thread operates only on the bottom end, and other worker threads who come for work stealing attempts operate only on the top end. A worker thread pushes and pops tasks from its deque in a LIFO (last in first out) manner, while it operates with a FIFO (first in first out) manner when interacting with other worker threads' deques. The deque provides three kinds of operations:

- `pushBottom`: a local thread appends a task into its deque from the bottom end
- `popBottom`: a local thread removes the first task from the bottom end of its deque
- `popTop`: a remote thread removes the first task from the top end of the victim's deque

This separation between the local thread's operations and remote threads' operations to different ends of the deque minimizes contention in deque accesses. Only local thread can manipulates the bottom end so it can process fast without so many contentions. The top end can possibly be manipulated by all other threads, hence it requires a synchronization mechanism to manage their concurrent accesses. When multiple remote threads come to acquire the top task of a deque at the same time, their pops need to be synchronized so that they do not accidentially get the same task. Another contention scenario is the case that there remains only one single task in the deque and the local thread and some remote thread both attempt to acquire that task at the same time.

Their pop and stealing attempts need to be synchronized so that they do not both get the same task. ABP work stealing employs a non-blocking lock-free implementation of deque which uses only two *compare-and-swap* (`cas`) operations, one is in the `popTop` function and another is in `popBottom` function.

The compare-and-swap operation `cas(addr,oldval,newval)` is atomic and takes three operands: an address `addr` pointing to the memory location whose value needs to be replaced, an old value `oldval` used to compare the memory location's current value with, and a new value `newval` used to swap with the memory location's current value. `cas` first compares the value stored in the memory location `addr` with `oldval`, if they are equal `cas` *swaps* the value at `addr` with `newval`. Otherwise, `cas` only copies the value at `addr` to `newval`, leaving the memory location `addr` untouched. So we know whether `cas` succeeds or not after the operation by comparing `newval` and `oldval`. If `newval` equals `oldval`, `cas` did succeed. The important characteristic is that the whole `cas` operation works atomically, i.e., without being interrupted in the middle by any other thread. This atomic characteristic is necessary to be used in updating the value of the deque's `top` pointer.

In the `pushBottom` operation, the worker only needs to append a task to the bottom end of the deque and increment the deque's `bottom` pointer accordingly. In the `popTop` operation, the thief worker first checks if there is any task existing in the queue by comparing the deque's `top` and `bottom` pointers. If there is task(s) in deque, it first gets the (pointer to the) task at top of the deque without modifying the top pointer and the deque's content, then it executes a `cas` operation to decrement the `top` pointer. If `cas` succeeds, it means the thief has stolen successfully and it should continue with the stolen task. Otherwise, it means that there had been someone (another thief) getting in the middle and taking that task already, i.e., the thief fails this time, and it should continue with another steal. In the `popBottom` operation, the local worker only needs to pay attention when the task it is going to take is the last task in the deque ($top == bottom$). Because at that indigent state there is likely some thief thread trying to take that task at the same time. In order to avoid this race condition, the local worker uses `cas` operation to watch out for any change in `top` pointer when incrementing its `bottom` pointer. But this watching out is only needed in cases when there would be no task remaining in the deque after the pop.

In ABP work stealing, the deque is physically implemented based on fixed-sized array which has problems of overflow (`top` or `bottom` pointer exceeds the array size). This problem is mitigated by a heuristic named *reset-on-empty*. This heuristic resets `top` and `bottom` to point to the beginning of the array when the deque becomes empty (after a `popBottom` operation). This heuristic helps to make overflow scenarios less frequent but does not eliminate it. Chase and Lev [18] have introduced an improvement to ABP work stealing's deque implementation that can eliminate this overflow problem. The improvement implements work stealing deque using a *dynamic-circular-array*. This kind of array has two features of dynamically-changing size and circular index which solve the overflow problem permanently.

A dynamic-circular-array also has two indexes `top` and `bottom` specifying the positions of the topmost and the bottommost elements of the deque in the physical array. The array is indexed modulo its size, i.e., indexes giving the same remainder when divided by the array size point to the same element of the array. This kind of circular access is implemented as shown in simple get() and put() functions in Fig. 2.4.

```
Object * get( Object * A, int size, int i ) {
  return A[ i % size ];
}
void put( Object * A, int size, int i , Object * o) {
   A[ i % size ] = o;
}
```

Figure 2.4: Circular array using modulo its size to access its elements

With this circular access manner, all slots in the array can be used efficiently no matter what the top's value is, and more importantly top index' values would be never decreased. They are initialized at zero, `pushBottom` and `popBottom` affect (increase/decrease) only bottom index, `popTop` only increases top index, never decreases it. This new nature of top index makes it easier for `cas` operation to compare old values with top index. Because there would never be the same top value stored in top index, we do not need to do other complicated mechanisms such as a flag index to indicate new phases of top index as in ABP deque's implementation.

In `pushBottom` function, if it detects that the deque gets full it will grow the deque dynamically by allocating a new array of bigger size and copy old array's content to the new one. Array size is increased as double the old size each time (two's power). All current elements in the old array are copied to the new array (of course with their circular indexes), and they remain being able to accessed with the same indexes thanks to the circular indexing. Therefore, high-level deque operations do not need to be aware about this change in physical base arrays.

Chapter 3

Task Parallel Programming Models and Runtime Systems

Due to fundamental physical constraints such as power consumption and heat dissipation, the development of computer hardware has changed from increasing clock speed of a single-core CPU to increasing the number of cores integrated in a multicore CPU [20]. There are more and more cores which are integrated in a computer's CPU, from several cores in a commodity PC up to dozens or hundreds of cores in a high performance computing server. Moreover, the emerging Many Integrated Core (MIC) architecture of Intel, which combines many smaller lower-performance cores on the same chip area, has promised a highly parallel era of shared memory computer hardware. This highly parallel hardware would make it harder for programmers to program parallel software using common parallel programming models such as SPMD and native threading libraries (e.g., POSIX Threads (pthreads) [25]) which involve programmers in dealing with low-level details of thread management, task scheduling, load balancing, etc.

Task parallel programming models release programmers from such low-level concerns by shifting these burdens to the runtime systems so that the programmers can concentrate on higher level aspects of the programming. It also promises to make parallel programming accessible for those programmers who are not familiar with low-level system and hardware issues. In task parallel programming, programmers just need to expose parallelism in their programs by creating tasks. These tasks are scheduled to execute in parallel dynamically by the runtime system.

With this high-level task parallelism, programmers can express parallelism at arbitrary places in their program even in recursive functions just by specifying task creation. This kind of programming model is well suited for the expression of nested parallelism in divide-and-conquer algorithms and unstructured parallelism in irregular computations. Therefore, it has been well supported gradually with time since the first language and runtime system dedicated for task parallelism Cilk [13] developed at Massachusetts Institute of Technology (MIT) was introduced in 1994. E.g., OpenMP Task has been added to OpenMP from version 3.0 (released in 2008) [19] [6]; Intel Threading Building Blocks (TBB) [57] which is a C++ template library provides a task parallel scheduler along with higher-level data structures and algorithm templates based on task parallel execution model for users to exploit parallelism in their programs easily; MassiveThreads (2012) [52] and Qthreads (2008) [71] are minimal implementations of the runtime task scheduler for research efforts, they expose a simple pthreads-like API for users to access their task runtime.

Tasks unlike threads are *light-weight threads* that do not support expensive features like per-thread identifiers, per-thread signal vectors, or preemptive multitasking. They can be context-switched in user space (not require getting into kernel space) without any requirement for signals nor saving a full set of registers, hence their context switch is less expensive than the original OS-level thread's (interrupt-based) context switch. Because of that advantage, runtime scheduler can easily hide communication latency by switching tasks that stall when waiting for data.

Task schedulers usually exploit a load balancing mechanism called work stealing to distribute tasks among underlying threads at runtime. Work stealing has been proved as an efficient and effective scheme by theoretical [12] or practical in preliminary Cilk language [13]. In original work stealing, each thread would maintain its own work queue of ready jobs waiting for execution. When a thread runs out of jobs in its queue, it will go stealing job from a randomly chosen victim thread's queue. This "randomly chosen" manner is good at balancing work among all threads globally and flatly. However, nowadays computer architecture is highly hierarchical with cores distributed on multiple sockets (chips) which are connected to each other by some kind of high-performance links and possess their own memory banks. Significantly different access latencies to data on local memory and remote memory has made computation-data placement matter. Therefore, many recent works have focused on techniques for improving data locality of task parallelism and work stealing by exploiting information about underlying memory and cache systems. Mirroring the hierarchical nature of the hardware in the runtime scheduler can be an good approach, e.g., one work queue is shared by all threads on the same sockets, and many other various approaches.

3.1 Multilisp

Multilisp [27] is an extension of the Scheme dialect of the Lisp programming language family. Lisp, whose name is derived from "LISt Processor", is a fully parenthesized prefix notation with a long history. Its best known dialects which are still in use today are Common Lisp, Scheme, Emacs Lisp, etc.

Multilisp introduces two constructs for expressing parallelism: `pcall` and `future`. `pcall` allows concurrency between the evaluation of two or more expressions that are arguments to a function. `future` enables the concurrency between the computation of a value and the use of that value. `future` is a more fundamental construct than `pcall`, i.e., `pcall` can be implemented in terms of `future`s. With these constructs, the programmer can explicitly specify concurrency that might not be found by automated analysis at compile time or run time.

3.2 Mul-T

In Mul-T "the programmer takes on the burden of identifying what can be computed safely in parallel, leaving the decision of exactly how the division will take place to the runtime system". Specifically, a programmer's job is to *expose* parallelism by annotating the program with `future`s without worrying about the task granularity, while the system's job is to *limit* parallelism by deciding when to spawn or inline tasks.

3.3 MIT Cilk

Cilk [13] is an extension of C language that supports task parallelism. Cilk++ [42] developed by Cilk Arts which was a company spinned off from MIT Cilk project in 2006, then acquired by Intel in 2009. Intel has renamed the language to Cilk Plus [1] [61] since 2009.

Frigo et al. [24] improved the Cilk language design and implementation with its 5th version (Cilk-5). Cilk-5 supports two keywords for specifying parallelism and synchronization: `spawn` and `sync`, two keywords for specifying nondeterminism: `inlet` and `abort`. The `cilk` keyword precedes a function definition to indicate that function as a *Cilk procedure*, which will be compiled with the two-clone strategy.

3.4 Intel Cilk Plus

Cilk Plus [32] [42] is a simple language extension to the C and C++ languages for expressing task parallelism (and data parallelism). Only three keywords are added and users are able to specify task parallelism in C/C++ languages. Cilk Plus is famous for its easy-to-learn, intuitive, yet powerful way of making parallel programs on shared memory systems.

The original Cilk language and runtime [13] [24] developed at MIT in the group of Charles E. Leiserson first appeared in 1994. Cilk's targeting market was restricted to high performance computing until 2006 at which it is commercialized by a spinoff company of MIT, Cilk Art, to leverage the emergence of multicore processors in mainstream computing. Cilk Art shipped its first commercial version of Cilk with the name Cilk++ 1.0 in 2008. One year later, Cilk Art was acquired by Intel and it has become a part of Intel until now.

Cilk Plus provides three keywords for users to express task parallelism at arbitrary points in the program. They are `cilk_spawn`, `cilk_sync` and `cilk_for` (Table 3.1). `cilk_spawn` specifies that a function (which becomes a task) can execute in parallel with the remainder of the calling function. `cilk_sync` specifies that all spawned calls (tasks) in the current function must complete before execution continues. `cilk_for` transforms iterations of a for loop to a set of tasks which can be executed in parallel. The two keywords `cilk_spawn` and `cilk_for` just express oppurtunities for parallelism, the actual parallelism that those tasks are actually executed in parallel is not guaranteed and only decided at runtime by the Cilk Plus runtime system which implements a work stealing scheduler. Beside tasking feature allowing users to exploit thread parallelism, Cilk Plus also provides array notations and `#pragma simd` directive for users to make use of vector processing capability inside processors. Within the scope of this survey, I am not going to focus on these vectorization features.

Create task	`cilk_spawn`
Synchronize tasks	`cilk_sync`
Parallel for loop	`cilk_for`

Table 3.1: Additional keywords of Cilk Plus

An example Fibonacci program written in Cilk Plus is shown in Fig. 3.1. Users do not need to declare a thread team creation at the beginning of the program like in OpenMP because Cilk Plus is built directly upon task notion, hiding all underlying threads which are managed automatically by Cilk Plus runtime. In the fib(n) function, the current task creates a new task executing fib(n-1) function call and it executes the fib(n-2) function call by itself, then it waits for the spawned task to finish before summing and returning result.

Cilk Plus needs support from compilers which can regconize its keywords. Following is compiling examples of Cilk Plus by ICC and GCC compilers. Although ICC automatically links to Cilk Plus runtime library at compile time, GCC needs to be specified explicitly linking options to Cilk Plus runtime.

```
ICC: $ icc cilkprogram.cpp −o cilkprogram
GCC: $ g++ cilkprogram.cpp −o cilkprogram −lcilkrts −ldl
```

3.5 OpenMP

OpenMP (Open Multi-Processing) [19] is an application programming interface (API) for C/C++ and Fortran languages, which is used to program multithreaded applications for shared memory multiprocessor systems. OpenMP is standardized by a joint committee of various computer hardware and software vendors so it is well accepted and portable. OpenMP consists of three components:

```
#include <stdio.h>
#include <cilk/cilk.h>

int fib( int n ) {
  if ( n < 2 ) return n;
  int x, y;
  x = cilk_spawn fib( n−1 );
  y = fib( n−2 );
  cilk_sync;
  return x + y;
}

int main() {
  printf ( "fib(10) = %d\n", fib(10) );
  return 0;
}
```

Figure 3.1: An example Fibonacci program written with Cilk Plus

compiler directives, runtime library routines and environment variables. The compiler directives (starting with "`#pragma omp`") are for specifying parallelism in the application's code. Runtime library routines and environment variables are used to control the behavior of OpenMP's runtime system at execution time, e.g., to get or set number of threads running, to get current thread's number, etc.

From its first release in 1997 until before the release of version 3.0 in 2008, OpenMP was all built around threads and structured workloads (e.g., focused on processing large array), lacking the capability to express irregular un-structured parallelism [7]. Two major features of OpenMP, parallel loops and parallel sections, both deal with workloads that are able to be divided equally into sub-workloads. The parallel loop (`#pragma omp for`) bears one constraint which is that the number of iterations of the loop to be parallelized needs to be determined at the entry of the loop's execution and this iteration count cannot be changed during the execution. At runtime, these iterations are divided appropriately (usually in an equal and round-robin manner) among available threads. Besides, OpenMP's parallel sections feature (`#pragma omp sections`) requires users to divide work into consecutive blocks statically at programming time.

Realizing the need for expressing irregular un-structured programs (e.g., nested parallelism) which is a commonly encountered problem, the committee of OpenMP specification has added feature of task parallelism into OpenMP from version 3.0. A task is an independent unit of work whose execution can be deferred to later time and can take place on an arbitrary available thread. The task notion is standardized in OpenMP with three design goals of "simplicity of use, simplicity of specification and consistency with the rest of OpenMP" in mind [7]. Basically, this tasking feature is based on two new directives of `#pragma omp task` and `#pragma omp taskwait`. The first directive is to create a task, the second one is to synchronize created child tasks. Major compiler directives of OpenMP are summarized briefly in Table 3.2.

parallel region	`#pragma omp parallel` {. . . }
single region	`#pragma omp single` {. . . }
parallel for loop	`#pragma omp for` for (. . . ; . . . ; . . .) {. . . }
parallel sections	`#pragma omp sections` { (section regions) }
section region	`#pragma omp section` {. . . }
create task	`#pragma omp task` {. . . }
synchronize tasks	`#pragma omp taskwait`

Table 3.2: Syntax of OpenMP pragmas

An example Fibonacci program written with OpenMP Task is shown in Fig. 3.2. This program uses four kinds of OpenMP directives: parallel region, single region, task creation and task synchronization. When the program starts, at first there is only one master thread executing. This master thread encounters `parallel` directive which demands creating a set of threads corresponding to the number of underlying cores (N). Hence, an additional number of $N - 1$ worker threads (excluding the master thread) get created. These master and worker threads all then continue executing the code block following the `parallel` directive. The `single` directive that follows is used to suppress these threads so that only one thread is allowed to enter the block and execute *printf*, calling *fib(10)*. Other threads would stay idle, waiting for work. Inside fib(10) routine, the proceeding thread creates a task executing fib(9) routine and executes fib(8) itself before synchronizing the fib(9) task then returning result. The fib(9) task would be stolen and executed by one of the idle threads. Inside fib(9) and fib(8) tasks, smaller tasks are continued to be created and stolen by free, waiting-for-jobs threads so that the work are balanced among available threads.

OpenMP needs to be supported by compilers so that its preprocessing directives can be recognized and processed. Most major compilers have already supported OpenMP such as GNU C compiler (GCC), Intel Fortran and C/C++ compiler (ICC). An example compiling command of OpenMP with GCC is shown below, a flag of *-fopenmp* is inserted to tell GCC to process OpenMP directives.

```
$ gcc −fopenmp ompprogram.c −o ompprogram
```

```
#include <stdio.h>
#include <omp.h>

int fib( int n ) {
   if ( n < 2 ) return n;
   int x, y;
#pragma omp task
   { x = fib( n−1 ); }
   y = fib( n−2 );
#pragma omp taskwait
   return x + y;
}

int main() {
#pragma omp parallel
#pragma omp single
   printf ( "fib(10)␣=␣%d\n", fib(10) );
   return 0;
}
```

Figure 3.2: An example Fibonacci program written with OpenMP Task

3.6 Intel Threading Building Blocks (TBB)

Intel Threading Building Blocks (TBB) [57] [34] is a C++ template library for writing parallel program on multicore processors. TBB is developed by Intel and was first released in 2006. Its latest version at the time of this writing is 4.3 update 4 released in March 2015. As being implemented as a library, TBB has an advantage over OpenMP and Cilk Plus that it does not require special supports from languages and compilers, hence it is more portable. Although, TBB can be used with C++ language only.

TBB consists of a large set of high-level data structures and algorithms for concurrent processing that allow programmers to exploit parallelism of various forms quickly and efficiently. Underlying below these high-level templates is the task scheduler that drives the design and implementation of them (Fig. 3.3). In order to access this task scheduler directly, beside a low-level task interface, TBB provides users with a much easier-to-use high-level interface of task group. The use of task group is generally similar to that of OpenMP Task and Cilk Plus with one interface to create tasks and another interface to synchronize tasks.

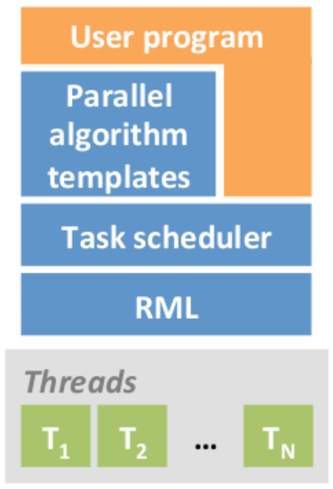


Figure 3.3: TBB layers [46]

An example Fibonacci program written with TBB's `task_group` is shown in Fig. 3.4. In order to use TBB's task group, first an object of `task_group` class need to be declared (line 7). A task is then created by passing its correspondent function (or lambda expression) to the method `run()` of the `task_group` object. All tasks created with the same `task_group` object get synchronized by a call to its `task_group.wait()` method. While we can only synchronize all child tasks at once in OpenMP and Cilk Plus, we can synchronize an arbitrary subset of child tasks in TBB by creating tasks that we want to synchronize at the same time with the same `task_group` object. TBB supports a more flexible tasking model, and fully-nested parallelism compared to OpenMP and Cilk Plus.

```
 1  #include <stdio.h>
 2  #include <tbb/task_group.h>
 3
 4  int fib( int n ) {
 5    if ( n < 2 ) return n;
 6    int x, y;
 7    tbb::task_group tg;
 8    tg.run( [&]{ x = fib( n−1 ); } );
 9    y = fib( n−2 );
10    tg.wait();
11    return x + y;
12  }
13
14  int main() {
15    printf ( "fib(10)␣=␣%d\n", fib(10) );
16    return 0;
17  }
```

Figure 3.4: An example Fibonacci program written with TBB

3.7 MassiveThreads

MassiveThreads [52] [51] is a light-weight tasking library encompassing a pthread-like API and an efficient work-stealing-based task scheduler which runs at runtime and load-balances tasks among available hardware processor cores. MassiveThreads developed by Taura group at the University of Tokyo with the purpose of researching task parallel scheduling algorithms.

3.8 Qthreads

Qthreads [71] is another research-based light-weight thread library developed by Sandia National Laboratories. Qthreads exposes a similar API as MassiveThreads', although its scheduler implementation has some differences in details.

Chapter 4

Performance Analysis and Visualization Tools

4.1 Analyzing Parallel Performance

Tallent et al. [68] categorized parallel execution time of a multithreaded program into 3 kinds of *work*, *parallel idleness* and *parallel overhead*, in which the overhead is time that workers spend on executing runtime system code and idleness is time that workers spend doing nothing. They use a sampling method that interrupts workers regularly after a fixed period of time to record a sample of where workers are working on. They proposed techniques to measure and attribute parallel idleness and parallel overhead back to application-level code based on an additional binary analysis process of the executable to re-construct the program's user-level call path. Their approach has been implemented in the HPCToolkit [4] performance tool of the Rice University. They claim that these two parallel idleness and parallel overhead metrics can help to pinpoints areas in a program's code where concurrency should be increased (to reduce idleness), or decreased (to reduce overhead).

Olivier et al. [54] had taken a step further than Tallent et al. [68] by identifying that the *inflation* in work is in some cases more critical than parallel idleness or parallel overhead factors in task parallelism. They systemize the contributions of the 3 factors of *work inflation*, *idlness* and *overhead* in the performance loss of applications in Barcelona OpenMP Task Suite (BOTS). Because work inflation occurs due to the increased overhead in getting data from remote modules in NUMA architecture, they developed a locality-aware scheduler which placed tasks near their data, and succeeded in mitigating work inflation in two benchmarks *health* and *heat*.

There have been many tools for analyzing parallel performance. The TAU performance system [63] is an open source system that has a powerful automatic instrumentation toolset. Intel VTune Amplifier software [33] uses sampling method and does not need to instrument the executable. These tools focus on the analysis of only one single execution of the application. They can pinpoint the most costly code blocks in the application-level code which consume most of the execution time. To analyze the work inflation factor we need to compare a pair of executions on fewer and more numbers of cores, which these tools do not support.

Liu et al. [43] has built a NUMA profiler for multithreaded programs. It can assess the severity of remote access bottleneck and provide optimization guidance for redistributing data based on memory access patterns of threads. For task parallel programs where tasks are distributed dynamically, however, the solution needs to take into account the structure of the DAG.

The Cilkview Scalability Analyzer [28] describes Cilkview tool which monitors logical parallelism during an instrumented execution of the Cilk++ application on a single processor core, then analyzes logical dependencies between tasks to predict the application's performance on a machine with more cores.

Many other analysis tools focus on only one programming model. Cilkview [28] traces the logical tasks and their dependencies in the serial run of a Cilk++ program and predicts its performance on higher core counts. It can detect problems like insufficient parallelism caused by coarse-grained tasks and show responsible tasks. Cilkprof [62] can measure work and span of specific call sites of choice rather than the whole program like Cilkview. Grain graphs [49] capture and visualize tasks from inside OpenMP scheduler. Their grain visualizations can also highlight problems like work stretch, and low-parallelism intervals. Flow Graph Designer [70] is a tracing and analysis tool specialized for the flow graph interface of TBB. Our Delay Spotter is applicable to many systems from the beginning and easy to be extended with a new one thanks to its simple and portable instrumentation scheme. Delay Spotter is able to compare and contrast differences between systems. We have not seen this ability of highlighting advantages and disadvantages of one system compared with others in previous approaches to the best of our knowledge. We realize this ability by focusing our analysis on the scheduling delays which are artifacts imposed by the systems on the parallel executions.

4.2 Performance Visualizations

Visualization is an highly useful tool in doing analysis. Visual elements can convey structure of the problem at a glance, and they may ignite insights to the solution that numbers and tables merely can hardly reveal. By sticking to the analysis mindset of "overview first, zoom and filter, the details on demand" [64], a visualization tool can support effectively the analysis of complex hierarchical large datasets.

Visualization has been used as an effective tool to deal with various specific performance problems. Knowing that communication cost in massively parallel applications on large distributed systems impacts heavily their performance, the authors in [38] have combined 2D and 3D views to visualize network traffic in order to explain and then optimize the performance of large-scale applications on a supercomputer. *CommGram* [73] invented a new kind of visualization to display network traffic data. It enhances bipartite graph style by replacing thin straight arrows by fat colorful brushy curves to represent data flow between communication nodes vividly.

Vampir Vampir [50] translates a trace file of an MPI program into a variety of graphical visualizations. Its main visualization is a timeline view (Gantt chart) of the execution of the parallel program. It simultaneously provides a statistical view that displays aggregate information of a chosen time interval. It can also provide system activities at a particular point of time. Iwainsky et al. [35] have used Vampir to visulize remote socket traffic on the Intel Nehalem-EX.

Jumpshot Jumpshot [74] is a scalable tool to visualize timelines. Task intervals of all workers written in file in *sslog* log file format can be converted into *slog2* format which can be read and visualized by Jumpshot. Jumpshot is really a scalable tool that can zoom into tiny intervals but it is not that easy and quick for users to perform zooming-in, zooming-out operations. One restriction of Jumpshot is that it can only display up to 10 different categories which have different colors. It means that, for example, the visualization can distinguish up to only 10 different task levels.

Paje Paje [36] provides timeline style visualization of parallel programs executing on multiple nodes each of which contains dynamically running multiple threads. Paje supports *click-back*, *click-forward* interaction semantics which mean that clicking visualization to show source code and clicking source code to show visualization. Paje has several filtering and zooming functionalities to help programmers to cope with large amount of trace information. These filterings give users simplified abstract view of the data (statistical graphs showing aggregate information of a chosen time slice). Users of Paje can also modify mapping between trace information entities and visual elements (arrows, boxes, triangles) which makes the visualization flexible.

Jedule Jedule [29] is a tool to visualize schedules of parallel applications in timeline style. It is built on Java. Users can adjust color style of Jedule's visualization, can zoom in by selecting a rectangular box, can export current view to images. Authors in [54] have used Jedule to visualize a timeline view for analyzing the locality of a scheduling policy.

ThreadScope Wheeler and Thain [72] in their work have demonstrated that visualizing a graph of dependent execution blocks and memory objects can enable identification of synchronization and structural problems. They use existing tracing tools to instrument multithreaded applications, then transform result traces to dot-attributed graphs which are rendered by GraphViz [11]. GraphViz tool is scalable up to only hundreds of nodes and very slow with large graphs of more than a thousand nodes because its algorithm [66] focuses on the aesthetic aspect of graphs rather than rendering speed. And most of all, GraphViz is not interactive.

Aftermath Aftermath [21] is a graphical tool that visualize traces of an OpenStream [58] parallel programs in timeline style. OpenStream is a dataflow, stream programming extension of OpenMP. Although Aftermath is applied in a narrow context of OpenStream (subset of OpenMP), it instead provides an extensive functionalities for filtering displayed data, zooming into details and various interaction features with users. Aftermath is built upon the GTK+ GUI toolkit [60] and Cairo graphics rendering library [59].

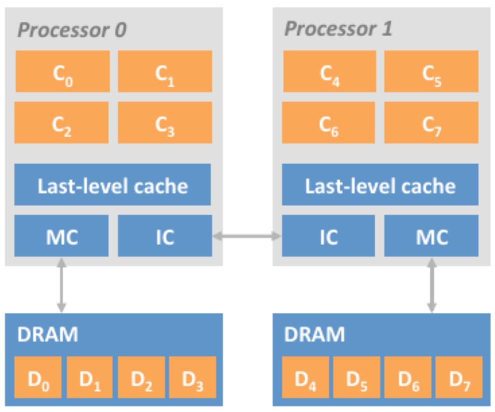
Many existing performance visualization tools such as Intel VTune Amplifier [33], Vampir [50], Jumpshot [74], Jedule [29], Aftermath [21] [58], Extrae-Paraver [44], etc. support the timelines view showing CPU-related events during the duration of the execution. Our tool can show timelines visualization together with the profile of running and ready parallelism, which is more useful for task parallel programs. One can grasp a general view of the parallel execution with this parallelism profile, and quickly identify tasks on timelines who are responsible in intervals with low running parallelism. Moreover, our task-centric DAG visualizations also allow one to zoom into any specific spot on the DAG and relate it back to timelines and parallelism profile. Without them we could not have identified the problems as in our case studies, like OpenMP could not resume the parent task because the task is tied to a busy worker in Alignment, or OpenMP restrains workers to steal work at deep recursions, Qthreads delays the scheduling of recursively created tasks while the worker gets into the last leaf child task in FFT. The timelines views alone would not be enough in these cases.

4.3 Data Locality

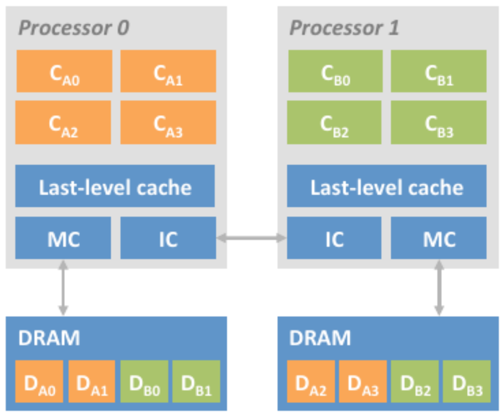
Work-first's execution order which is similar to that of a serial execution is said to be able to maintain the data locality that exists in the serial execution [3]. Besides, with local LIFO access manner and remote FIFO access manner of work stealing deque it ensures that the older tasks are stolen first, the newer tasks are prioritized to be executed locally without being migrated. This scheme leverages the data locality inherent in divide-and-conquer algorithms where a task tends to operate on the same data as its parent and sibling tasks. Newer tasks whose data is still hot in local cache are the first to be scheduled locally and the last in line to be stolen.

That is the data locality favor of a single deque's operations. But it becomes anti-data locality in work stealing's current organization of each deque per-worker thread and randomized victim selection. Existing work stealing approaches seem to be developed based on the assumption of an underlying flat system model in which every processor core has the same computation and memory access capabilities. This assumption might be appropriate at the old time when there were still few cores residing in a node and they are fit onto a single chip. That symmetric multiprocessing (SMP) model was really actually symmetric. However, things have changed with time, there are more and more cores integrated in a single machine now (it can be up to hundreds with Xeon Phi architecture), and they are no longer fit on a single chip, but they are usually spread out on multiple chips each of which contains several cores. These chips are connected by some special high speed interconnect channel, and arranged based on some best-effort-based optimized layout. The memory and cache systems has become more hierarchical and (implicitly) distributed too. Each chip is usually equipped with a shared last level cache for its cores, and is attached with a memory bank. This kind of system is usually referred to as NUMA (non-uniform memory architecture).

The system is still symmetric from the view of software, all inter-chip communication and accesses to remote memory banks are transparent to users. They still acknowledge as if there is only one unified memory in the system. However, there is still a fact that the accesses to remote memory banks are much slower than to local memory bank. For example, on an Intel Nehalem machine, local accesses to last level cache (L3) and memory (DRAM) take 38 and 190 cycles, whereas remote accesses take 186 and 310 cycles. They slow down with factors of 4.9x and 1.6x respectively [46].



(a) optimized


Processor 0
C_{A0} C_{A1} C_{A2} C_{A3}
Last-level cache
MC
IC
Processor 1
C_{B0} C_{B1} C_{B2} C_{B3}
Last-level cache
IC
MC
DRAM
D_{A0} D_{A1} D_{B0} D_{B1}
DRAM
D_{A2} D_{A3} D_{B2} D_{B3}

(b) not optimized

Figure 4.1: Computation and data placement when optimized for data locality and not optimized [46]

Because of this large gap in latency and also bandwidth, it cannot be avoided to admit that executing tasks on cores of the chip that holds their data in local memory will result in a considerable performance gain (Fig. 4.1a, Fig. 4.1b). Therefore, researches about NUMA-aware work stealing schemes have emerged and become more critical to performance on nowadays hierarchical "assymmetric" computer architecture.

In current work stealing, a victim worker is chosen uniformly at random without considering about its chip and NUMA domain. Olivier et al. [55] have proposed a shared deque for each chip, local threads still operate in LIFO manner and remote threads still operate in FIFO manner when stealing (Fig. 4.2). When a thread goes stealing, it will steal a bunch of jobs on behalf of the threads on the chip, and it restricts that there is only one thread goes stealing at a time. Although its current implementation still uses lock to synchronize accesses to the shared deque, its scalability result is encouraging that several benchmarks reach speedup of 90x-150x on 196 cores. On data-intensive sort and health benchmarks they have observed a sharp increase in computation time due to increased load latencies compared to sequential execution.

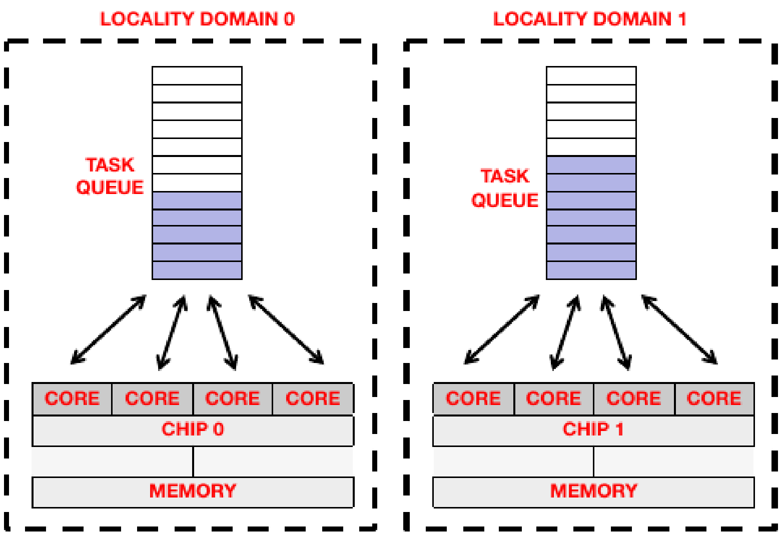
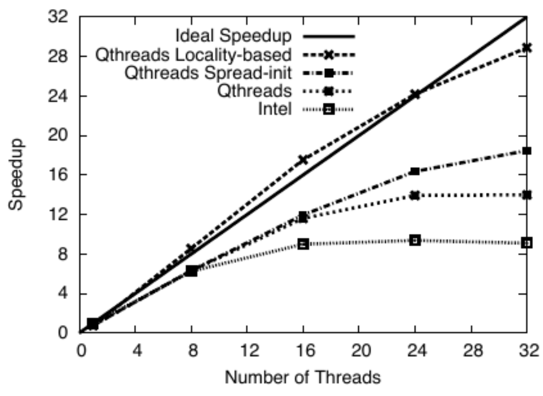
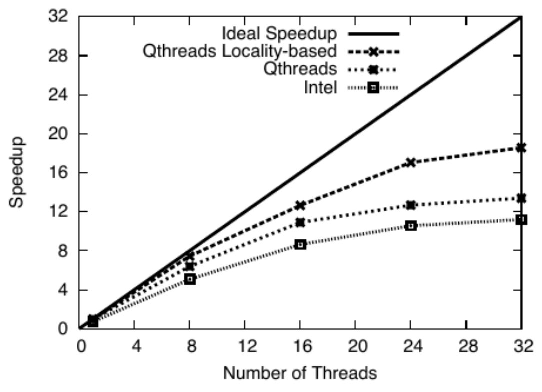


Figure 4.2: Each locality domain per-chip has one shared deque [46]

Olivier et al. in [54] applied the similar locality framework as above to benchmarks of Health and Heat. simulations and they yielded promising results with great increase in speedup (Fig. 4.3a, Fig. 4.3b). Another interesting metric is QPI (Quick Path Interconnect - Intel's interconnect technique between sockets) traffic, i.e, the amount of data transferred between sockets during execution. Fig. 4.4 shows the measured amounts of data in gigabytes.



(a) Health benchmark



(b) Health benchmark

Figure 4.3: Speedups of Health and Heat benchmarks with the proposed Qthreads locality-based scheduler [46]

	Seq. (gcc)	Intel	Qthreads	Q Locality
Health	0.067	34	26	1.0
Heat	0.077	31	31	0.34

Figure 4.4: Data transferred (GB) over interconnects (QPI) between chips [46]

Majo et al. [46] proposed an programming library TBB-NUMA based on Intel TBB which enables portable and composable NUMA-aware programming. TBB-NUMA provides a unified interface to the runtime system and allows programmers to define thread affinity, and do explicit memory system-aware resource management.

Chapter 5

Analyzing Performance Differences based on Scheduling Delays

Modern task parallel programming models provide sophisticated runtime task schedulers for handling the scheduling of logical tasks on a large and varying number of hardware parallel resources at runtime. The performance of these programming models increasingly rely on how fast their runtime schedulers do their job. The more delay a scheduler incurs in matching a ready task to a free processor core at any point in time, the more impact it causes to the program's parallel execution. We have developed a tool that is able to detect these delayed intervals caused by the scheduler in a parallel execution, and spot them specifically on two kinds of visualizations: the logical task graph captured at runtime (DAG visualization) and time-series visualizations of threads (timelines). By further analyzing positions of these delays on those visualizations the tool could identify possible scheduling issues in the scheduler that causes these delays, yielding improvement insights for the development of task parallel programming models. From an application programmer's perspective, our tool is useful by being able to contrast differences of various task parallel models executing the same program, helping users choose the right model for their application. We demonstrate that usefulness by using the tool to analyze 10 applications in BOTS benchmark suite in our case studies.

5.1 Background

Computer systems have become increasingly parallel with more nodes, more cores, and more threads. This has made writing parallel applications for those systems more difficult based on traditional SPMD programming models such as MPI and POSIX Threads. In these models, programmers have to schedule work and balance load manually on a large number of processor cores, which is a tedious job for programmers. Modern parallel programming models have been developed to shift these burdens onto the runtime systems, freeing programmers from them. Programmers can be unaware of underlying hardware resources such as how many cores or nodes available, and can concentrate better on algorithmic and creative aspects in their application developments. Task parallel programming models, which are provided by various languages and libraries such as OpenMP Tasks [56], MIT Cilk [24], Intel Cilk Plus [61], Intel Threading Building Blocks (TBB) [57], Qthreads [71], and MassiveThreads [51] [52], are examples of this approach. In these models, programmers are encouraged to provide ample logical parallelism by creating a large number of tasks which are independent work units that can be executed in parallel. The runtime systems are responsible for mapping these logical tasks to available hardware resources dynamically and automatically at runtime.

As the hardware parallelism grows in scale and hierarchy, it is more challenging for runtime schedulers to keep task scheduling efficient. A larger hardware parallelism tends to impose more delays in their scheduling. A ready task needs to wait longer before being assigned to a free worker, not only because the thief needs more time to find a worker holding a task to steal, but also because a scheduler may be employing sophisticated scheduling policies which may prolong execution in the scheduler's code. In general, a sophisticated scheduler may incur larger delays, but as a trade-off it is more likely to achieve better task executions (shorter runtimes) through other aspects like better memory subsystem usages and better data localities. In contrast, a greedy scheduler which is swift in matching a ready task with a free worker may miss or overlook those opportunities. There are various choices in scheduling parallel tasks, and each system has made its own choices in its design and implementation. Analyzing specific causes of scheduler-caused delay, and precisely quantifying their impacts on the parallel execution are crucial for improving performance of task parallel programming models.

At any point in the execution of a parallel program, a worker is either working on the application code or not working on the application code. An interval during which a worker is working on the application code is referred to as a *work*. An interval during which a worker is not working on the application code, but rather the system code like scheduling, or just waiting in idle due to lack of work, can be further classified into two categories of either *delay* or *no-work*. The two categories are distinguished based on the ready tasks available at the time; if the number of ready tasks was enough to feed the worker, but it has not successfully acquired any task to work on yet, the interval is a delay. On the other hand, if there was no task available during the interval, or the number of available tasks were not enough to feed the worker, the interval is a no-work. Regarding all executing workers, the cumulative execution time of the execution (= elapsed time × workers) can be divided into three components of *work*, *delay*, and *no-work* (*cumul. exe. time* = $work + delay +$ *no-work*) which are the sums of all intervals of the corresponding types across all workers.

Work is the useful computation, whereas delay and no-work are inefficiencies contributing to the scalability loss of the parallel execution. A delay can always be blamed on the system for not doing its job well enough, whereas a no-work can be blamed both on either the scheduler or the application. We use the *ready path analysis* in order to separate the part of no-work that is caused by the scheduler (*no-work-sched*) and the other part caused by the insufficient parallelism issue in the application (*no-work-app*). Now we can refer to the sum of delay and no-work-sched to quantify the impact of the scheduler on a parallel execution.

5.2 Breakdown of Cumulative Execution Time based on Scheduling Delays

5.2.1 Performance loss in parallel execution

Consider a short time interval of length Δt during an execution. We assume it is so short that p, q, as well as the number of tasks are constant within the interval. Let p be the number of workers executing application code and q the number of remaining workers. We count the time spent by those p workers as *work*. That is, work during this interval is $p\Delta t$. We count the time spent by the other q workers as either *delay* or *no-work*, depending on the number of *ready tasks* (tasks ready to execute but not being executed) in this interval. Specifically, of the q workers not executing the application code, up to the number of ready tasks (r) is counted as *delaying* the execution of application, as they *could* have executed those ready tasks. That is, $\min(q, r)\Delta t$ is counted as *delay* and the remaining $\max(0, q - r)\Delta t$ counted as *no-work*.

Accumulating them over time, we can define work, delay, and no-work components of the entire cumulative execution time across all workers in an obvious manner (Fig. 5.1). Both delay and no-work represent wasted time (time not spent on application code), but the distinction is important; the former represents the time the scheduler could have reduced by dispatching ready tasks to available workers more quickly, whereas the latter is the time wasted due to lack of tasks. The sum of work, delay, and no-work is always the elapsed time × the number of workers.

In order to calculate these three components, we need the time-series information of running

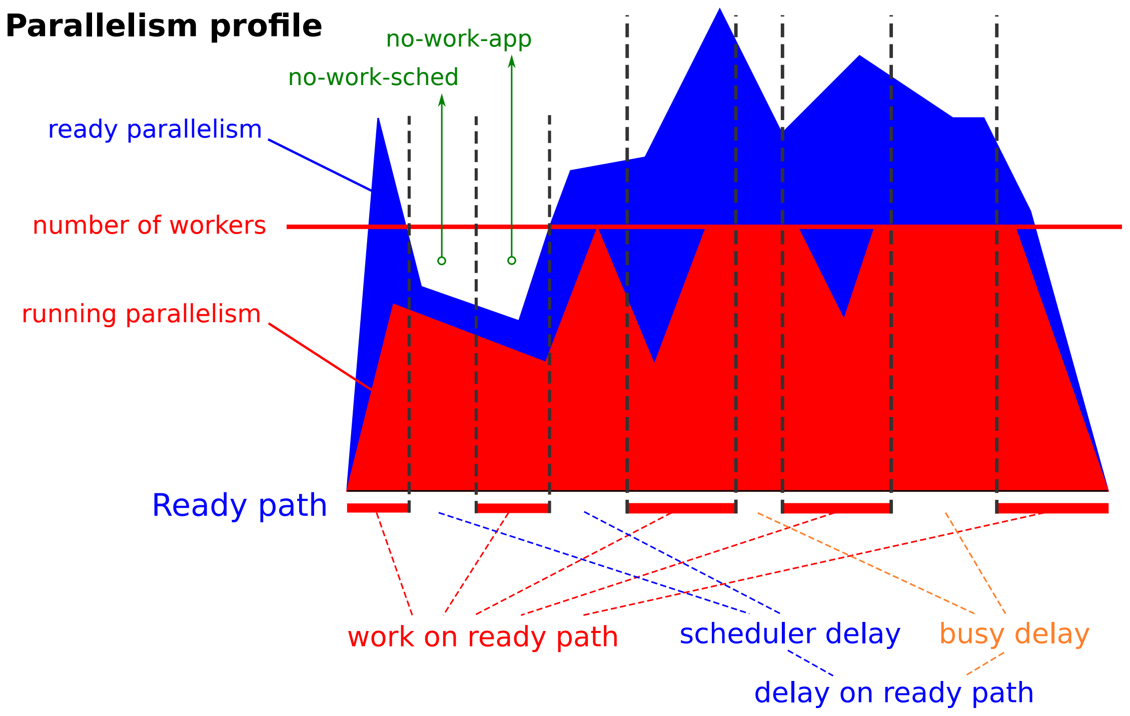


Figure 5.1: Parallelism profile and the ready path. Work, delay, and no-work components of the cumulative execution time are respectively red, blue, and white (empty) area below the red line of the number of workers. The ready path (one among numerous paths existing in a DAG) is divided into work, busy delay, and scheduler delay.

and ready parallelism during the program execution. Running parallelism at a point in time is the number of workers executing the application code at that point; ready parallelism is the number of ready tasks. A time series of running and ready parallelism, with the latter stacked on the former, is called parallelism profile (Fig. 5.1). In Fig. 5.1, red part represents running parallelism and blue part represents ready parallelism. The flat red line above the red area denotes the number of participating workers, which is also the maximum possible running parallelism. From Fig. 5.1, it can be simply understood that the work component of the cumulative execution time is the red area, delay is the blue area *below the red line*, and no-work is the white (empty) area *below the red line*. Hence, the cumulative execution time which is the sum of work, delay, and no-work, is the area of the rectangle confined by the x-axis ($y = 0$), the y-axis ($x = 0$), the red line ($y = \#workers$), and the vertical line at the end time point of the execution ($x = elapsed\_time$).

An application has a perfect (linear) speedup when parallel executions with different numbers of workers have the same work, zero delay, zero no-work. However, in reality a parallel execution on higher core counts tends to incur non-zero delay, no-work; and even increasing work. The amount of delay and no-work is determined by many factors such as the scheduling overhead inside the runtime system, the amount of serial and low-parallelism sections remaining in the parallel application. Work on higher core counts tends to increase due to such causes as more contentions among workers accessing the shared variables, more remote memory accesses, or less effective use of cache and memory subsystem. We use the work in a serial execution, which is usually the smallest, as the baseline performance. Compared with that serial work, the surplus amount of a parallel work is referred to as *work stretch*. Therefore, the *performance loss* (or scalability loss) of a parallel execution ($perf.\ loss = cumul.\ exe.\ time - work_{serial}$) includes work stretch ($= work_{parallel} - work_{serial}$), and delay and no-work components of that parallel execution. Although all three performance loss factors (work stretch, delay, no-work) are no doubt important, within the scope of this paper (Delay Spotter) we focus our analysis on delay and no-work. As work stretch is caused mainly by the bottlenecks in the memory hierarchy, it is necessary to take into account memory-related metrics (e.g., cache misses, remote accesses) to analyze it.

5.2.2 Ready path analysis

While the work as defined above represents a useful computation which executes application code, delay and no-work represent inefficiencies imposed by the runtime system, the application, or other factors of the execution environment. A large delay is likely to be an artifact of an implementation of the runtime system; as there are free workers that should be able to pick up ready tasks of that point, a better runtime system could have matched them up sooner. In contrast, no-work represents a lack of tasks (parallelism), which can generally be attributable to the application. It must be noted, however, that no-work can as well be a consequence of a delay caused by the runtime system. 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. These no-works should blame the scheduler, for delaying the crucial master task whose progress would have made more tasks available for execution.

It is thus important to quantify how much of the observed no-work is caused by the scheduler and how much is caused by the application. To this end, we divide the no-work component into *no-work-sched* and *no-work-app* by looking into the structure of the *task graph* of the execution, which is a graph representing tasks and their dependencies. A *critical path* on the task graph is a serial chain of dependent tasks from the start of the entire computation to the end. Of many such critical paths, our analysis is focusing on the particular path along which *there is always a task running or ready*. We call it *the ready path*. It is easy to see that there is always such a path; from the end of the computation, we trace the task graph backwards, choosing the last finished one when a node has multiple predecessors. Our tool analyzes how the computation progressed along this path, and classifies the entire execution time into the following three parts: *work*, during which a task on the ready path was running; *busy delay*, during which no task on the ready path was running but all workers were busy working on other tasks (not on the path); and *scheduler delay*, during which no task on the ready path was running and there is at least one free worker. An example of ready path is shown at the bottom in Fig. 5.1. Work on the ready path is a part of the work component of the cumulative execution time and scheduler delay is a part of the delay component, but busy delay is not a part of any component; it simply represents an interval in which there is no progress along the ready path, simply because all the workers are busy on other tasks. In the parallelism profile, you can consider it being in the ready parallelism (blue area) *above* the red line denoting the number of workers. A large busy delay does not indicate an issue of the runtime system but just an ample logical parallelism in the application.

No-work that happened in a scheduler delay interval is, at least partially, the runtime system's fault. Had the runtime system shorten the interval, the no-work would have been smaller. We therefore count them as no-work-sched, indicating it may indicate a runtime system's problem. The other part of no-work (i.e., no-work that happened when a task was running on the ready path) is counted as no-work-app, indicating an insufficient parallelism issue caused by the application (Fig. 5.1). Based on our empirical experience, no-work-app is usually an inherent metric of an application; it remains more or less identical across different executions of the same application by different systems.

5.3 Related Work

In contrast to Tallent et al. [68] [4], we classify no-work time of workers into delay and no-work based on the availability of logical ready tasks, so as to judge if an idleness is mainly caused by the scheduler or the application. Our classification is useful for task parallel programming models in which not all idleness are attributable to the lack of parallelism in the application. We are able to do this thanks to computation DAG traces.

Chapter 6

Recording and Visualizing Computation DAG Traces

In task-based parallel programming, programmers can expose logical parallelism of their programs by creating fine-grained tasks at arbitrary places in their code. All other burdens in the parallel execution of these tasks such as thread management, task scheduling, and load balancing are handled automatically by runtime systems. This kind of parallel programming model has been conceived as a promising paradigm that brings intricate parallel programming techniques to a larger audience of programmers because of its high programmability. There have been many languages (e.g., OpenMP, Cilk Plus) and libraries (e.g, Intel TBB, Qthreads, MassiveThreads) supporting task parallelism. However, the nondeterministic nature of task parallel execution which hides runtime scheduling mechanisms from programmers has made it difficult for programmers to understand the cause of suboptimal performance of their programs. As an effort to tackle this problem, and also to clarify differences between task parallel runtime systems, we have developed a toolset that captures and visualizes the trace of an execution of a task parallel program in the form of a directed acyclic graph (DAG). A computation DAG of a task parallel program's run is extracted automatically by our lightweight portable wrapper around all five systems which incurs no intervention into the target systems' code. The DAG is stored in a file and then visualized to analyze performance. We leverage the hierarchical structure of the DAG to enhance the DAG file format and DAG visualization, and make them manageable even with a huge DAG of arbitrarily large numbers of nodes. This DAG visualization provides a task-centric view of the program, which is different from other popular visualizations such as thread-centric timeline visualization and code-centric hotspots analysis. Besides, DAGViz also provides an additional timeline visualization which is constructed by individual nodes of the DAG, and is useful in coordinating user attention to low-parallelism areas on the DAG. We demonstrate usefulness of our DAG visualizations in some case studies. We expect to build other kinds of effective visualizations based on this computation DAG in future work, and make DAGViz an effective tool supporting the process of analyzing task parallel performance and developing scheduling algorithms for task parallel runtime schedulers.

6.1 Background

Due to fundamental physical constraints such as power consumption and heat dissipation, the development of computer hardware has changed from increasing clock speed of a single-core CPU to integrating increasingly more cores in a multi-core CPU [20]. Recently emerging architectures, such as Intel's Many Integrated Core (MIC) which combines many smaller lower-performance cores on the same chip area, may potentially lead to a highly parallel era of shared-memory computer hardware. This highly parallel hardware will make it harder for programmers to program parallel software using common parallel programming models such as SPMD (e.g., MPI) and native threading libraries (e.g., POSIX Threads [25]) which involve programmers in dealing with low-level details of thread management, task scheduling, load balancing, etc.

Task parallel programming models release programmers from such low-level concerns by shifting these burdens to the runtime systems. In task parallel programming, programmers just need to expose logical parallelism in their programs by creating fine-grained tasks, each of which is a work unit that can be executed in parallel with the rest, at arbitrary places in their code (including recursion). These tasks are scheduled to execute in parallel dynamically by the runtime system. As a result, programmers can concentrate better on the algorithmic aspect of the programming. However, this automation and nondeterminism of task parallel models also removes a great deal of performance out of the programmers' control. The same task parallel program executed by different runtime systems could possibly present significantly different performance. And programmers often lack clues to understand why their programs perform badly.

Common analysis methods such as hotspots analysis and timeline visualization are not sufficient for task parallel programs. Hotspots analysis which shows functions that consume the most CPU time is useful in analyzing sequential execution but fails to pinpoint concurrency bottlenecks in parallel execution. Timeline visualization (a.k.a. Gantt chart) which displays thread activities in the course of the execution is thread-centric and not sufficient for task parallel programs which have dynamic scheduling characteristics and nondeterminism in where tasks are executed. Comparing runs of the same task parallel program is more consistent when we compare them based on their common logical task structure. For that reason, our approach is to measure and extract the computation directed acyclic graph (computation DAG) from a task parallel execution, which records relevant runtime behaviors based on the program's logical task structure (DAG), and visualize it for the performance analysis purpose. In our toolset, the measurement part (DAG Recorder) extracts the DAG during the execution and stores it in a file, while the visualization part (DAGViz) visualizes the DAG and provides visual supports for analyzing performance.

We define a generic task parallel computation model that DAG Recorder can extract a DAG from. This model basically includes only two task parallel primitives of task creation and task synchronization. We build a simple macro wrapper that translates these generic primitives to their equivalents in five separate systems that are currently supported: OpenMP [19], Cilk Plus [61], Intel TBB [57], Qthreads [71], and MassiveThreads [52] [51]. The details of this generic model will be discussed in Section 2. Programmers write task parallelism using our generic model and their code can get translated automatically to these five systems. We extract the DAG by instrumenting this wrapper to invoke DAG Recorder's measure code at appropriate positions These instrumentations are done automatically in the wrapper, requiring neither more work from users nor any intervention into supported parallel runtimes. Hence, although our profiling method requires users to rewrite their code using our generic primitives and re-compile it with our wrapper and DAG-recording libraries, the rewriting work is kept to a minimum that is just replacing primitives and what the users gain is that their code can run with five (and more) different systems that our wrapper library supports, and their code can also be profiled/traced seamlessly by our tools.

In the DAG, nodes represent sequential computations and edges represent dependencies between nodes. Nodes are grouped hierarchically such that a collective node contains in it a subgraph of other collective nodes and leaf nodes which contain no subgraph. Thus, the initially only collective node representing the whole application can get expanded step by step into subgraphs of increasing depths and finally become the full graph of only leaf nodes. Higher-level collective nodes hold aggregate performance information of their inner subgraphs while leaf nodes hold the performance information of their corresponding sequential code segments. We leverage this hierarchical structure of DAG in its storage file format and visualization techniques to make our tools manageable even with a huge DAG of arbitrarily large numbers of nodes (Section 3). By loading and displaying DAG with on-demand levels of details, we can avoid loading the whole big DAG file into memory at once, but need to load only a fraction of DAG file corresponding to the visible part of the DAG on screen.

With DAG visualization we are able to make all nodes of a DAG visual and interactive on the screen. While important, this by itself is not enough since a DAG with up to thousands of nodes is already too large for users to comprehend. Therefore, supportive statistical analyses and other kinds of general visualizations are needed to direct users to trouble areas in a huge DAG. Our tool currently provides an additional timeline visualization which is constructed by individual nodes of the DAG, and is useful in coordinating user attention to low-parallelism areas on the DAG (Section 4). We demonstrate usefulness of these visualizations in two case studies of Sort and SparseLU programs (Section 5). We are working on other kinds of useful visualizations based on computation DAG, among other things, with the prospect of making our toolset an effective platform supporting the process of analyzing task parallel performance and developing scheduling algorithms for task parallel runtime schedulers.

6.2 tpswitch

	tbb, mth, qth	omp	clkp	clk
mk_task_group	mtbb::task_group	Ø	n_children = 0	Ø
create_task	create	create	create	create
wait_tasks	wait	wait	wait	wait
pragma_omp_parallel_single	Ø	omp parallel single	Ø	Ø
create_task_and_wait	call→wait	create→wait	call→wait	create→wait
spawn	Ø	Ø	cilk_spawn	spawn
cilk_begin	Ø	Ø	dr_start_cilk_proc()	dr_start_cilk_proc()
cilk_return	Ø	Ø	dr_end_task()→return	dr_end_task()→return
mit_spawn	Ø	Ø	Ø	spawn
cilk	Ø	Ø	Ø	cilk
call_task	call	call	call	create→wait

omp:
(1) wrap root task with pragma_omp_parallel_single in order
    for omp to initialize the worker thread team
(2) invoke the last task spawn before synchronization with
    create_task_and_wait instead of directly calling it,
    e.g., {f(); wait_tasks;} → create_task_and_wait(f());

clkp:
(3) use spawn keyword in create_task, e.g., create_task(spawn f());
(4) insert cilk_begin/cilk_return in functions that are spawned

clk:
(5) use mit_spawn keyword in create_task_and_wait & call_task,
    e.g., create_task_and_wait(mit_spawn f()); call_task(mit_spawn f());
(6) insert cilk/cilk_begin/cilk_return in functions that are spawned and
    functions that spawn
(7) always spawn or invoke a cilk function by call_task instead of
    directly calling it, e.g., f(); → call_task(mit_spawn f());
(8) detach the lambda closure from leaf function of pfor and make it a
    normal function prefixed with the cilk keyword

(*1) include header: #include <tpswitch/tpswitch.h>
(*2) initialize runtimes: tp_init();

Figure 6.1: tpswitch API

serial

```
void f() {


  f();
  f();


}

void g() {

  for (int i = first; i < last; i += step) {
    /* some computation */
  }






}

int main_task() {


  f();
  g();

  return 1;
}

int main() {


  main_task();

  return 1;
}
```

tbb, mth, qth

```
#include <tpswitch/tpswitch.h>

void f() {

  mk_task_group;
  create_task( f() );
  f();
  wait_tasks;

}

void g() {

  pfor(first, last, step, grainsize,
        [&] (int from, int to) {

          for (int i = from; i < to; i += step) {
            /* some computation */
          }

        });

}

int main_task() {

  mk_task_group;
  create_task( f() );
  g();
  wait_tasks;
  return 1;
}

int main() {
  tp_init();

  main_task();

  return 1;
}
```

tbb, mth, qth, omp

```
#include <tpswitch/tpswitch.h>

void f() {

  mk_task_group;
  create_task( f() );
  create_task_and_wait( f() );


}

void g() {

  pfor(first, last, step, grainsize,
        [&] (int from, int to) {

          for (int i = from; i < to; i += step) {
            /* some computation */
          }

        });

}

int main_task() {

  mk_task_group;
  create_task(  f() );
  create_task( g() );
  wait_tasks;
  return 1;
}

int main() {
  tp_init();
  pragma_omp_parallel_single(nowait, {
    main_task();
  });
  return 1;
}
```

tbb, mth, qth, omp, clkp

```
#include <tpswitch/tpswitch.h>

void f() {
  cilk_begin;
  mk_task_group;
  create_task( spawn f() );
  create_task_and_wait( f()  );

  cilk_void_return;
}

void g() {
  cilk_begin;
  pfor(first, last, step, grainsize,
        [&] (int from, int to) {
          cilk_begin;
          for (int i = from; i < to; i += step) {
            /* some computation */
          }
          cilk_void_return;
        });
  cilk_void_return;
}

int main_task() {

  mk_task_group;
  create_task( spawn  f() );
  create_task( spawn g() );
  wait_tasks;
  return 1;
}

int main() {
  tp_init();
  pragma_omp_parallel_single(nowait, {
    main_task();
  });
  return 1;
}
```

tbb, mth, qth, omp, clkp, clk

```
#include <tpswitch/tpswitch.h>

cilk void f() {
  cilk_begin;
  mk_task_group;
  create_task( spawn f() );
  create_task_and_wait( mit_spawn f() );

  cilk_void_return;
}

cilk void g() {
  cilk_begin;
  call_task( mit_spawn pfor(first, last, step, grainsize,
                            pfor_leaf_func) );

                              cilk void pfor_leaf_func(int from, int to) {
                                cilk_begin;
                                for (int i = from; i < to; i += step) {
                                  /* some computation */
                                }
  cilk_void_return;             cilk_void_return;
}                             }

cilk int main_task() {
  cilk_begin;
  mk_task_group;
  create_task( spawn  f() );
  create_task( spawn g() );
  wait_tasks;
  cilk_return(1);
}

cilk int main() {
  tp_init();
  pragma_omp_parallel_single(nowait, {
     call_task( mit_spawn main_task() );
  });
  return 1;
}
```

Figure 6.2: tpswitch exmples

6.3 Computation DAG

6.3.1 Computation model

In this section, we describe the generic task parallel model that our toolset can extract a computation DAG from, and how other models get translated into our generic one. In our generic model, a program starts as a single task performing its main function. A task can execute ordinary user computation, which does not change the program's parallelism, and additionally other task parallel primitives, which can change the program's parallelism. These primitives are following three semantics:
CreateTask : The current task creates a new child task.
WaitTasks : The current task waits for all tasks in current *section*, explained below, to finish. This primitive also terminates the current section.
MakeSection : This primitive is used to mark the creation of a section inside a task or another section. A section is defined as a synchronization scope which is ended by a WaitTasks primitive and all tasks created inside it get synchronized all together by that WaitTasks. The purpose of section notion is to support a task that waits for a subset of its children. Our generic model supports sections that are either nested or disjoint, but must not intersect.

Task parallel primitives of OpenMP and Cilk Plus models can be translated to our model straightforwardly. The *task* and *taskwait* pragmas in OpenMP are replaced by CreateTask and WaitTasks respectively. The *cilk_spawn* and *cilk_sync* in Cilk Plus are also replaced by CreateTask and WaitTasks respectively. In addition, however, a *task* pragma and a *spawn* operation perform an additional MakeSection operation if the current task has no open section.

Intel TBB model is more flexible than our generic one. The section notion is represented by *task_group* class in Intel TBB. A task is created by calling *run* method of a *task_group* object, and a call to a *task_group* object's *wait* method would synchronize all tasks created by that object's *run* method. One can choose an arbitrary subset of children of a task to synchronize in Intel TBB by creating these children with the same *task_group* object, whereas our generic model does not allow intersected task subsets, and a new section is opened only when the previous section has been closed. Except this restriction, Intel TBB code can be translated into our model by replacing *task_group.run* with CreateTask, *task_group.wait* with WaitTasks, and *task_group* object's declaration with MakeSection.

Qthreads and MassiveThreads are both lightweight thread libraries that expose a POSIX Threads-like interface: one function call to create a task and one function call to synchronize a task of choice. They are as flexible as Intel TBB and translating them to our generic model is imposed with the same restriction.

We have built a lightweight macro wrapper that translates code written with our generic model to these five systems automatically. Hence, by writing code once users can get five separate executables for five systems. Beside these five systems, our toolset can be extended easily to support any other task parallel system that can conform to our generic model.

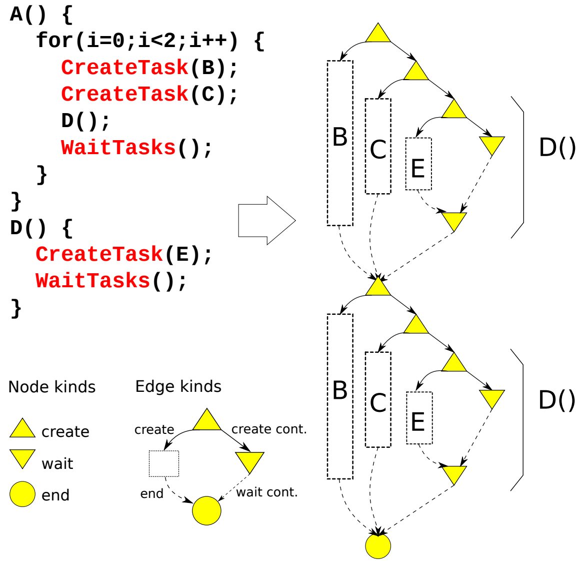
6.3.2 DAG structure

We instrument measure code in the macro wrapper implicitly (requiring no work from users) so that DAG Recorder can get invoked at appropriate positions to record the DAG. Specifically, we instrument at following six positions: the beginning and the end of CreateTask, the beginning and the end of WaitTasks, right before and right after invoking the child task in CreateTask. These instrumentations are put as near the program code as possible with the purpose of capturing the transitions out of the program code and back into the program code. As a consequence, work of the program (i.e., total execution time on all workers of the program code) is broken down into sequential intervals each of which corresponds to a seamless code segment containing no task parallel primitive in the program code and executes uninterruptedly on one worker (core). Although such a sequential interval always happens entirely on a single worker, two consecutive ones separated by a task parallel primitive may take place on two different workers. This is because the execution control is always given back to the runtime system at task parallel primitives where a task migration, among other runtime mechanisms, may happen and change the worker that executes the next interval.

An execution **interval** is modeled as a **node** in the execution's computation DAG. A node (interval) starts either by the first instruction of a task or the instruction immediately following CreateTask or WaitTasks, and it ends either by the last instruction of a task or an instruction immediately before CreateTask or WaitTasks. We classify nodes into three kinds by the ways how they end. A node ends by calling CreateTask primitive is of `create` kind, ends by calling WaitTasks primitive is of `wait` kind, and ends by the last instruction of a task is of `end` kind.

An **edge** in the DAG represents the **dependency** between two nodes that it connects. In other words, an edge is one reflection of a task parallel primitive in the program's execution. There are three kinds of dependencies that an edge can represent: *creation*, *continuation*, and *synchronization*. A node ended by a CreateTask primitive has a creation dependency with the first node of the new task. Two nodes of two contiguous code segments in the program separated by a task parallel primitive have continuation dependency. This continuation dependency can be divided further into create cont. and wait cont. based on the task parallel primitive intermediating the two code segments. The last node of a task has synchronization dependency with the node of the code segment following the WaitTasks primitive that synchronizes that task.

The recursive task creation and nested synchronization scope in the program code are reflected on the DAG by collective nodes of two kinds: `task` and `section` which contain in them subgraphs of leaf nodes (`create`, `wait`, `end`) and other nested collective nodes. A node of kind `task` corresponds to a task in the program code, it can contain zero, one or more `section` nodes before ending by an `end` node. The `section` node kind corresponds to the section notion in our generic model. A `section` node contains one or more `create` nodes along with `task` nodes that these `create` nodes spawn, and zero or more nested `section` nodes, before ending by a `wait` node. All those child `task` nodes of the `section` are synchronized by its end `wait` node and are connected to the successor node of the `section` on the DAG by synchronization edges. The `section`'s end `wait` node is also connected to the successor node but by a wait cont. edge. Fig. 6.3 shows an example task parallel program and an illustration of its corresponding DAG.



```
A() {
  for(i=0;i<2;i++) {
    CreateTask(B);
    CreateTask(C);
    D();
    WaitTasks();
  }
}
D() {
  CreateTask(E);
  WaitTasks();
}
```

Figure 6.3: An example task parallel program and its DAG. The whole execution is originally the only `task` node which is expanded into two `sections` and one `end`. The two `sections` are further expanded into two similar inner topologies as they are two iterations of the same *for* loop.

At measurement points, beside code position (file name, line number) DAG Recorder also records time and current worker (core) so that we can know when and where a node starts and ends. Each node v in the DAG is augmented with information such as start time ($v.start$), end time ($v.end$), the worker ($v.worker$) on which the node was executed, the start and end locations of the corresponding code segment. In case of collective nodes, DAG Recorder additionally stores aggregate information about their inner subgraphs. Two important items of aggregate information are the total work,

$$total\_work(u) = \sum_{v \in u} (v.\mathtt{end} - v.\mathtt{start})$$

and the critical path length of the subgraph. For any subgraph that was executed wholly on a single worker (i.e., there is no work stealing or task migration inside it), DAG Recorder can abolish the subgraph, retain only the collective node (without its inner topology) and its aggregate information. This automatic collapsing technique is optionally conducted on-the-fly during the measurement, and significantly useful in making DAG Recorder scalable because the size of the computation DAG now does not scale with the number of task creations anymore but with the number of task migrations (i.e., work stealing).

6.4 DAG Recorder

6.5 DAGViz

Different from code-centric hotspots analysis and thread-centric timeline visualization, DAG visualization provides a task-centric view of the execution which is the logical task structure of the program. This logical task structure is more familiar from the programmer's perspective, and consistent regardless of runtime schedulers, hence it is fit for the need to compare executions based on different runtimes to clarify the subtle differences between them and between scheduling policies for the purpose of developing scheduling algorithms.

DAG Recorder flattens the computation DAG to a file as a sequence of nodes when the execution ends. In the sequence, a `create` node holds an offset pointing to the child `task` node that it spawns.

A `task` or a `section` node will hold an offset pointing to the subsequence of nodes of its inner subgraph. DAGViz memory-maps the file, which lazily loads only the accessed parts of the file into memory rather than the whole file at once. The visualization is also organized hierarchically with on-demand expansion/contraction. This hierarchical approach helps reducing stress on the memory even with huge DAG(s). DAGViz is built with GUI based on *GTK+* widget library [60], and canvas rendering based on *Cairo* vector graphics library [59].

6.5.1 Hierarchical layout algorithm

DAGViz traverses the DAG recursively from its root `task` node to a user-adjusted on-demand depth limit. At each traversal step, DAGViz proceeds next to these three directions in turn: the inner subgraph of current node (inward), the leftward subgraph following the creation edge (if the current node is a `create`), and the rightward subgraph following the continuation edge. In order to assign absolute (x, y) coordinates to every node, DAGViz needs to make two passes over the graph. At the first pass, it calculates in a bottom-up fashion the bounding boxes of three subgraphs (inner, leftward, and rightward) around every node. At the second pass, DAGViz assigns absolute coordinates to all nodes in a top-down fashion. At each traversal step from the root node down to leaf nodes, it assigns coordinates to the current node first before aligning three subgraphs (and all nodes inside) around it to their absolute coordiates based on their calculated bounding boxes. The root `task` node is first assigned with $(0, 0)$ coordinates.

Fig. 6.4 shows visualizations of the DAG extracted from an execution of Sort program. Node color represents the worker that has executed the node. The mixed color (of orange, yellow, and cyan) indicates that the node's subgraph was executed collectively by multiple workers rather than a single one. Fig. 6.5 shows the same DAG that has been expanded to depth 6 while the full DAG has max depth of 66 and contains dozens of thousands of nodes.

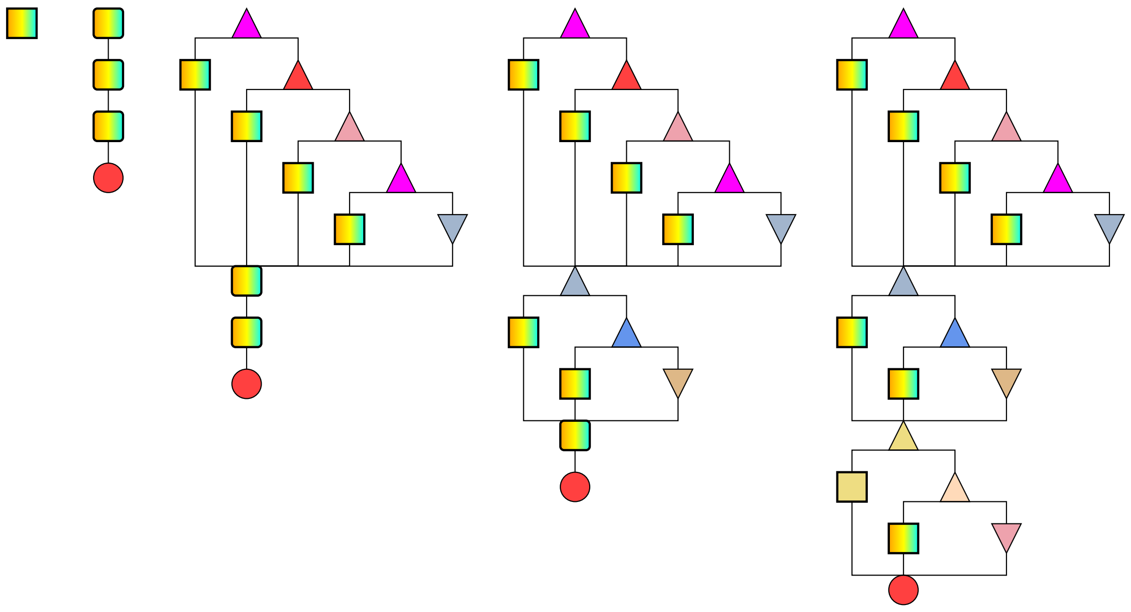


Figure 6.4: Sort's DAG(s) at depth 0 (first), 1 (second) and 2 (later 3). The DAG initially has only one node (the left most), from left to right it shows the DAG's hierarchical expansion. The original node gets expanded into three `section`s and one `end`, then the first `section` gets expanded, and the second and the third ones.

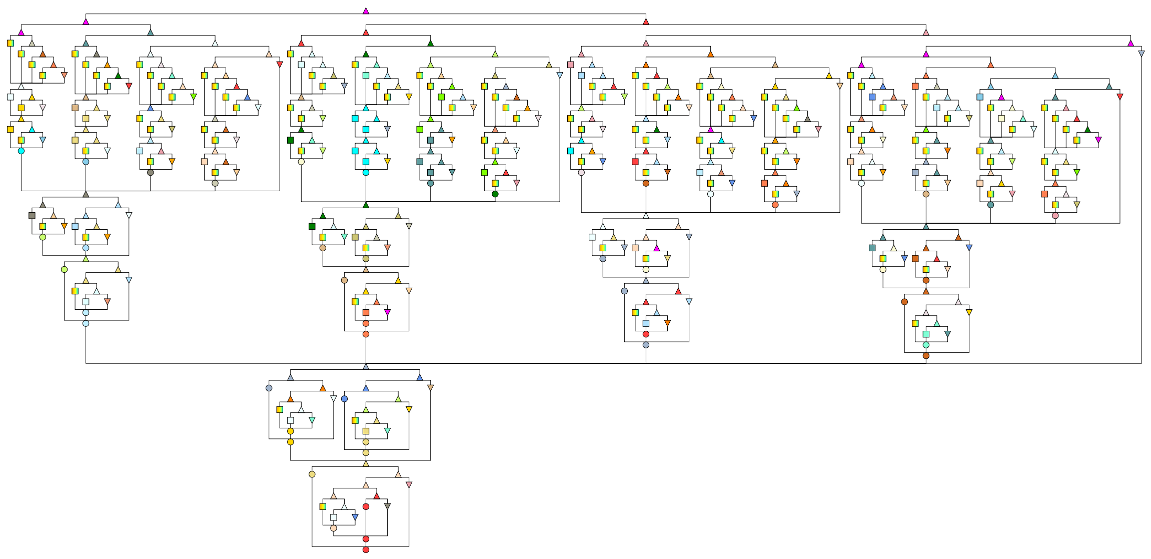


Figure 6.5: Sort's DAG expanded to depth 6 with less than 500 nodes but overwhelming already. While at max depth of 66 it contains up to dozens of thousands of nodes.

On DAGViz's GUI, users can interact freely with the DAG by panning it around to any part and zooming in or out at any part to enlarge or reduce that part of the graph infinitely (this is achieved largely by vector graphics feature of *Cairo*). Moreover, the DAG is not a static picture on screen but it can be expanded at once to any depth level of choice, or users can choose to expand it partly into any direction by clicking on any node to make it expanded. The expansion and contraction are enhanced with animation by gradual transitions between a collective node and its inner subgraph's topology so that these graph transformations look beautiful and importantly natural to the user perception.

6.5.2 Timelines with parallelism profile

The layout algorithm of the DAG can be modified a little to produce a timeline view of the execution. In timeline view the x-axis is the time flow and y-axis consists of a number of rows each of which corresponds to one worker thread. The rows contain boxes representing work that workers were doing at specific points in time during the program's execution. Each node of the DAG becomes a box in the timeline, so its y coordinate is fixed based on its worker number. The node's x coordinate is calculated based on its start time, and its length is based on its work ($= v.end - v.start$). Besides, DAGViz also draws a parallelism profile along with and placed right above the timeline. In Fig. 6.6, the lower part consisting of 32 rows is the timeline, the upper part (from red area upward) is the parallelism profile of the execution which is the time series of actual and available parallelisms of the execution:

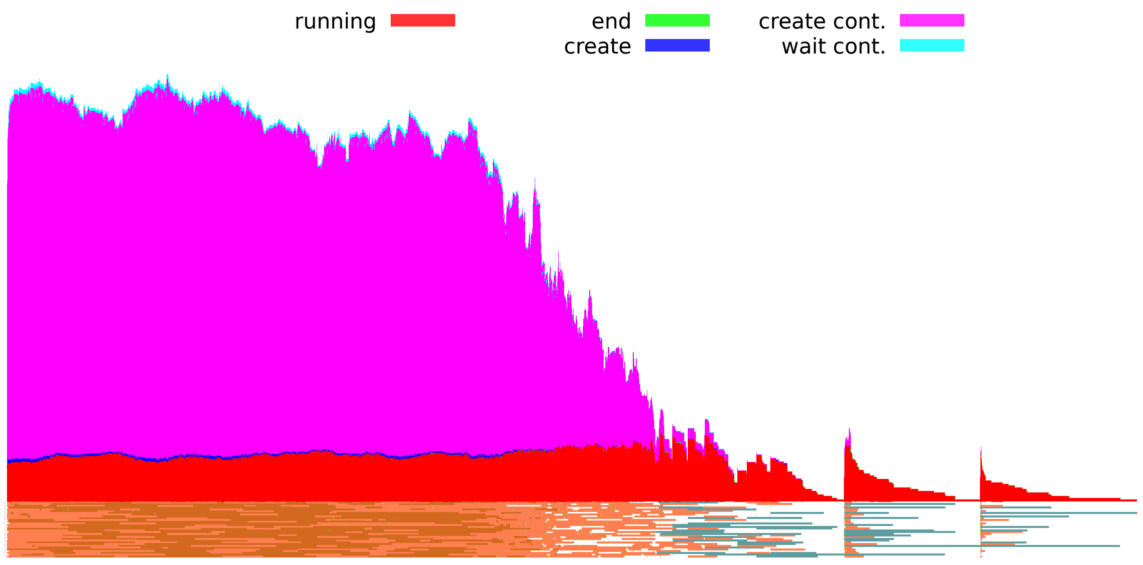


Figure 6.6: Sort's timelines are the lower part consisting of 32 rows. Sort's parallelism profile is the upper part consisting of a red area (actual parallelism) and stacked-up areas of other colors (different kinds of available parallelisms).

Time series of actual parallelism (red part): is the number of tasks actually running at every point in time. Actual parallelism at time t, denoted by $P_{\text{actual}}(t)$, can be obtained by:

$$P_{\text{actual}}(t) = \sum_{v \in V} \text{running}(v, t)$$

where V is the set of all nodes in DAG, $\text{running}(v, t)$ is 1 if v is running at time t and 0 otherwise. Formally,

$$\text{running}(v, t) = \begin{cases} 1 & \text{if } v.\mathtt{start} \le t \le v.\mathtt{end} \\ 0 & \text{otherwise} \end{cases}$$

Time series of available parallelism (upper parts of other colors): is the number of tasks *ready to run* but not actually running at every point in time. Available parallelism at time t, $P_{\text{avail}}(t)$, can be obtained by:

$$P_{\text{avail}}(t) = \sum_{v \in V} \text{ready}(v, t)$$

where $\text{ready}(v, t)$ is 1 if all of v's predecessors have been finished at time t but v has not been started; and 0 otherwise. Formally,

$$\text{ready}(v, t) = \begin{cases} 1 & \text{if } u.\mathtt{end} < t < v.\mathtt{start} \text{ for all} \\ & u \to v \\ 0 & \text{otherwise} \end{cases}$$

6.5.3 Kinds of visualizations

6.5.4 Related work

HPCToolkit [68] and Intel VTune Amplifier [33] both use sampling method and does not need to instrument the executable. These tools focus on hotspots analysis and timeline-based analysis.

Vampir [50] visualizes traces of an MPI program. Its main visualization is a timeline view with edges pointing from boxes to boxes to represent communication among processes. It simultaneously shows a statistical view that displays aggregate information of a chosen time interval in the timeline. Jumpshot [74] is a more general timeline visualizer. It visualizes data from text files of its own format. Jumpshot is not very flexible. It can only display up to 10 different categories which have 10 different colors. Jedule [29] is a tool to visualize schedules of parallel applications in timeline style. Olivier et al. [54] has used Jedule to visualize a timeline view for analyzing the locality of a scheduling policy. Aftermath [21] is a graphical tool that visualizes traces of an OpenStream [58] parallel program in timeline style. OpenStream is a dataflow, stream programming extension of OpenMP. Although Aftermath is applied in a narrow context of OpenStream (a subset of OpenMP), it provides extensive functionalities for filtering displayed data, zooming into details and various interaction features with users.

Wheeler and Thain [72] in their work of ThreadScope have demonstrated that visualizing a graph of dependent execution blocks and memory objects can enable identification of synchronization and structural problems. They convert traces of multithreaded programs to dot-attributed graphs which are rendered by GraphViz [11]. GraphViz is scalable (i.e., sufficiently fast for making animation possible) up to only hundreds of nodes, and quite slow with larger graphs because its algorithm [66] needs to care much about the aesthetic aspects of the graph such as node layering, edge crossing minimization. On the other hand, we leverage intrinsic characteristics of the computation DAG such as layered nodes (directed acyclic aspect), non-crossing edges to simplify the layout algorithm. DAGViz visualizes the DAG interactively with on-demand hierarchical expansion & contraction rather than a static whole-graph picture provided by GraphViz.

6.6 Case Studies

We have measured DAG(s) of all ten programs in the Barcelona OpenMP Task Suite (BOTS) [22] with five task parallel runtime systems DAG Recorder currently supports: OpenMP, Cilk Plus, Intel TBB, Qthreads and MassiveThreads. The experimental environment is shown in Table 6.1, and parameters for each benchmark described in Table 6.2. The overhead of DAG Recorder with MassiveThreads library is shown in Fig. 6.7. Except for particular cases of Health and UTS programs which create too many fine-grained tasks, DAG Recorder is feasible for all other programs with overhead within 10% of the original program's runtimes.

Compiler	Intel Compiler 14.0.2
OS	CentOS 6.4 (Linux 2.6.32-x86_64)
CPU	AMD Opteron 6380 2.5GHz 16 cores (8 modules) per socket
# Sockets	4 sockets (64 cores or 32 modules in total)
Runtimes	OpenMP, Cilk Plus, Intel TBB, MassiveThreads, Qthreads

Table 6.1: Environment

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`
Health	2^{14}	manual	`-f medium.input -x 3`
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`
UTS	2^{14}	-	`-f tiny.input`

Table 6.2: Summary of benchmarks settings

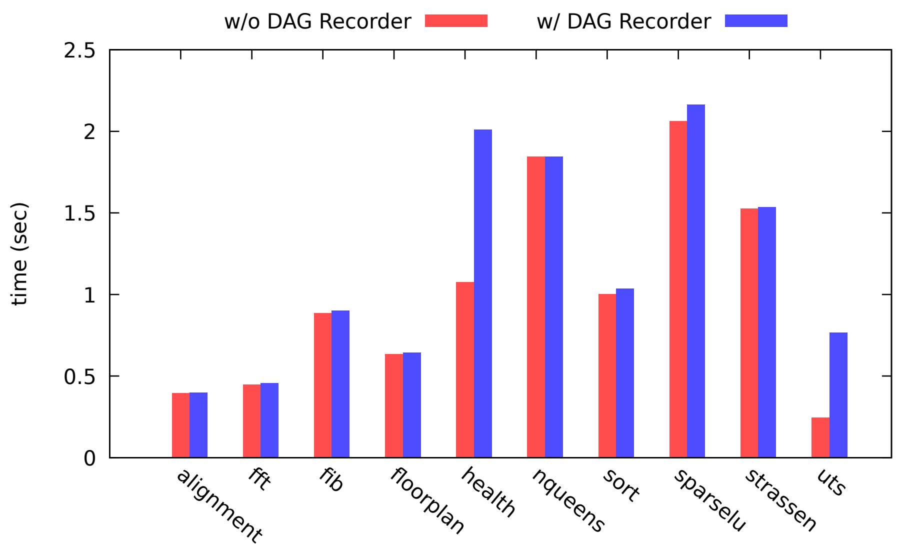


Figure 6.7: DAG Recorder's overhead in running programs in BOTS with MassiveThreads on 32 cores

We show a summary of the utilizations (= speedup/cores) on 32 cores of the benchmarks with five systems in Fig. 6.8. Each dot represents the utilization of an execution of a program by a system; the higher it is, the better. Among many cases of our interest, we look into two of them here. First, Sort's speedup is poor in all systems, which suggests that the program's code is the cause of performance bottleneck. The other case is SparseLU, as it is a peculiar case in which Cilk Plus's scalability is poorer than other systems, while Cilk Plus performs well in most other benchmarks.

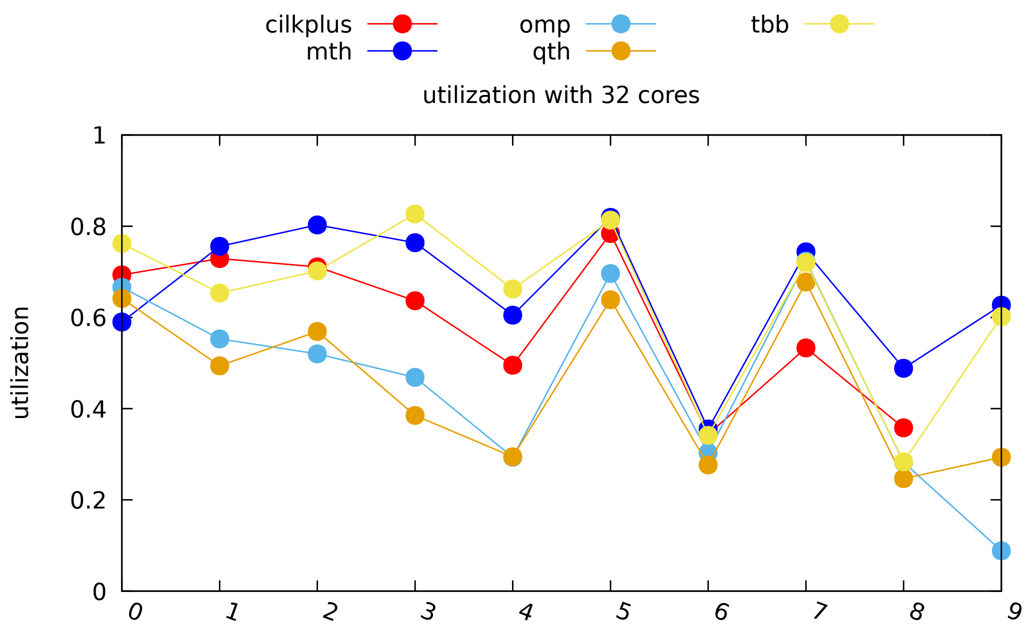
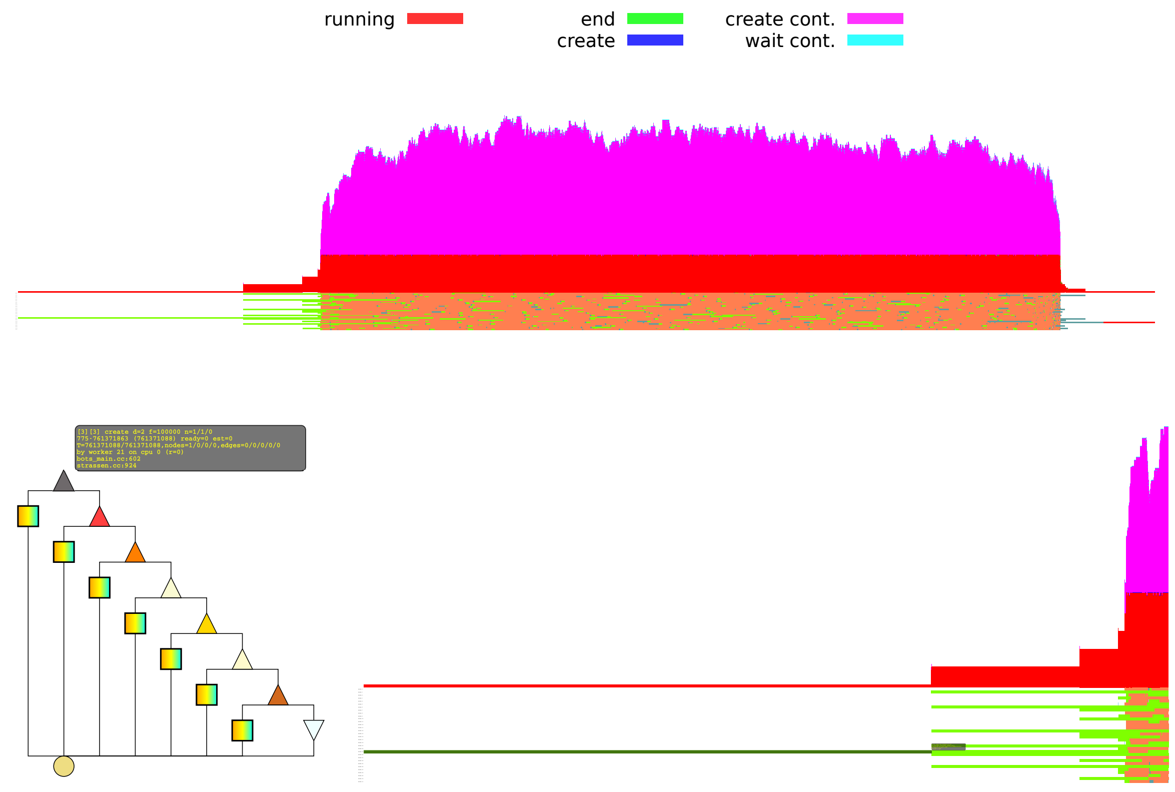


Figure 6.8: Utilizations of BOTS run by 5 systems on 32 cores

6.6.1 Sort

Sort program sorts a random permutation of n 32-bit numbers with a parallel variation of mergesort [22]. The input array is divided into smaller parts which are sorted recursively before being merged, also recursively, to become the sorted result array. In the algorithm, the recursive parallel merge is turned to simple sequential 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 in Fig. 6.6. By replacing this sequential memory copy with a version of parallel memory copy, the lack of parallelism in merging phase was fixed.

Similar to Sort, Strassen is another example where performance suffers from the lack of parallelism. The timeline of Strassen program in Fig. 6.9 shows that the program's parallelism is very low near the start. By zooming in and relating the long running box with DAG structure, we identified the code segment which enforced this low parallelism situation.



```
[3][3] create d=2 f=100000 n=1/1/0
775-761371863 (761371088) ready=0 est=0
T=761371088/761371088,nodes=1/0/0/0,edges=0/0/0/0/0
by worker 21 on cpu 0 (r=0)
bots_main.cc:602
strassen.cc:924
```

Figure 6.9: Strassen DAG's top node was actually a too-long-running interval demonstrated by the timeline view.

6.6.2 SparseLU

SparseLU program computes an LU matrix factorization over sparse matrices [22]. DAG visualization of SparseLU (Fig. 6.10) and its source code both show that it has a serial loop creating very many tasks, none of which recursively creates further tasks. Therefore, the program's parallelism increments one only after each iteration of the loop. The comparison of DAG(s) from Cilk Plus and Intel TBB in Fig. 6.11 expresses a noticeable difference between two systems. All nodes along the spine in Intel TBB's DAG (left one) are executed together by the same worker (of orange color), whereas in Cilk Plus's DAG (right one) these spinal nodes are executed separately by different workers (of different colors). This is because in Intel TBB when a worker creates a new task it pushes the new task into its work queue and continues executing the current one (help-first), whereas in Cilk Plus the worker would pause the current task to switch to executing the new task (work-first). Therefore, every parallelism increment requires a work stealing operation in Cilk Plus's execution, hence it is understandable that systems with help-first policy (OpenMP, Intel TBB, Qthreads) would execute SparseLU better than systems with work-first policy (Cilk Plus, MassiveThreads).

However, MassiveThreads still has significantly better utilization than Cilk Plus. We can observe it from Fig. 6.12 which shows parallelism profiles of MassiveThreads and Cilk Plus on 32 cores. It is noticeable that Cilk Plus exposes a low parallelism (around 25, as opposed to nearly 32 of MassiveThreads). The reason why MassiveThreads performs better than Cilk Plus can be explained by Cilk Plus's expensive work stealing operation. Fig. 6.13 compares the distribution of time gaps between two consecutive nodes on the spine. Cilk Plus takes much longer to advance a computation along it, implying that it takes longer to steal a task. Additionally, in our previous microbenchmark we have confirmed that work stealing operation in MassiveThreads is more than an order of magnitude faster than in Cilk Plus [69].

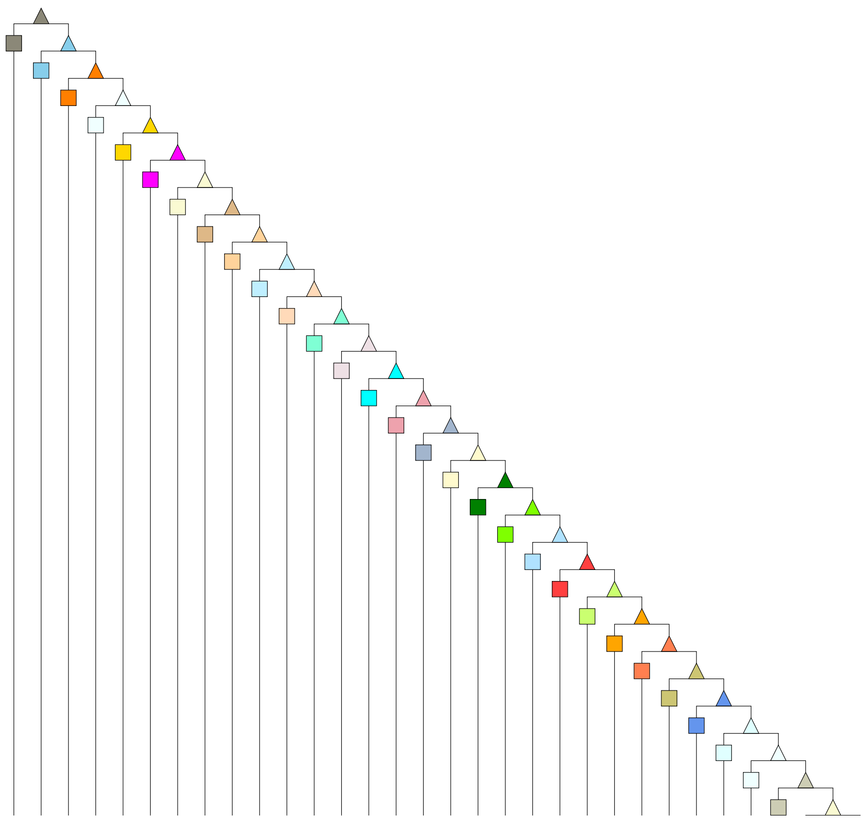


Figure 6.10: (Head part of) SparseLU's DAG by Cilk Plus

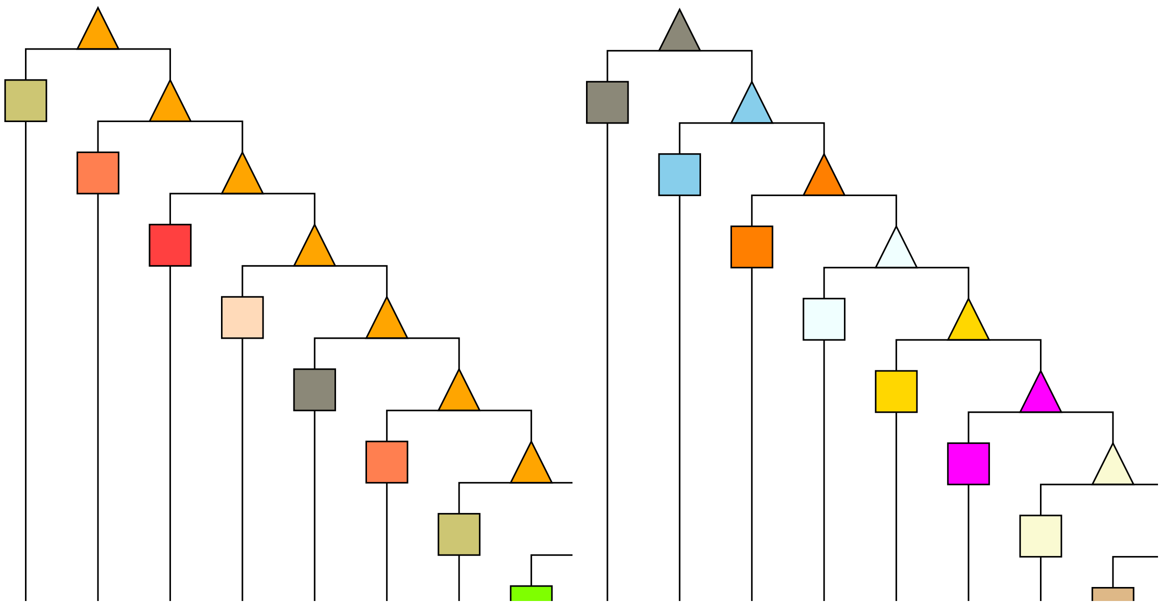
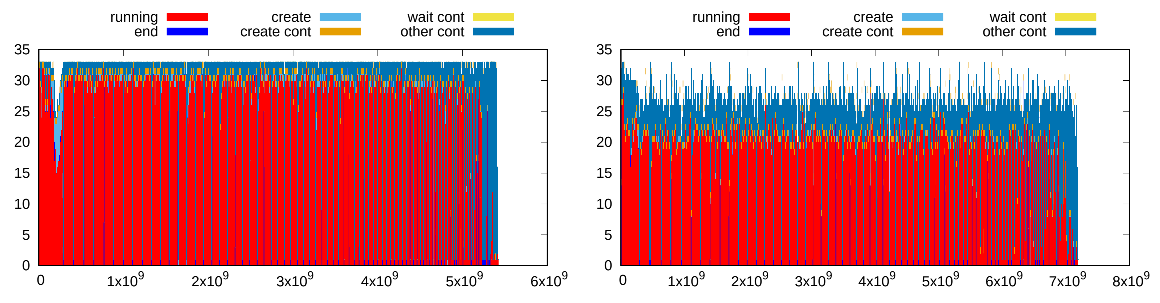


Figure 6.11: (Head parts of) SparseLU's DAG(s) by Intel TBB (left) and Cilk Plus (right)



(a) MassiveThreads

(b) Cilk Plus

Figure 6.12: SparseLU's parallelism profiles by MassiveThreads and Cilk Plus. While MassiveThreads consistently reaches 32 parallelism, Cilk Plus mostly floats around 25.

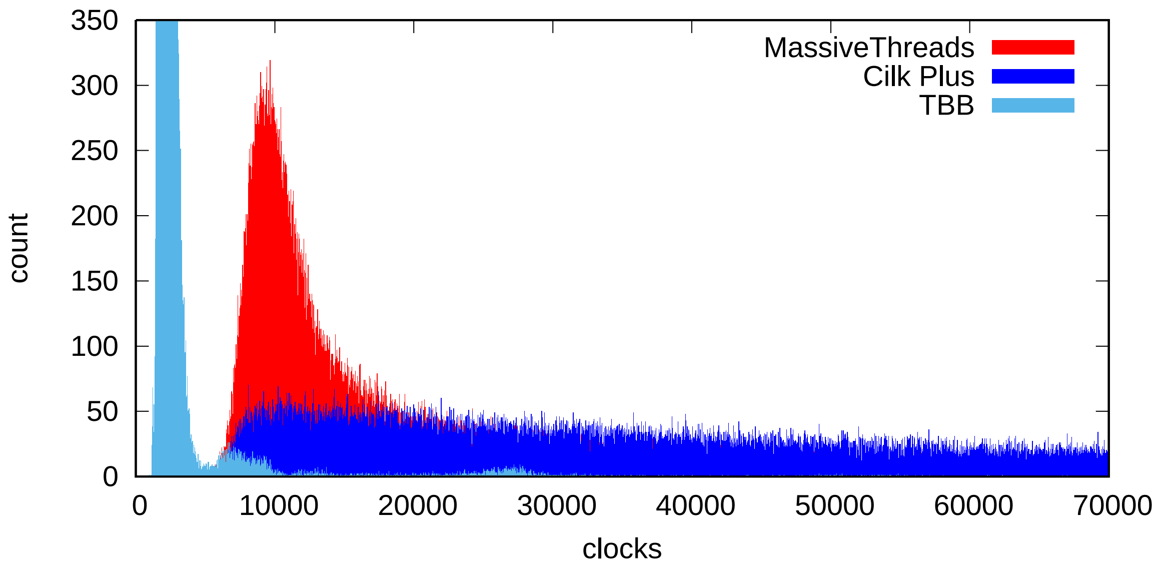
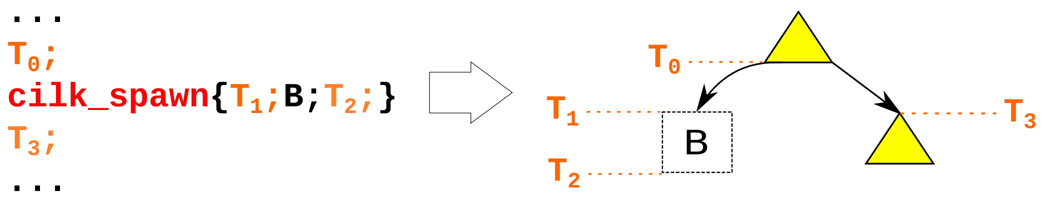


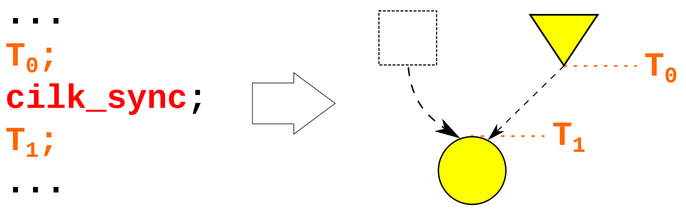
Figure 6.13: Distribution of work stealing time in SparseLU

6.7 Delay Spotter



```
...
T0;
cilk_spawn{T1;B;T2;}
T3;
...
```

(a) `CreateTask` translated to `cilk_spawn` with instrumentations



```
...
T0;
cilk_sync;
T1;
...
```

(b) `WaitTasks` translated to `cilk_sync` with instrumentations

Figure 6.14: Task parallel primitives are automatically instrumented to take timing information when they are translated to corresponding primitives of a specific system.

Delay Spotter is built upon our previous work on a performance visualization tool - DAGViz [31] - which provides interactive and scalable directed acyclic graph (DAG) visualizations and timelines (time-series of CPUs) visualizations based on the *computation DAG* traces. A computation DAG records the task graph from an execution of a task parallel program. It separates the execution of the application code into serial intervals which do not contain any task parallel primitives. Those serial intervals are denoted by nodes on the DAG; their dependencies are denoted by edges between nodes (Fig. 6.3). A task in the application may be divided into multiple nodes on the DAG based on multiple serial code segments interleaved by task parallel primitives in its code. A node is named after the primitive that *ends* it; a *create* node is ended by the task creation primitive (`CreateTask`), a *wait* node is ended by the task synchronization primitive (`WaitTasks`). Additionally, a node ended by the end of a task is named *end*. Edges are realizations of these task parallel primitives on the DAG. A `CreateTask` makes up two edges which both originate from the same create node representing the code segment ended by that `CreateTask`, but one connects to the child task (*create* edge), and the other connects to the continuation of the parent task following that `CreateTask` (*create cont.* edge). A task synchronization primitive first makes a *wait cont.* edge connecting from the wait node ended by that primitive to the continuation of the parent task following that primitive. Besides, the primitive also makes one or more *end* edges connecting from any child tasks that it synchronizes to the node that follows it. The delay component may be further divided into four subcategories based on these four kinds of edges on which a delay happens. There are also collective node kinds such as *task*, which represents a whole task and contains the sub-DAG of the task inside.

We define a generic set of API for basic task parallel primitives such as task-spawning `CreateTask` and tasks-synchronizing `WaitTasks`. Our thin layer based on preprocessing macros (*tpswitch* [31]) automatically translates these generic primitives into corresponding ones in specific target systems. Besides, along the translation, our tracing tool (*DAG Recorder* [31]) also automatically instruments necessary time measurement codes around the task parallel primitives in order to collect timing information. An example with Cilk Plus as the translation target is shown in Fig. 6.14. Our instrumentation method does not need to peek into the runtime's code, so it is more portable, easily to be adapted to a new system as long as the system supports a task-spawning primitive and a tasks-synchronizing primitive. With these instrumentations we can capture every time point when a task starts, stops, resumes, or ends; hence, work intervals can easily be separated apart from delay and no-work intervals. Beside start time and end time, a node on the DAG also has a *ready time* indicating the time point from which it becomes ready. The ready time of a node is calculated by tracing back all predecessors of the node and taking end time of the last finished one as the node's ready time. The interval from ready time to start time of a node is a delay of that node.

DAG has been used broadly in the literature to model a static parallel program. We leverage it to model a dynamic parallel execution. Our instrumentation-based measurement might seem to have a large overhead, especially when a large number of tasks are created in the application. But we have implemented a mechanism to collapse *uninteresting* (executed entirely by one worker) sub-DAGs dynamically on-the-fly in order to keep both time overhead and memory overhead under reasonable limit. Now the measurement overhead scales with the number of work-stealing across workers rather than the number of task creations. In actual experiments with BOTS, our tool's overhead was less than 10%.

DAGViz [31] visualizes the computation DAG traces captured by DAG Recorder in multiple kinds of visualizations. It provides a basic DAG visualization with node shapes similar to what are shown in Fig. 6.3, and an extended one with timing represented on y-axis. It also provides timelines visualization, which is made up by rearranging nodes into timelines of workers, along with a parallelism profile on top. In order to construct parallelism profile, each node on the DAG is traversed to compute its contributions to running and ready parallelism; one node adds one unit to ready parallelism from its ready time to its start time, and adds one unit to running parallelism from its start time to its end time. DAGViz allows us to zoom into any particular spot of the DAG on DAG visualizations, timelines, or parallelism profile; furthermore we can relate the structures of the same spot on multiple kinds of visualizations side by side, and compare them, get to understand the scheduling mechanisms.

6.8 Big DAG handling mechanisms

6.8.1 DAG-collapsing mechanisms

6.8.2 Big DAG visualizing mechanisms

Chapter 7

Task-Parallelizing PARSEC Benchmarks

The original PARSEC benchmark suite consists of a diverse and representative set of benchmark applications which are useful in evaluating multicore architectures for modern workloads. However, it supports only three programming models: Pthreads (SPMD), OpenMP (parallel for), TBB (parallel for, pipeline), lacking supports for emerging and widespread shared-memory task parallel programming models. In this work, we present a task-parallelized PARSEC (TP-PARSEC) in which we have added translations for five different task parallel programming models (Cilk Plus, MassiveThreads, OpenMP Tasks, Qthreads, TBB). Task parallelism enables a more intuitive description of parallel algorithms compared with the direct threading SPMD approach; besides, it also helps get rid of synchronizations (e.g., thead barriers) and ensures a better load balance among a large number of processor cores based on the proven work stealing technique. TP-PARSEC is not only useful for task parallel programming model developers to analyze their runtime systems with a wide range of workloads from diverse areas, but also enables them to compare performance between task parallel runtime systems. TP-PARSEC is integrated with a task-centric performance analysis and visualization tool which effectively helps users understand the performance, pinpoint performance bottlenecks, and especially analyze performance differences between systems.

7.1 Background

Multicore processors and shared-memory systems have been widespread, with increasingly many cores integrated on a processor chip. These higher core counts have put a pressure on the software layer; programmers need to be more careful in order to keep their parallel programs run efficiently, because more threads mean more contentions, more synchronizations, and longer remote memory accesses. Task parallel programming models are a popular approach in shared-memory programming. With task parallelism, programmers do not need to be aware of low-level details in the systems, like how many threads there are, then manually crafting the program's workload to the number of threads and scheduling it on these threads. A specialized runtime task scheduler in a task parallel programming model handles those things for users. The users are presented with a unified interface of tasks, they just need to focus on the program's logics to extract logical parallelism and denote them as tasks. Task parallel runtime systems will map these logical tasks onto available processing units automatically and dynamically at runtime. This dynamic scheduling is the basis for hiding latencies and tolerating noises.

Task parallel programming models are promising to be able to deliver both programmability and performance to a wider audience. However, there are still a lot of work to do to improve their performance. They need good benchmarks in order to be developed in the proper directions. A popular benchmark for task parallelism is the Barcelona OpenMP Tasks Suite (BOTS) [23], but it includes only basic divide-and-conquer computations such as fibonacci, nqueens, merge sort, matrix multiplication. Recursive algorithms are important, and task parallelism is well suitable for expressing recursions. However, evaluating task parallel programming models with mainstream workloads is also important to demonstrate their applicability in real-world applications.

The Princeton Application Repository for Shared-Memory Computers (PARSEC) [10] is a popular benchmark suite that contains representative workloads from a wide range of areas such as image recognition, financial analytics, physics simulation, data mining. It has been extensively used in research of multicore shared-memory systems. PARSEC is shipped with supports for POSIX Threads (Pthreads), OpenMP, and Intel Threading Building Blocks (TBB); benchmarks in PARSEC are mainly programmed with SPMD (single program multiple data) model based on Pthreads and parallel for loop models based on OpenMP, TBB. They lack the supports for task parallel programming models. That is why we have task-parallelized PARSEC, and presented a new benchmark suite TP-PARSEC (Task Parallel PARSEC) which adds supports for not one, but up to five different task parallel programming models (Cilk Plus, MassiveThreads, OpenMP Tasks, Qthreads, TBB). On one hand, TP-PARSEC extends the original PARSEC with emerging parallel programming models. On the other hand, TP-PARSEC brings a new set of state-of-the-art realistic workloads to system developers for them to evaluate the implementations of different task parallel programming models.

The performance of a task parallel programming model depends substantially on its runtime system (esp. runtime task scheduler) which automatically handles almost all details in the parallel execution. Different task parallel programming models may expose largely varying performance even when executing the same program because of their differences in, e.g., scheduling policies, load balancing algorithms. For example, in facesim benchmark, MassiveThreads has exposed up to 63% better speedup over Cilk Plus; or in canneal benchmark, Qthreads performs slightly (~22%) better than TBB does until 24 cores, but from 28 cores Qthreads suddenly degrades, resulting in ~42% lower performance than TBB's. By supporting multiple systems, TP-PARSEC becomes a useful benchmark suite that enables system developers to compare their system with others, analyze performance differences, and work on improvements of their system implementation.

In order to support users effectively in analyzing these performance differences, we have integrated into TP-PARSEC a task-centric performance analysis and visualization tool: Delay Spotter [30] (and DAGViz [31]) which can contrast differences between systems with a scheduling delay-based novel statistical metric and spot their specific causes on multiple kinds of visualizations (e.g., DAG, timelines, parallelism profile). These visualizations are also useful in helping users easily understand the performance and interactively explore the execution traces.

7.1.1 PARSEC

The original PARSEC benchmark suite developed by Princeton University [10] is a large benchmark suite consisting of 13 parallel applications and kernels. These applications and kernels are representative workloads in various areas: computer vision (bodytrack), animation physics (facesim, fluidanimate, raytrace), computational finance (blackscholes, swaptions), chip engineering (canneal), storage systems (dedup), search engines (ferret), data mining (freqmine, streamcluster), and media processing (vips, x264). They contain state-of-the-art algorithms for solving their specific problems in the fields.

These benchmarks are provided with three parallel implementations based on three multithreading libraries: Pthreads [53], OpenMP [19], and TBB [57]. All benchmarks have Pthreads versions, except for freqmine which has only one OpenMP version. Most of Pthreads versions of these benchmarks are implemented with *SPMD* model in which the data space (e.g., loop iterations) are divided equally among available threads; besides, dedup and ferret use *manual pipeline* models which are implemented manually upon threads, facesim and raytrace use *manual task queues* implemented upon threads. Some benchmarks additionally have OpenMP versions (blackscholes, bodytrack, freqmine) and TBB versions (blackscholes, bodytrack, ferret, fluidanimate, streamcluster, swaptions). The OpenMP versions use OpenMP's parallel loop model (*omp parallel for* directive). The TBB versions use either TBB's parallel loop model (*tbb::parallel_for* template) (blackscholes, bodytrack,

Table 7.1: Programming models of each version of each benchmark. A blank cell indicates the version does not exist.

App	Computation	Pthreads model	OpenMP model	TBB model	Task models
blackscholes	for (100 runs) { 1 for loop }	SPMD	omp parallel for	tbb::parallel_for	pfor
bodytrack	for (261 steps) { 5 for loops }	manual task queue	omp parallel for	tbb::pipeline tbb::parallel_for	pipeline tasks pfor
canneal	for (6000 steps) { 1 for loop }	SPMD			leaf tasks
dedup	for (streaming) { pipeline }	manual pipeline			pipeline tasks
facesim	for (100 steps) {21 for loops}	(SPMD) manual task queue			(SPMD) leaf tasks
ferret	for (3500 queries) { pipeline }	manual pipeline		tbb::pipeline	pfor
fluidanimate	for (500 steps) { 1 for loop }	SPMD		tbb::task	pfor
freqmine	7 for loops		omp parallel for		pfor
raytrace	for (200 steps) { recursive rendering }	manual task queue			recursive tasks
streamcluster	for (streaming) { 9 for loops }	SPMD		tbb::parallel_for tbb::task	pfor (SPMD) leaf tasks
swaptions	1 for loop	SPMD		tbb::parallel_for	pfor

streamcluster, swaptions), pipeline model (*tbb::pipeline*) (bodytrack, ferret), or low-level tasking interface (*tbb::task*) (streamcluster, fluidanimate). A summary of programming models in use is shown in Table 7.1.

7.1.2 Task parallel programming models

In task parallel programming models, a task is a logical unit of concurrency which can be created arbitrarily at any point in the program. These logical tasks are automatically mapped on available processor cores at runtime by the runtime system (with a task scheduler as an essential component). Because tasks can be created at arbitrary points, many typical parallel programming patterns such as for loops and recursions can be built on top of tasks. With these automatic load balancing and dynamic scheduling of tasks, programmers are relieved from many burdens, load imbalances and runtime noises can be effectively handled. Thus, task parallel programming models are promising in delivering both high performance and high productivity.

Many task parallel programming models exist. They have different concepts for scheduling, load balancing; and they differ substantially in their designs and implementations. Therefore, it is important to support multiple models in a benchmark suite intended for task parallelism. Our TP-PARSEC currently supports five different task parallel programming models: (1) Cilk Plus [61] which is a language extension of C/C++, providing two simple keywords for expressing task parallelism (`cilk_spawn` and `cilk_sync`); (2) MassiveThreads [52] [51] which is a lightweight thread library, and like Cilk Plus, uses work-first scheduling policy and random work stealing technique in its task scheduler; (3) OpenMP [19] which is a widely-used framework for shared-memory parallel programming (we use OpenMP Tasks which has been introduced from OpenMP 3.0); (4) Qthreads [71] which is also a lightweight thread library, with a locality-aware scheduler; (5) TBB [57] which is a commercial-level threading library equipped with a wide range of parallel programming patterns and algorithms.

7.2 TP-PARSEC

Our TP-PARSEC is based on PARSEC 3.0, the latest version (as of October 2017). It is equivalent to the PARSEC's core package, excluding input datasets which can be downloaded separately from the

Table 7.2: Corresponding task parallel primitives in specific models

	Cilk Plus	OpenMP	MassiveThreads	Qthreads	TBB
`create_task`	cilk_spawn	#pragma omp task	myth_create()	qthread_fork()	tbb::task_group::run()
`wait_tasks`	cilk_sync	#pragma omp taskwait	myth_join()	qthread_readFF()	tbb::task_group::wait()

PARSEC website.

7.2.1 A unified task parallel API

By defining a thin generic macro-based wrapper covering all five underlying models, we could simplify our conversion. We just need to write the code once using the generic task parallel primitives, then the program can be preprocessed automatically into supported underlying systems. The generic wrapper is called "'tpswitch"', and it is published in our MassiveThreads [2] repository (*massivethreads/src/tpswitch/tpswitch.h*). The wrapper includes two basic primitives: `create_task` for creating a task, and `wait_tasks` for synchronizing tasks (of the innermost scope). These primitives are translated to corresponding API of specific models (Table 7.2) during the preprocessing stage of the compilation. `create_task` is translated to Cilk Plus' cilk_spawn statement, MassiveThreads' myth_create() function, OpenMP's "#pragma omp task" directive, Qthreads' qthread_fork() function, and TBB's tbb::task_group::run() method. `wait_tasks` is translated to cilk_sync, myth_join(), "#pragma omp taskwait", qthread_readFF(), and tbb::task_group::wait() respectively.

Besides, we also introduce `pfor` (parallel for) primitive which divides the for loop's iterations recursively into two halves and creates two tasks executing them at each recursive level. `pfor` uses the above `create_task` and `wait_tasks` primitives to spawn tasks. It also accepts an input grain size value which indicates at what point the recursive division should stop and the leaf computation should be executed on the current set of iterations. This grain size notion is similar to the chunk size option in the "schedule" clause of OpenMP's parallel for directive and the grain size parameter in TBB's *tbb::parallel_for* template. Additionally we have implemented `pfor_reduce` which is similar to `pfor` but with an additional feature of reducing private values across all iterations. Its notion is identical to that of OpenMP's "reduction" clause and TBB's *tbb::parallel_reduce* template. In TP-PARSEC, only streamcluster benchmark uses this `pfor_reduce` primitive.

7.2.2 Task-parallelizing PARSEC

In this section, we describe the computation model of each benchmark, how it is implemented in the original Pthreads, OpenMP, TBB versions, and how we translated it into task versions. We have translated 11 out of 13 benchmarks excluding vips and x264 which have large code base because of limited time. We assume the *native* input set (the largest one), when talking about specific numbers of, e.g., elements, loop iterations, input images. We use N to denote the problem size, P to denote the number of threads. The grain size (work granularity) of the SPMD model is N/P because SPMD model divides data space into P equal parts for P threads to execute, each part contains N/P data elements. A summary of programming models used in each benchmark is shown in Table 7.1. A summary of the grain size set for each benchmark is shown in Table 7.3.

(1) Blackscholes

Blackscholes is a workload in computational finance, it calculates the prices of a portfolio with the Black-Scholes partial differential equation. This benchmark has a simple programming model: there is only one flat for loop iterating over ten million (10^7) options (which is repeated for 100 times). Because loop iterations are independent from each other and load-balanced, blackscholes can easily be loop-parallelized and it has actually been loop-parallelized with OpenMP's parallel for directive

Table 7.3: Work granularity of each version of each benchmark

App	Pthreads	OpenMP	TBB	Task
blackscholes	N/P	def.	def.	10000
bodytrack	4 – 32	1 – 32	def.	16
canneal	N/P			100
dedup	∅			∅
facesim	N/P			N/P
ferret	∅		∅	1
fluidanimate	N/P		$N/(P \times 8)$	1
freqmine		def., 1		1
raytrace	32			8
streamcluster	N/P		N/P	50, N/P
swaptions	N/P		1	1

(N/P: coarse-grained like SPMD; def.: default; ∅: none)

and TBB's parallel for template in its OpenMP and TBB versions. In Pthreads version, the loop iterations are divided and distributed equally among participating threads (SPMD model). In task versions, we do similarly by simply applying `pfor` in place of the parallel for primitives of OpenMP or TBB, and the loop is hierarchically divided into fine-grained tasks, with grain size 10000. How this grain size was chosen is discussed in Chapter 8.

(2) Bodytrack

Bodytrack is a computer vision workload which recognizes a human body and tracks its movement through a sequence of images input from observation cameras. At each frame, multiple images from multiple cameras capture a scene of a person from different angles, and the person moves from frame to frame. Bodytrack regconizes poses of the human body in the input images, marks these poses and returns annotated images. This benchmark represents the significance of computer vision algorithms popularly used in areas like video surveillance, character animation, computer interfaces, etc.

At a time step (frame), the benchmark processes 4 input images through three stages: *read* images in, *process* images, and *write* the processed images out. In the parallel implementations (Pthreads, OpenMP, TBB), five for loops in the second stage are parallelized, other than that the program executes sequentially stage after stage (*read* → *process* → *write*), and frame after frame. Pthreads version employs a manual thread pool imlementation (*WorkerGroup*) which creates P worker threads and makes them wait on a condition variable until there are jobs. The iterations of a parallel for loop are not divided into exactly P parts for P threads, but into many more parts with finer-grained sizes: 8, 8, 8, 32, and 4 for each loop respectively. The worker threads compete with each other through a mutex lock to acquire the next part to execute until all iterations are processed. Besides P threads created for executing the loop, Pthreads version also deploys an additional separate thread specialized in doing *read* stages. OpenMP and TBB versions use their parallel for primitives for all five loops. OpenMP version does nothing more than that. But TBB version additionally employs a pipeline model on the program's three stages; the pipeline is created with two stages: the first one contains the program's *read* stage and two first loops of *process* stage, the second one contains the latter three loops and the program's *write* stage.

In our first implementation of task versions, we only parallelized the five for loops (with `pfor`) just like the OpenMP version, without any pipeline. However, we then realized, by our performance tool, that *read* and *write* stages are considerably long, and run sequentially in long serial intervals, making other threads wait wastefully, hindering the scalability of the whole computation. Therefore, in order to improve the performance we have changed the parallelization model a little bit by overlapping computation (*process* stages) and communication (*read* and *write* stages) across three consecutive frames (to retain stage dependencies). Although three stages in a frame are serially dependent, and *process* stage is exclusive between frames, *write* stage of the current frame is independent from *process* stage of the next frame, and two of them are independent from *read* stage of the next next frame. Hence, it is possible to overlap *process* stage of frame t with *write* stage of frame $t-1$ and *read* stage of frame $t+1$. In order to realize that we have made each stage a separate task and run the three tasks *process*(t), *write*(t-1), and *read*(t+1) in parallel. By making this change we could reduce serial intervals considerably. The performance improvement is discussed in the Chapter 8. Task parallelism has enabled this computation-communication overlapping to be done easily.

(3) Canneal

Canneal is a kernel that optimizes the routing cost of a chip design. It uses a simulated annealing algorithm. During execution it goes through 6000 temperature steps. At each temperature step, 15000 moves are made and tested. Each move picks a random element, exchanges its position, and evaluates whether it is beneficial for the optimization goal. After all moves at a step are completed, the global temperature for the simulated annealing is adjusted. The temperature steps need to be processed one by one in order to incrementally adjust the global temperature. There is only Pthreads version for canneal in PARSEC, no OpenMP and TBB implementations provided. Ptheads version divides 15000 moves equally for participating threads, whereas task versions divide them into fine-grained tasks, each of which processes 100 moves. Since an element could potentially be modified by multiple tasks at the same time, protection is necessary. A library that provides lock-free access is used. It uses data race recovery instead of avoidance. This is kept the same in task versions.

(4) Dedup

Dedup is a kernel that compresses an input data stream. Pthreads version uses a manual pipeline model implemented on top of threads in which each input data chunk is processed sequentially through five stages:
fragment → *refine* → *deduplication* → *compress* → *reorder*
Different numbers of threads are deployed for each stage: 1 thread for *fragment*, n threads for *refine*, n threads for *deduplication*, n threads for *compress*, and 1 thread for *reorder* (1 → n → n → n → 1) (n is the number of threads specified via the management script's "-n" option). *Fragment* stages reading data in and *reorder* stages writing data out are serially dependent. In task versions, we make each chunk with its middle three stages as a task; the root task executes *fragment* stages serially: reads data in and creates a task for each chunk to execute *refine*, *deduplication*, *compress* of that chunk; after every 27 chunks, the root task synchronizes these 27 child tasks, then creates a task for executing serially 27 *reorder* stages of these 27 chunks which have just been processed. This *reorder* child task is run in parallel with the next 27 compute child tasks. The number 27 was chosen empirically based on our experiments, it provides a good enough granularity for *reorder* tasks to be run in parallel with other compute tasks.

(5) Facesim

Facesim computes a realistic animation of a human face by simulating a time sequence of muscle activation. The important data structure is a statically partitioned mesh. Multiple processes are applied to this mesh in every frame that is simulated: (1) applying Newton-Raphson method to solve a nonlinear system of equations, (2) iterating over all tetrahedra of the mesh to calculate the force contribution of each node (3) using the conjugate gradient algorithm to solve a linear equation system.

Pthreads version parallelized totally 21 loops in the program code; in the native input run, with 100 frames (time steps), these loops were invoked totally 61601 times. These loops generally applies some kinds of operations on the whole mesh data structure, e.g., clearing array, copying array, array addition. The mesh has been organized at the program's beginning so that it is readily broken into a *fixed* number of sub-meshes equal to the number of threads. That is why facesim can only run with a power-of-2 number of threads. Pthreads version uses a manually implemented task queue (*TaskQ*) in order to schedule work (tasks) onto threads. When one of the processing operations is to be applied on the mesh, tasks are created to operate on every sub-mesh. The number of tasks equals the number of sub-meshes and the number of threads. TaskQ's scheduler simply assigns newly created tasks to threads in a round-robin fashion which is less efficient than the work stealing method usually deployed in a genuine task parallel programming model. TaskQ provides two main functions: `TaskQ.Add_Task()` for adding a task to the queue, and `TaskQ.Wait_For_Completion()` for synchronizing created tasks. We have translated facesim to task parallelism simply by replacing the calls to `TaskQ.Add_Task()` with `create_task`, and the calls to `TaskQ.Wait_For_Completion()` with `wait_tasks`. Tasks of the program are then scheduled by a genuine work stealing scheduler of the supported models instead of TaskQ.

(6) Ferret

Ferret is a content-based similarity search tool of feature-rich data such as audios, videos, images; in this benchmark it is configured as an image similarity search workload. It inputs 3500 query images for each of which it finds (up to 50) similar images from an image database containing vectorized data of 59695 images. Ferret is originally provided with two versions Pthreads and TBB. In these versions, the benchmark is organized as a pipeline programming model processing 3500 input images one by one. The pipeline consists of 6 stages:
load → *segment* → *extract* → *index* → *rank* → *output*
Each image goes through these stages one by one. There is a difference in the number of threads used in Pthreads and TBB versions. Pthreads version, like in dedup, deploys different numbers of threads for stages: 1 for the first and last stages, n for other stages in the middle of the pipeline (1 → n → n → n → n → 1); whereas TBB version deploys exact n threads which are shared among all stages during the program execution.

Task versions also deploy exact n threads. In task versions, we remove the pipeline and exploit the data parallelism among a specific number (100) of input images. For every 100 input images, we apply `pfor` to recursively create tasks processing them. In order to make images able to be processed in parallel, it is necessary to resolve the exclusiveness of the *output* stage which modifies the global data structure. We have detached it out of the tasks processing images, we do *output* stages of all processed images serially at the end of the program. These *output*s just write a small amount of texts to file, so they actually run quickly and do not leave any noticeably long serial interval at the end of the execution.

(7) Fluidanimate

Fluidanimate is a stencil computation which operates on a 3-dimension grid (mesh) through 500 steps, the grid at each step is computed based on its state at the previous step. Pthreads version divides the grid into a number of identical blocks which is equal to the number of threads, by splitting uniformly along x-axis and z-axis (keeping y dimension the same). TBB version further divides a block into 8 sub-blocks by splitting further along the z-axis, and uses *tbb::task* interface to create tasks each of which works on one sub-block. TBB version creates 8 times as many tasks as Pthreads version does. In task versions, we still divide the grid into identical blocks along x-axis and z-axis like in Pthreads version, but these blocks are now more fine-grained (grain size 1). One task works on one block. We use `pfor` to create these tasks hierarchically rather than using a flat loop creating all tasks at once. The benefit of this hierarchical division is that it enables closer tasks to be more likely executed by the same thread, hence exploiting better locality.

(8) Freqmine

Freqmine is a program that detects frequent patterns in a transaction database and uses association rule mining which is very common in data mining applications. Freqmine uses an array-based version of the Frequent Pattern-growth method. In the original PARSEC, only OpenMP version is provided, containing 7 parallel for loops. The algorithm in general consists of three steps. The first one is to build the FP-tree header. In this step, the database is scanned and a table with frequency information is built. It is implemented with 1 parallel loop. The second step performs another scan of the database; it consists of 4 parallel loops. The third step is the actual data mining. Multiple FP-trees are constructed from the existing tree using 2 parallel loops. In task versions, we just replace OpenMP's parallel for directives with `pfor` primitives. We set the grain size at 1 for all 7 `pfor`(s), though in OpenMP version some loops were set with 1 and some were not set (the default is OpenMP implementation-dependent, and usually the coarse-grained one: N/P).

(9) Raytrace

Raytrace is a well-known rendering algorithm, it synthesizes an image by simulating the camera, light sources, objects, and tracing all light rays from every pixels in the image to determine if it can reach back to any of the light sources. In this benchmark, 200 continuous frames are rendered, each has a resolution of 1920×1080 pixels. Pthreads version exploits a manual task queue just like in facesim. But it is another task queue implementation (*MultiThreadedTaskQueue*). Each task handles an area of 32×32 pixels (or smaller) of the full image, so there are totally around $60 \times 34 = 2040$ tasks created. Although the task queue implementation is quite complicated, participating threads basically compete with each other through a mutex lock to acquire each avaialable task to execute. The threads acquire lowest tasks to execute first, then proceed to higher tasks along x-axis, then y-axis.

In task versions, we create more finer-grained tasks. Each task now handles a smaller area of 8×8 pixels, hence there are 16 times as many as tasks created (around $240 \times 135 = 32400$ tasks) than it is in Pthreads version. These leaf tasks are not created all at once, but recursively. At each recursive stage, the 2-dimension frame (1920×1080) is split along the longer dimension until it reaches the size of 8×8 pixels. Similarly to fluidanimate, it is more likely to achieve a better locality with this recursive division.

In this benchmark, 200 frames were identical, but in real applications these frames can be continuous pictures of the objects, camera, or light sources that move. Therefore, these frames are serially dependent, and need to be processed sequentially, not in parallel.

(10) Streamcluster

Streamcluster is a kernel solving the clustering problem commonly seen in data mining workloads. In this benchmark, there are totally 10^6 input points which are divided into 5 blocks, each of which contains 2×10^5 points and is input to the program as simulated streaming data. A small subset of points are selected as local centers for each block and these subsets are cumulated (up to 500 points) after each block is processed. After finishing all blocks, these selected local centers are clustered again in order to select a predefined smaller number of final centers ($10 - 20$ points). There are 9 parallel for loops which operate on the array of 2×10^5 points of a block. Pthreads version applies SPMD model, dividing loop iterations into equal parts for available threads (each has $N/P = 2 \times 10^5 \div 36 \approx 5556$ iterations). TBB version applies *tbb::parallel_for* (and *tbb::parallel_reduce*) pattern to 4 loops (with grain size N/P), and applies *tbb::task* interface to the other 5 loops. It creates exactly P *tbb::task*(s) for P threads (so grain size N/P). Therefore, TBB version is basically using the SPMD model. In task versions, we follow TBB version's model, and apply `pfor` in place of *tbb::parallel_for* (`pfor_reduce` in place of *tbb::parallel_reduce*), `create_task` in place of *tbb::task*. However, we set the grain size for `pfor`(s) at a low value 50, in order to make fine-grained tasks.

(11) Swaptions

Swaptions computes the prices of a portfolio of swaptions using Monte Carlo (MC) simulation. Its parallelization model is as simple as that of blackscholes, there is only one parallel for loop. Swaptions is provided with Pthreads and TBB versions. Pthreads version applies SPMD model, dividing the loop into equal parts for available threads (grain size N/P). TBB version applies its parallel for pattern with grain size 1. In task versions, we applies `pfor` with also grain size 1. One note is that swaptions code is currently not auto-vectorized by both gcc and icc, whereas blackscholes code is auto-vectorized by icc (not gcc). There are three main reasons: (1) swaptions' compute functions are scattered in multiple source files, (2) the for loop has multiple exits, (3) its data arrays are not aligned yet.

7.2.3 Performance analysis tool

TP-PARSEC is integrated with our performance analysis and visualization tool [30] [31] which is specialized for task parallelism. The tool has two parts: a tracer and a visualizer. The tracer (DAG Recorder) captures a directed acyclic graph (DAG) of tasks from an execution (of a task version), and the visualizer (DAGViz) visualizes the trace to help users understand the performance and pinpoint the bottlenecks. DAGViz enables users to explore the trace through multiple kinds of interactive visualizations such as a network graph (DAG) which represents the logical task structure of the program, timelines of threads, or a parallelism profile which is a time series of runnable and running parallelism during the execution. Timelines and parallelism profile visualizations we show in this paper are provided by DAGViz [31]. Parallelism profile allows users to get an overall understanding of the performance, then DAG and timelines visualizations enable users to zoom into any spot of the whole DAG of the execution and inspect in detail any task that caused the problem.

Moreover, the tool provides a novel task-centric statistical metric which helps users quickly acquire a first impression on how well the program scales: the breakdown of the cumulative execution time into four categories of *work*, *delay*, *no-work-sched*, and *no-work-app* (cumul. exe. time = elapsed time × threads = work + delay + no-work-sched + no-work-app) based on scheduling delay, which is described in our previous work [30]. Work is the total time that all threads spend executing the program code. Delay is the time during which a thread is not executing the program code and there is at least a ready task in the system that is waiting to be executed, a delay is caused by the runtime scheduler for not matching up the free thread and the ready task fast enough. No-work (= no-work-sched + no-work-app) is also the time during which a thread is not executing the program code, but there is no ready task in the system at that time to feed that thread. No-work is actually not caused solely by the program's algorithm for not creating enough parallelism, but sometimes caused by the scheduler for, e.g., not resuming a critical parent task (that can spawn more parallelism for idle threads) fast enough. So no-work-sched is that part of no-work caused by the scheduler, and no-work-app is the other part caused by the insufficient parallelism in the program's algorithm. Delay can be considered as a measurement of scheduling overhead (e.g., task creation, synchronization, work stealing). No-work-app can be considered as a measurement for the impact of serial regions remaining in the parallel program's code.

7.2.4 Improved central management script

The original PARSEC is equipped with a handy central management script (*parsec/bin/parsecmgmt*) which allows users to do all things through it: from compiling, cleaning, to running the benchmarks with different configurations, different inputs, different numbers of threads. We have extended the script to *parsecmgmt2* which does not only maintain all things that *parsecmgmt* can do, but also supports new configurations for the newly added task parallel versions. The names of new configurations follow existing PARSEC naming pattern: {compiler}-{type}-{extension}; compiler can be “gcc” or “icc”, same as before; type is not only “pthreads”, “openmp”, “tbb”, same as before, but also “task_cilkplus”, “task_mth”, “task_omp”, “task_qth”, “task_tbb” which are task versions based on Cilk Plus, OpenMP, MassiveThreads, Qthreads, and TBB respectively; extension can be none or “hooks”, same as before, and now additionally “dr” which indicates to compile and run with DAG Recorder tracer. For example, re-compiling and running two benchmarks blackscholes and bodytrack with icc, MassiveThreads, DAG Recorder, native input, and 36 threads can now be done with only one command below:

```
tp−parsec/bin $ ./parsecmgmt2 −a uninstall build run
  −p blackscholes bodytrack −c icc−task_mth−dr
  −i native −n 36
```

7.3 Related Work

Barcelona OpenMP Tasks Suite (BOTS) [23] is a popular task parallel benchmark suite which consists of 10 applications. Most of the applications are simple divide-and-conquer algorithms parallelized only by OpenMP Tasks. They are not representative of realistic applications and do not supoprt other task parallel programming models.

PARSECSs [17] ports 10 PARSEC benchmarks to the OmpSs model and its runtime system (based on OpenMP 4.0 Tasks). PARSECSs achieved equivalent scalability improvements by using OpenMP’s tasks and dataflow model. The implementation is limited to only one runtime system, and the support for original versions has been removed from the suite (to attain reduction in lines of code) which makes it less than a complete benchmark suite. This work is mainly a showcase to demonstrate the advantages of task parallelism over SPMD, manual pipeline, or manual task queue; it is not intended as a benchmark suite for general usage.

In [40] three of the PARSEC applications are ported to a pipeline task parallel model. They use a novel extension of the Cilk language to express pipeline parallelism.

Various papers characterize PARSEC benchmarks and introduce optimizations over them. Optimizations regarding NUMA and prefetching for three of the PARSEC benchmarks are introduced in [45]. A scalability analysis considering input sizes of PARSEC benchmarks is given in [65]. Data sharing patterns are examined in [8]. In [9] the benchmarks are evaluated using hardware performance counters. A vectorized version of PARSEC is introduced in [16] and characterized using hardware performance counters. The PARSEC paper [10] includes a hardware-centric analysis of the benchmarks such as working set size, cache miss rates, shared data, cache traffic, and off-chip traffic, but it is based on simulations, not real machines. We analyzed TP-PARSEC on a large multicore machine with a built-in DAG-based performance tool which puts the focus on tasks and performance differences between systems.

7.4 Conclusion

We have presented TP-PARSEC - a benchmark suite extended from PARSEC with supports for multiple task parallel programming models and integrated with a powerful task-centric performance analysis and visualization tool. TP-PARSEC maintains all good aspects of PARSEC: a large set of emerging workloads in diverse areas, state-of-the-art techniques and algorithms in those areas, good support for research with a central management script. TP-PARSEC is intended to be a useful benchmark suite for task parallel programming model developers to study task parallel applications and analyze performance differences between runtime task schedulers. TP-PARSEC is also useful for system architects to study their systems with widespread task parallel programming models together with emerging application workloads.

Chapter 8

Evaluation

- C: GTK+ – Cairo – Gnuplot
- C++: Qt – QPainter – Qt Charts (QCustomPlot)
- Python: PyQt – Matplotlib

While GIMP and Inkscape are still based on GTK+, Wireshark has been ported away from GTK+ to Qt. VLC is another best example of using Qt. GTK+ was originally developed because Qt was a close-source project at that time. GTK+ is too focused on GNOME desktop systems with little care for other systems.

Qt has some advantages over GTK+ like:

- Qt is **more cross-platform**. Beside Windows, macOS, and Linux/Unix, Qt also supports Android, iOS, Ubuntu Phone, WebOS, Symbian, Windows CE.
- GTK+ does not give **native look-and-feel** on macOS and Windows, while Qt tries to use native OS API where possible.
- Qt has powerful supports for **plotting** (with native Qt Charts, QGraphicsView, or add-ons like Qwt, QCustomPlot) and **vector drawing** (with native paint system of QPainter → QPaintEngine → QPaintDevice), while GTK+ does not have a good native plotting support (although GTK+ has good vector drawing based on the `cairo` library).

FLow Graph Designer [70].

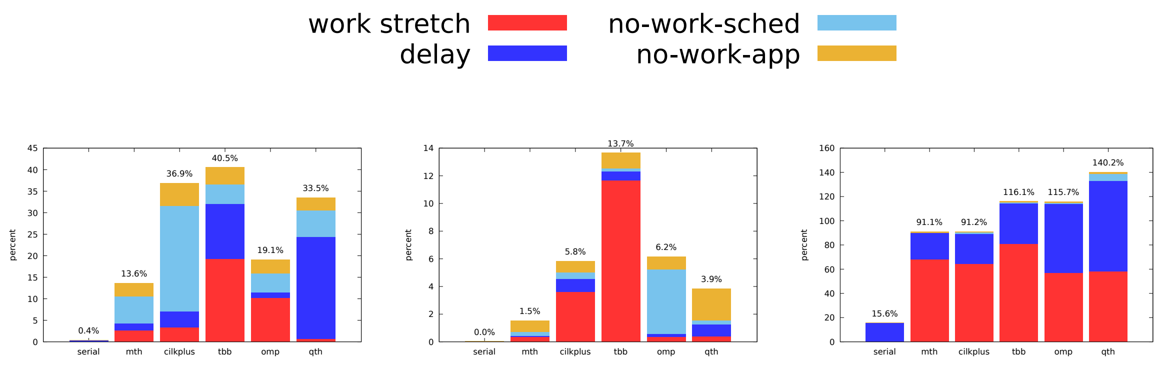
8.1 BOTS

We have evaluated our tool with 10 benchmarks from Barcelona OpenMP Tasks Suite (BOTS) [22] and 5 different task parallel runtime systems: Cilk Plus, MassiveThreads (0.97), OpenMP, Qthreads (1.10), and TBB (2017 Update 1). MassiveThreads, Qthreads, and TBB were compiled from source with Intel C++ Compiler (icc) 14; Cilk Plus and OpenMP were the implementations distributed with the icc 14 package. The measurement overhead is feasible for 8 benchmarks (except Health and UTS) in which execution times increase less than 10% with the measurement tool. Health and UTS algorithms create too many fine-grained tasks, making the DAG huge, causing execution times to increase up to 2-3x. The arguments passed to each benchmark are summarized in Table 8.1: task stack size (for MassiveThreads and Qthreads), manual cut-off threshold ("-x"), input file ("-f"), and problem size ("-n"). The experiment machine was a 2.30 GHz Xeon E5-2699 v3 (Haswell) server with 36 cores (two 18-core sockets). We analyze the executions of each benchmark run by each system using 36 workers (threads) on full 36 cores. Besides, each benchmark has one baseline

Table 8.1: BOTS benchmark arguments

App	task stack	cutoff	other args
Alignment	2^{20}	-	`-f prot.100.aa`
FFT	2^{15}	128	`-n` 2^{24}
Fib	2^{15}	manual	`-n 47 -x 19`
Floorplan	2^{17}	manual	`-f input.20 -x 7`
Health	2^{14}	manual	`-f medium.input -x 3`
NQueens	2^{14}	manual	`-n 14 -x 7`
Sort	2^{15}	manual	`-n` 2^{27} `-a 512 -y 512`
SparseLU	2^{14}	-	`-n 120 -m 40`
Strassen	2^{14}	manual	`-n 4096 -x 7 -y 32`
UTS	2^{14}	-	`-f tiny.input`

version, which elides all task creations and synchronizations and runs serially on one thread. Note that this is different from running a parallelized program on a single core, which still incurs some task management overheads; our baseline is a serial program that has no such overhead.



(a) SparseLU (b) Alignment (c) FFT

Figure 8.1: SparseLU, Alignment, FFT: performance loss breakdowns of executions on 36 cores by five systems MassiveThreads, Cilk Plus, TBB, OpenMP, and Qthreads. The *serial work* has been subtracted from all bars. The percentage represents the proportion of the remaining surplus amounts against the serial work. Although the serial version does not include task management overhead, but it includes the overhead of our DAG-recording measurement. Serial version's delay represents this tracing overhead.

8.1.1 SparseLU

SparseLU performs the LU decomposition of a sparse matrix. With problem size of $n = 120$ which our experiment used, the benchmark consists of 120 phases each of which includes two consecutive for loops. The loops are written such that each iteration creates a task. The first loop can create up to $2 * (n - 1) = 238$ tasks (iterations), the second loop can create a large number of (up to $(n - 1)^2 = 14161$) tasks (iterations). Each task performs a leaf computation which does not recursively spawn any further children. Hence, only the parent task can spawn more parallelism; the computation's available parallelism depends totally on the progress of the parent task's loops.

Work-first schedulers like MassiveThreads and Cilk Plus which immediately switch to the new child upon a task creation, leaving current parent back to ready queue, are disadvantageous in this kind of parallelization patterns. At every loop iteration, the parent task is left back to queue to be stolen.

There is only one ready parallelism at a time which is the waiting parent task (the thin magenta line above the red areas of Cilk Plus and MassiveThreads in Fig. 8.2). A free worker doing work-stealing needs to pick the right victim among a large number of workers available in order to successfully acquire the parent task, to resume the loop and spawn more parallelism. In parent-first schedulers like TBB, OpenMP, and Qthreads, the parent task (and its loop) is kept executing continuously on one worker without interruption; so ready parallelism increases fast and abundantly (blue areas of OpenMP, Qthreads, and TBB in Fig. 8.2, truncated in the figure as it does not fit otherwise). However, these ready tasks are residing solely on one worker (the one that is executing the parent task). A free worker doing work-stealing still needs to pick the right victim among others in order to get a task to execute.

Regardless of work-first or parent-first, after finishing a task, a worker performing a work-stealing operation needs to find the sole worker who holds the ready parallelism(s) (i.e., either the single parent task or all the spawned children) in order to get a new task. Unlike a recursive algorithm in which ready tasks are scattered among multiple workers, in SparseLU, only one worker holds all available ready tasks, and a free worker needs to pick the right one among many possible victims to steal work successfully. The intervals of a worker doing work-stealing are represented by white spaces between task boxes on the timeline of that worker in Fig. 8.2. Dense timelines with small white spaces imply fast work-stealing speed, while sparse timelines with many large white spaces imply slow work-stealing speed. Looking at Fig. 8.2, we can understand that Cilk Plus and TBB have slow work-stealing performance, resulting in sparse timelines. MassiveThreads, which implements work-first policy, nevertheless has dense timelines due to its fast work-stealing implementation. OpenMP also has impressively dense timelines, implying its fast work-stealing.

Those white-space delays incurred when workers switch from a task to another are accounted into the delay and no-work-sched components in the performance loss breakdown (Fig. 8.1a). Larger sums of delay and no-work-sched of Cilk Plus and TBB compared with MassiveThreads and OpenMP represent their slower task stealing speed. Cilk Plus has large no-work-sched because during white intervals there is only one ready parallelism which is the parent task, causing no-works on many free workers. TBB has a large delay because an abundant number of ready tasks have been created on a worker, waiting for being stolen and executed. The difference between MassiveThreads' delay and Cilk Plus' delay represents how much longer delays Cilk Plus scheduler imposes between iterations of the loops compared with MassiveThreads. We have made a microbenchmark for comparing work-stealing speeds of the systems (Fig. 8.3). The benchmark was run with only two workers, the parent was executed on one worker, and the only created child was stolen and executed on the other worker. The benchmark measures times from the task creation to the child's start and to the parent's resume. The result shows that Cilk Plus has significantly large work-stealing overhead (more than double other systems). Although this microbenchmark shows that TBB has equivalent work-stealing speed with MassiveThreads and OpenMP, in the real execution environment TBB may suffer from other factors such as larger numbers of available workers, more contentions on the worker holding tasks.

Qthreads has one special characteristic in its scheduling policies. It does not make child tasks available for other workers to steal immediately when they are spawned, but deliberately delays them until a later time, e.g., when the current worker enters a synchronization primitive. In the case of SparseLU's loop parallelism, it is until a task-generating loop finishes and the worker enters a `WaitTasks` following the loop. This policy has caused an outstandingly large delay component in the performance loss breakdown (Fig. 8.1a), and a noticeably large white space in timelines (Fig. 8.2). For a task-generating loop like this, Qthreads accumulates tasks and schedules them collectively at once; it tends to assign tasks from consecutive iterations to one worker. This behavior is shown clearly by DAG visualizations in Fig. 8.4 (colors denote workers; only the first loops are shown; loop iterations are tiny, nearly invisible, nodes on the spine). While TBB and OpenMP migrate children one by one to random workers (indicated by random colors along the horizontal axis) due to the random work-stealing, Qthreads have children from consecutive iterations executed by the same worker (indicated by boxes of the same color next to each other). Maybe because of this task grouping, Qthreads could achieve a very good locality; it executed work quickly and kept work stretch prominently low compared with other systems (short task nodes in Fig. 8.4c and low red part in Fig. 8.1a), well compensating for its large deliberate delay.

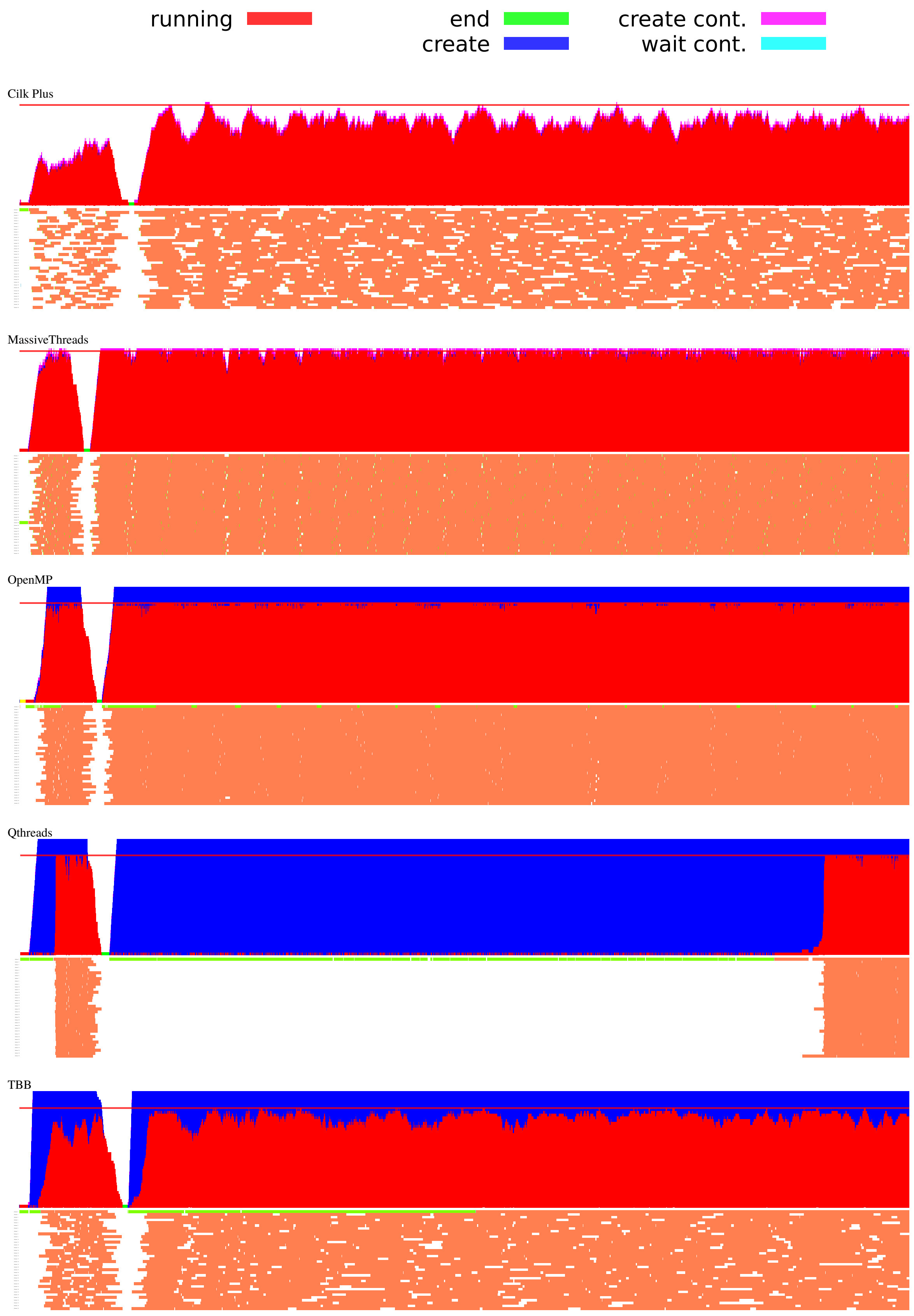


Figure 8.2: Timelines and parallelism profiles of SparseLU benchmark run by 5 systems. In each figure, the lower half is timelines and the upper half is parallelism profile (running parallelism in red color, and ready parallelism in other colors). For viewability, only the first loop and a part of the second loop in the first phase (among 120 phases) are included in x-axis; and y-axes of TBB, OpenMP, and Qthreads are truncated at top because their ready parallelisms get too high.

```
      volatile int parent_to_child;
      volatile int child_to_parent;

      void child() {
        $t_c$ :
        child_to_parent = 1;
        while (parent_to_child == -1) {}
      }

      void parent() {
        parent_to_child = -1;
        child_to_parent = -1;
        $t_0$ :
        CreateTask( child() );
        $t_p$ :
        parent_to_child = 1;
        while (child_to_parent == -1) {}
        WaitTasks;
      }

      void main() {
        for (i = 0; i < 100000; i++)
          parent();
      }
```

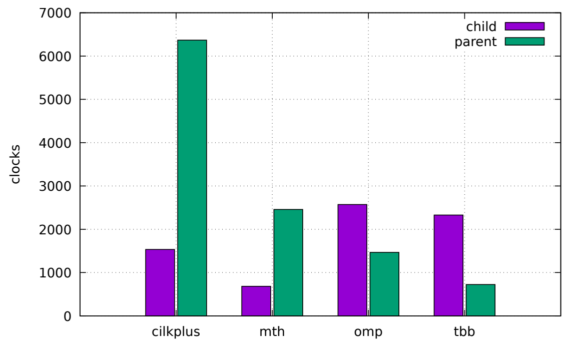
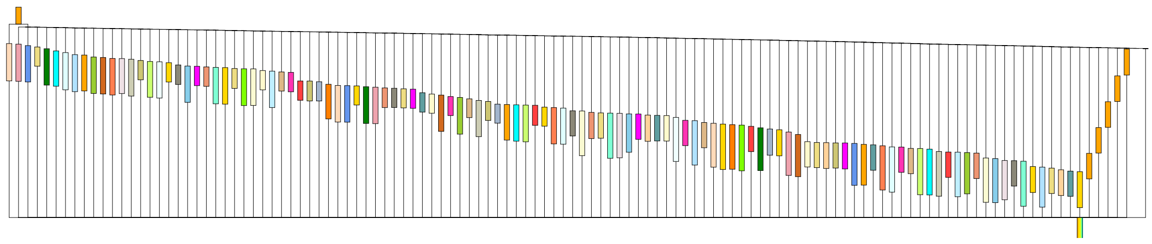
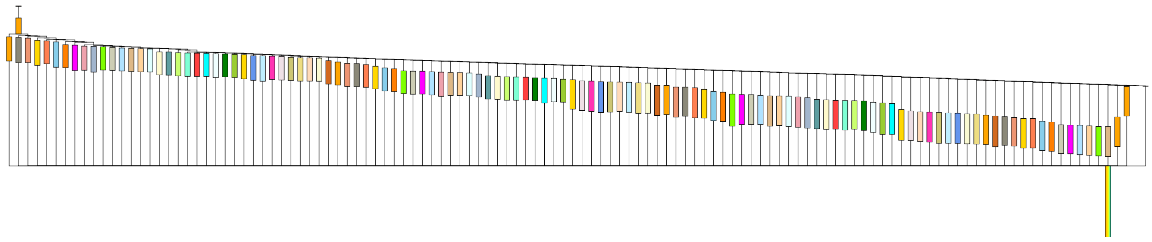


Figure 8.3: A microbenchmark for testing work-stealing speed of the systems: *child* is the time from t_0 to t_c, *parent* is the time from t_0 to t_p. The benchmark is run with only two workers. Qthreads does not run with this benchmark because it delays the child's execution until WaitTasks.


(a) TBB: slower progress than OpenMP, many tasks get long.


(b) OpenMP: faster progress than TBB, task sizes are more uniform.

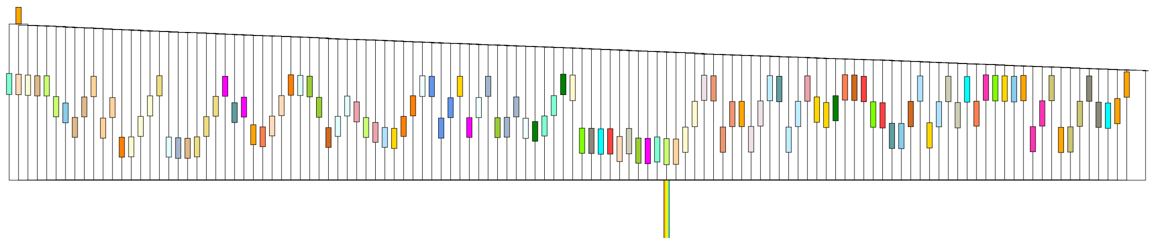

(c) Qthreads: all children are delayed, then assigned in groups of consecutive tasks to other workers.

Figure 8.4: DAG with timing on y-axis of the first loop of SparseLU. TBB and OpenMP implement the typical principles of work stealing: LIFO (last in first out) local task execution, and FIFO (first in first out) remote task stealing, a created task is made available for stealing asap; whereas Qthreads delays children until a later time when it can group and assign collectively multiple consecutive tasks to each worker. Despite wasteful delays on free workers, Qthreads tends to have less work stretch owing to possibly better task localities.

8.1.2 Alignment

Alignment benchmark aligns $n = 100$ protein sequences against each other. Every alignment of a pair of sequences is scored and the scores of all $n \times (n-1) \div 2 = 4950$ alignments are returned. The benchmark's parallelization model is simple; it consists of only one for loop of 4950 iterations each of which creates a serial child which computes an alignment and does not spawn any more child. There is a degree of load imbalance in the children: some are short, but some are very long. However, because the number of tasks is abundant, there should not be any issue of insufficient parallelism in the program.

Alignment has the same parallelization model as SparseLU does: loops create one task in each iteration. But because Alignment has only one loop (compared with SparseLU's $2 \times n = 240$), and many children are long (i.e., less frequent task switches), the inefficiencies which happened in SparseLU such as slow work-stealing (Cilk Plus and TBB), large initial delay (Qthreads) have been compensated, and become insignificant in Alignment. Performance loss breakdowns in Fig. 8.1b show that delay and no-work-sched are small for MassiveThreads, Cilk Plus, TBB, and Qthreads.

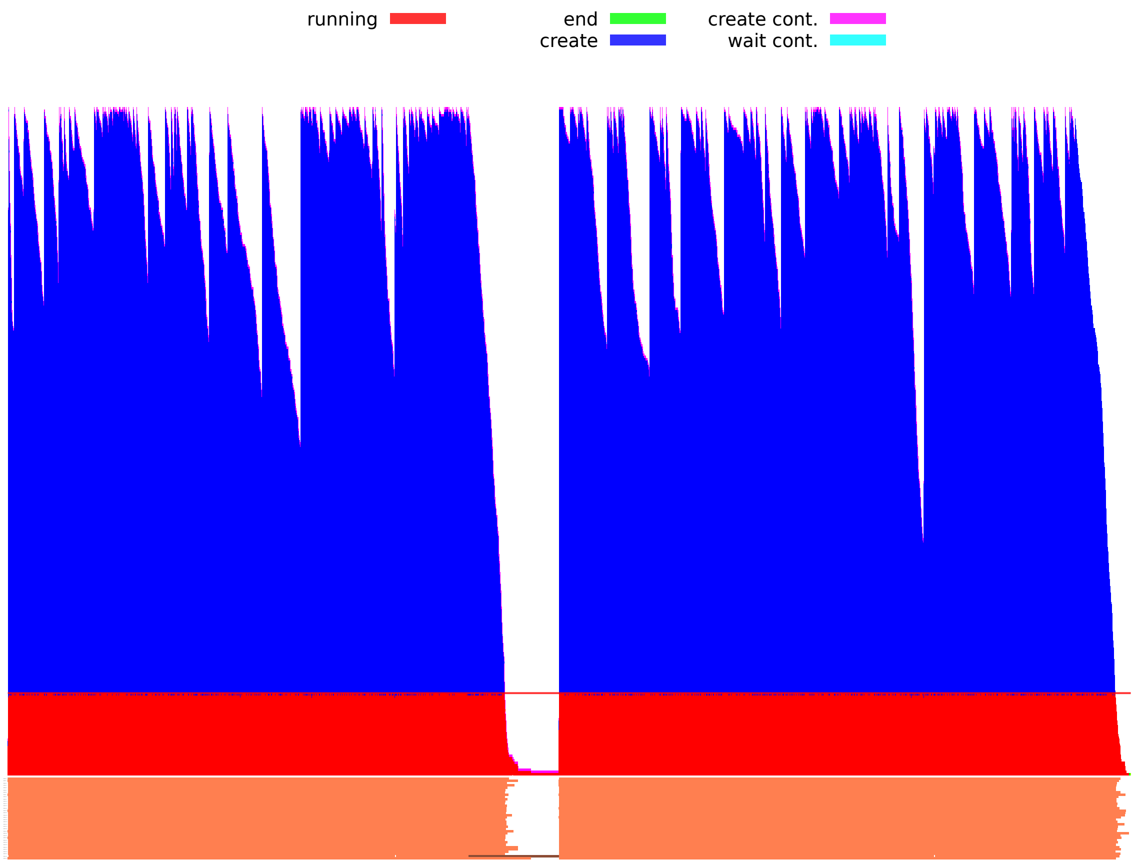
However, OpenMP does have a significantly large no-work-sched. Timelines visualization in

Fig. 8.5a has revealed that there was a noticeable period of white area in the middle of the timelines. A closer zoom into this spot on the DAG visualization is shown in Fig. 8.5b, in which the y-axis denotes the exact timings of nodes based on their start and end times. All tiny nodes on the spine of the DAG are loop iterations each of which spawns a child task to the left edge, and continues to the next iteration to the right edge. The orange worker to which the parent task is tied executes all these iteration nodes. However, unlike TBB and Qthreads, the worker does not keep executing the whole loop continuously until it finishes, but usually suspends the parent task in the middle, to switch to the immediately created child, after which it gets back to the parent task. This behavior is represented by some orange nodes occasionally scattered closely to the spine. The reason of this situation is because OpenMP limits the number of ready tasks in the queue. When the limit is reached, OpenMP tends to suspend current task, pick and execute a child from the queue, and get back to the suspended task after finishing the child.

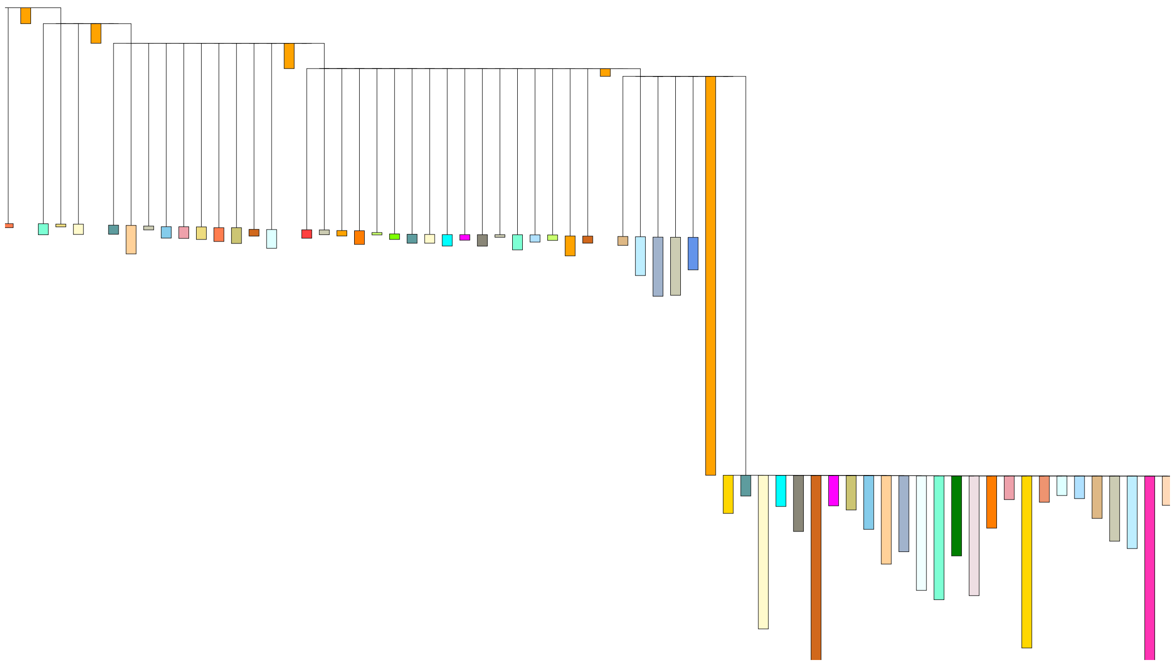
The problem here is that the tied orange worker happened to execute a long child in the middle of the computation (long orange node at center of the graph in Fig. 8.5b, and dimmed long node in the middle of the timelines in Fig. 8.5a). While the worker was busy running that long task, other workers finished their work and waited in idle because there was no more available tasks to steal. The parent task which was the only one who could spawn more parallelisms to feed free workers was unfortunately tied to the busy worker, unable to be stolen and migrated to another worker.

The problem was caused by an intricated combination of the following four factors; two residing in OpenMP: (1) tasks are tied, (2) a task queue has an upper bound in size; and two in the Alignment: (3) loop parallelism, (4) unbalanced load among iterations. TBB also has the condition (1) of tied tasks, but it does not have the condition (2), so it did not suffer from this problem. Parallelism profiles of TBB and OpenMP are placed side by side in Fig. 8.6 and zoomed out to cover all the height of ready parallelism of TBB so that we can see that TBB's ready parallelism reaches its peak at around 5000 soon after the execution started, whereas OpenMP's ready parallelism always stops increasing and drops when it approaches near 300. SparseLU also has condition (3) of loop parallelism, but it does not have the condition (4); children in SparseLU are quite balanced. Therefore, OpenMP running SparseLU did not encounter this problem.

One more noticeable point from the performance loss breakdowns in Fig. 8.1b is that the no-work-app of Qthreads was large, larger than that of other systems. A closer look into the end of the timelines and parallelism profiles of Qthreads has revealed the reason. It was simply because Qthreads happened to execute long children late near the end of the computation when there was almost no work left, causing many no-works on many workers (large white-space area across multiple timelines), which were accounted into its no-work-app.



(a) Timelines and parallelism profile: a large significant white space happens at the middle of the computation.



(b) DAG: a long child happens to be executed by the worker to which the master task is tied. That long work prevents the worker from resuming the master task, so no more parallelism is created while other workers are starving for work.

Figure 8.5: Alignment by OpenMP suffers from an issue caused by OpenMP's upper-bound task queue, and Alignment's unbalanced iterations.

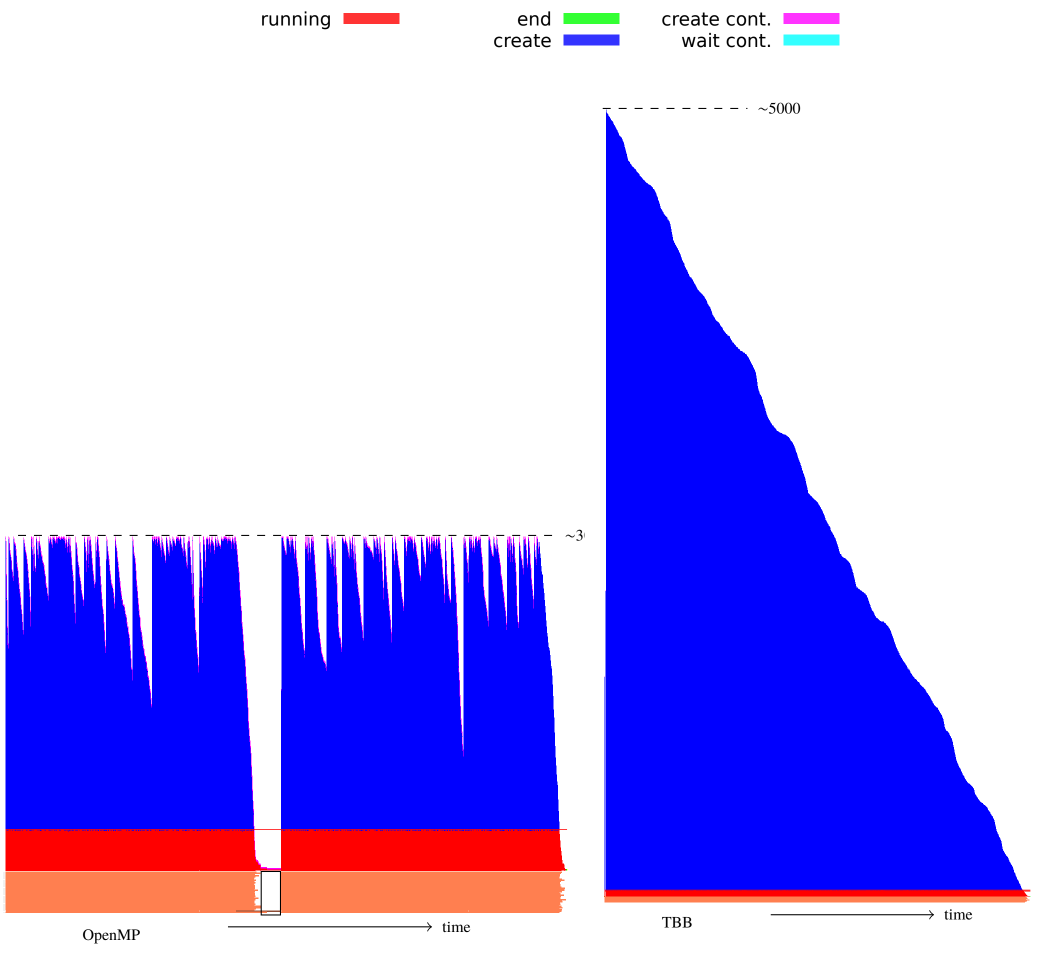


Figure 8.6: Parallelism profiles of Alignment run by TBB (left) and OpenMP (right): TBB creates all child tasks at once continuously (up to 5000 tasks at peak), whereas OpenMP suspends task creations whenever the number of ready tasks floats near 300.

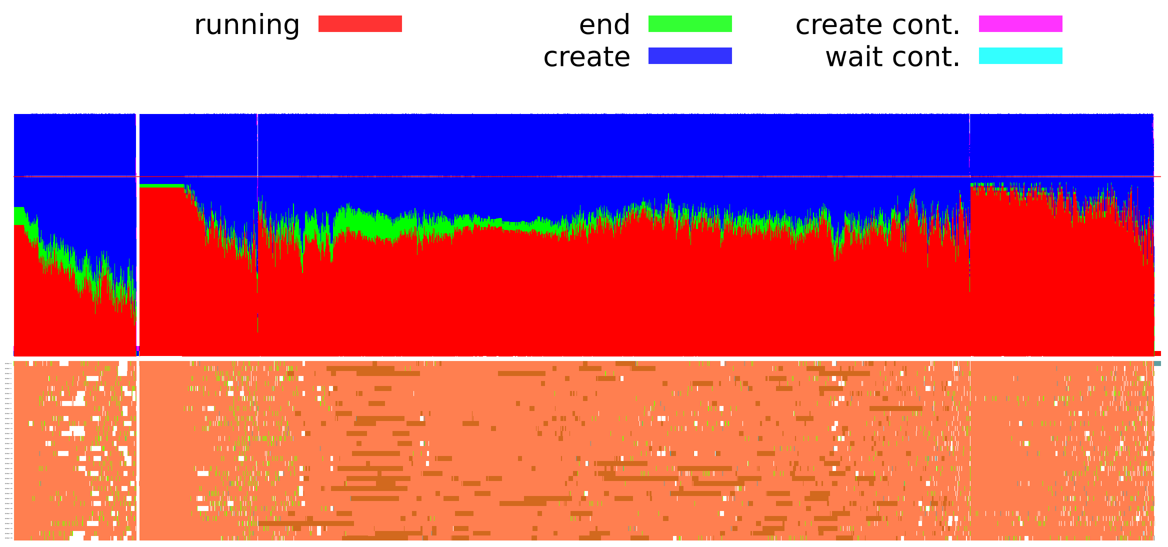
8.1.3 FFT

FFT benchmark computes the one-dimensional Fast Fourier Transform of a vector of $n = 2^{24}$ complex values in a divide-and-conquer fashion. FFT creates recursively fine-grained tasks to divide the input $n = 2^{24}$ elements into leaf tasks handling $n < 128$ elements each. Fig. 8.1c shows that OpenMP and Qthreads have significantly larger delays compared with other systems. A closer look into their timelines, parallelism profiles, and DAG visualizations have revealed that these inefficiencies originated from different, unique characteristics in their scheduling policies.

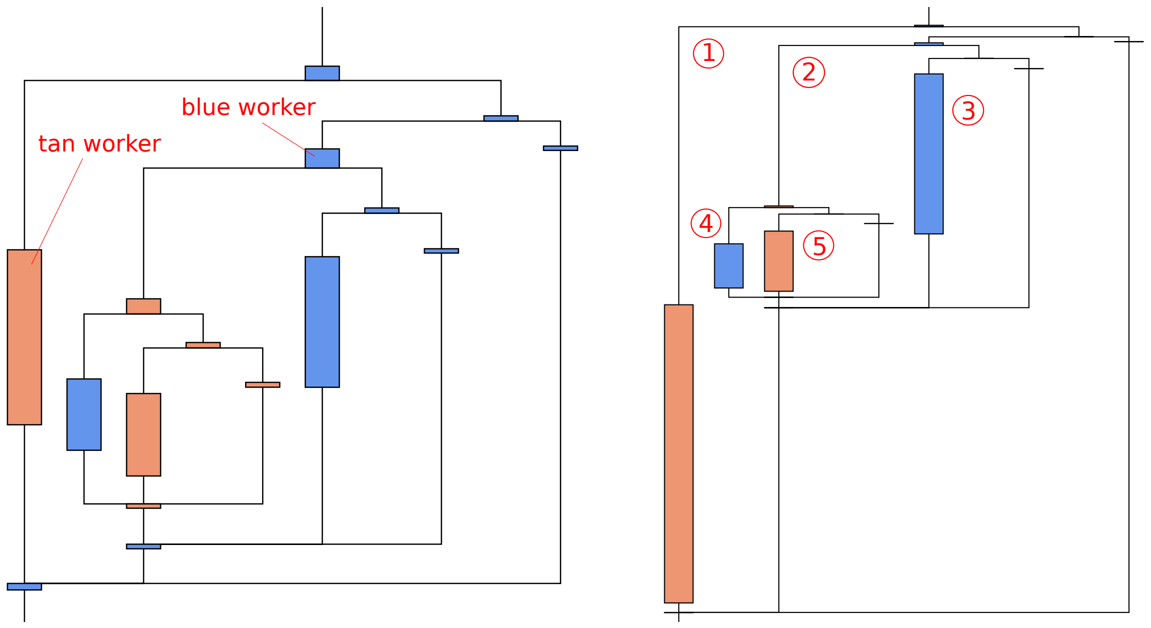
OpenMP's timelines (Fig. 8.7a) become sparser by many scattered white spaces at the end of a recursive phase. By zooming into some spots of the DAG according to those white spaces, we have observed an interesting inefficiency of OpenMP scheduler. One of the spot is shown in Fig. 8.7b in which the left and the right subfigures show the same spot just with different scales, power scale on the left for better viewability of the task structure, and linear scale on the right for displaying exact timing of task execution order. The sub-graph in Fig. 8.7b was started and also finished by the blue worker which did not execute any other sub-graph during the span of this one. After finishing the task 3, the blue worker quickly stole and executed the task 4 from the tan worker. However, the blue worker stayed idle after finishing that task 4 for a long time without stealing any work to do, just waiting in idle for the tan worker to finish the long task 1. After the tan worker finished it, the blue worker almost immediately resumed the parent task. Moreover, after finishing this sub-graph, the blue worker immediately stole and executed a task from another part of the DAG. This is to say that OpenMP deliberately restrains workers from stealing work and makes them wait in idle until they can resume and finish their current tied tasks.

Our inspection revealed a similar pattern arises in many places of the OpenMP's DAG. We presume the factor causing this issue is OpenMP's stack-overflow prevention. In order to avoid stack-overflow, OpenMP prevents workers from stealing a new task on a distant sub-graph from an arbitrary worker; it only allows a worker to steal tasks from the sub-graph that has originated from the worker itself, like the case of task 4, which was spawned from a sub-graph originating from the blue worker and now stolen and migrated back to the blue worker. After finishing this sub-graph, the stack of the blue worker was released a certain amount of space, allowing it to steal a new distinct task on an arbitrary sub-graph from any worker. Fig. 8.8 shows three more spots which represent the same pattern pervasively happening on the DAG of OpenMP. These sparse areas only appeared near the end of a recursive phase when the stacks had grown full. At the beginning of a recursive phase, timelines were always dense.

From timelines of Qthreads (Fig. 8.9a), we can see that there were many intervals during which very few tasks were running and many workers were idle, despite the abundance of ready tasks (the high blue area). These low running parallelism intervals have made up large delay and large no-work-sched components for Qthreads. Fig. 8.9b and Fig. 8.9c zoom into two spots of the DAG visualization according to two of the suspicious white-space intervals on the timelines. The problem here is the same as what happened with the SparseLU benchmark: scheduling of children is delayed until their parent task enters a synchronization primitive. In FFT's binary recursive call tree, the second recursive call was serialized as a normal function call instead of spawning a task. This made the worker who was executing the parent function call get into the execution of the second recursive call before it could reach the synchronization primitive, causing the scheduling of previously spawned children to be delayed longer, and other workers to wait longer (while there were definitely ready tasks).

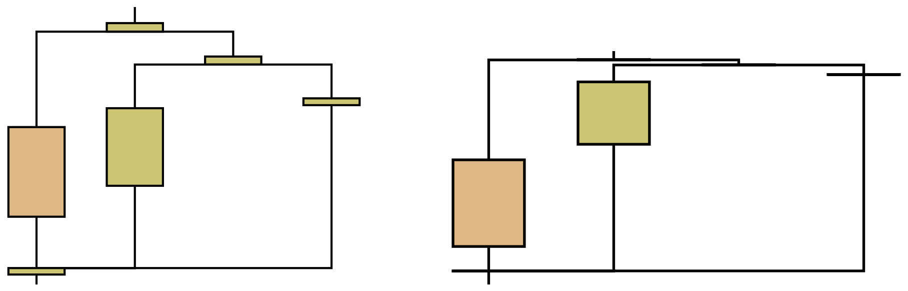


(a) Timelines (lower half) and parallelism profile (upper half): task density tends to become sparser at the end of a recursive computation phase.

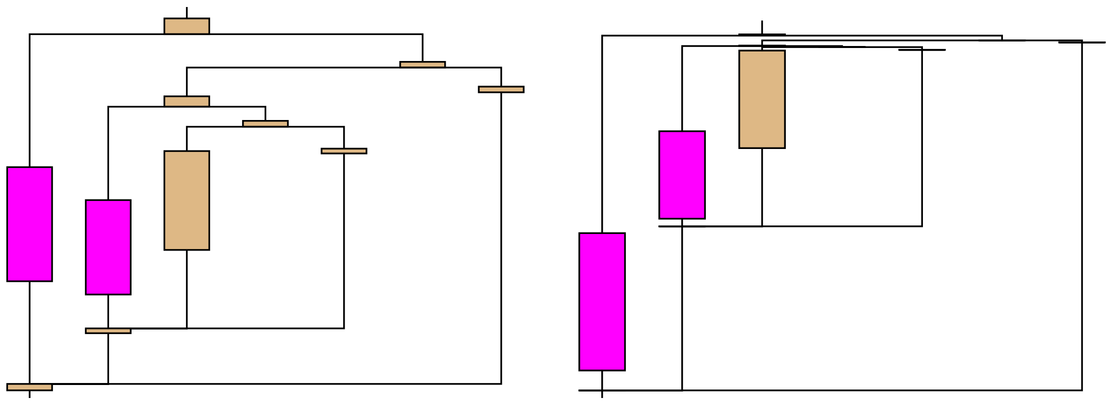


(b) A spot on the DAG (power scale on the left for better viewability, and linear scale on the right with y-axis denoting exact timing) which corresponds to one white space on the timeline of the blue worker. The blue worker stalled, avoided work-stealing, and just waited in idle for the tan worker to finish a long task; then after tan worker finished, it immediately resumed its current tied parent task.

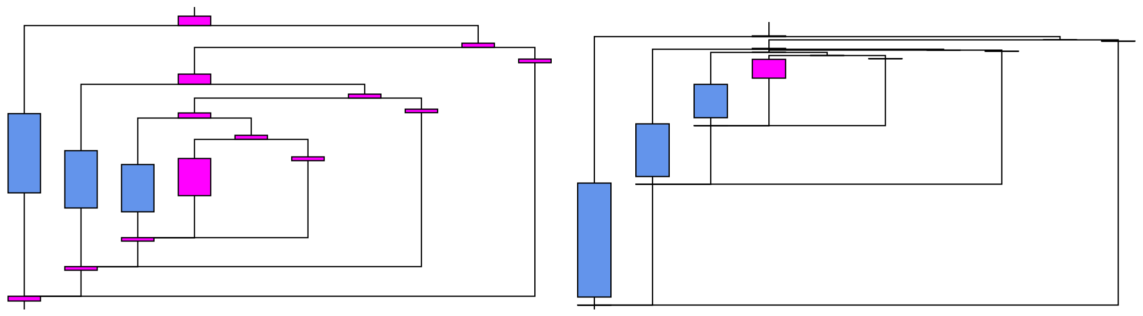
Figure 8.7: FFT by OpenMP: timelines, parallelism profile, and a zoomed-in spot according to one of the white space on the timelines.



(a) One child stolen (power scale on the left for better viewability, and linear scale on the right with exact timing on y-axis)

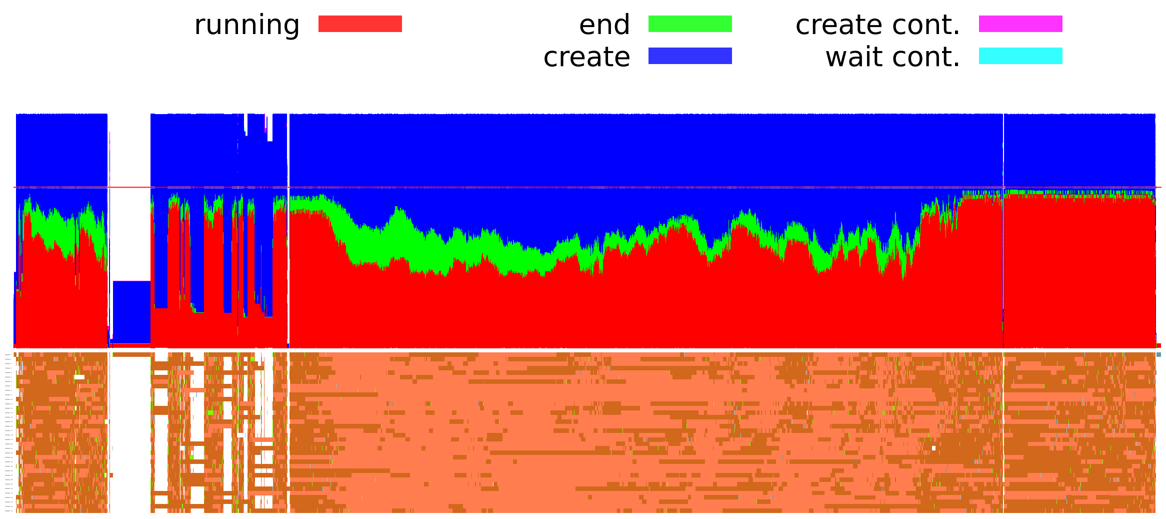


(b) Two children stolen (power scale on the left for better viewability, and linear scale on the right with exact timing on y-axis)

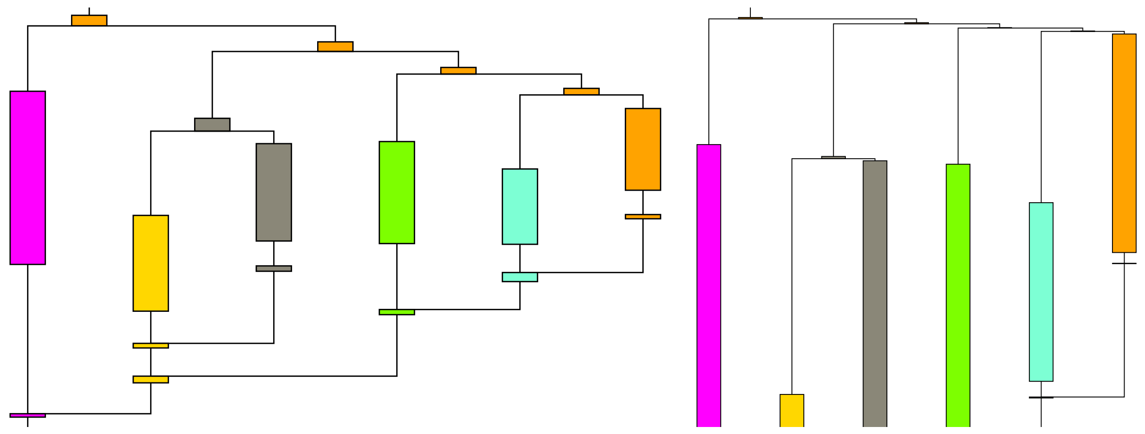


(c) Three children stolen (power scale on the left for better viewability, and linear scale on the right with exact timing on y-axis)

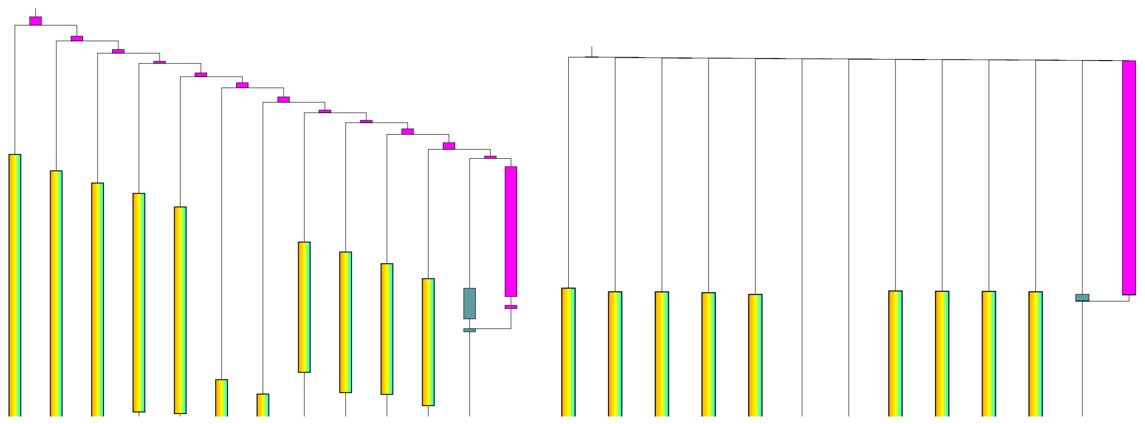
Figure 8.8: Delay examples of FFT by OpenMP: a worker executing a parent task waits idly (without doing work-stealing) for each of one, two, or three children, which have been stolen, to be finished before resuming the parent and synchronizing each of them. These delay patterns occur pervasively on DAG.



(a) Timelines (lower half) and parallelism profile (upper half): there are awkward intervals of few running workers (low red area) despite many available ready tasks (high blue area).



(b) Four recursive tasks are delayed while the worker gets into the execution of the last leaf one (power scale on the left, and linear scale on the right)



(c) Twelve recursive tasks are delayed while the worker gets into the execution of the last leaf one (power scale on the left, and linear scale on the right)

Figure 8.9: FFT by Qthreads: delayed scheduling of ready children has made workers idle wastefully. The situation has been made worse by FFT's binary recursive call tree which serialized the second recursive call instead of creating a task, this has delayed the worker more until it can reach the synchronization primitive. Figures 8.9b and 8.9c are two example spots.

8.1.4 Other benchmarks

In other benchmarks (Fibonacci, Floorplan, NQueens, Sort, and Strassen in Fig. 8.15), OpenMP has large delays in Fibonacci, Floorplan, and Sort, which are recursive algorithms with fine-grained grain sizes, due to the same reason as in FFT. OpenMP has workers avoid performing work-stealing at deep recursions in order to prevent stack-overflow, hence causing larger delays than other systems. Floorplan, Sort, and Strassen incur almost equivalent no-work-app across all systems, indicating an insufficient parallelism issue inside the applications. The visualizations revealed that Floorplan has long serial sections at the end of the computation, Sort lacks parallelism in its later half of the computation, Strassen has long serial sections at both the start and the end of its computation.

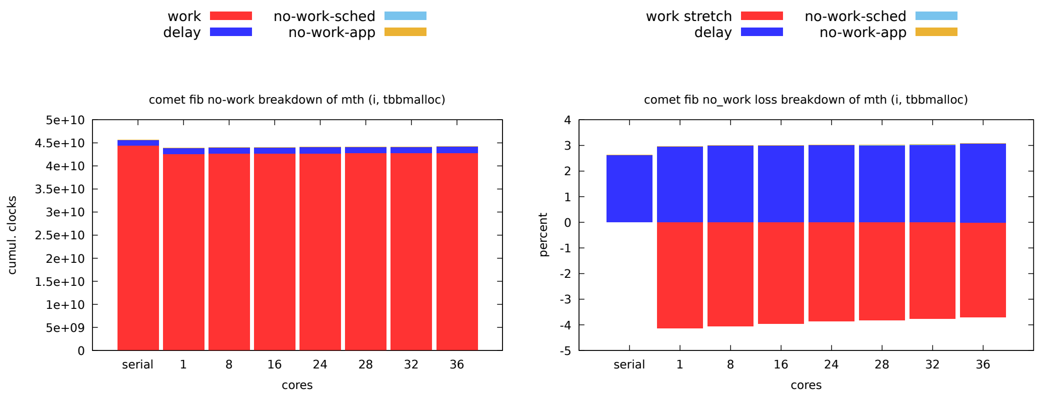


Figure 8.10: Fib: performance loss breakdown of 36-core executions run by MassiveThreads, Cilk Plus, TBB, OpenMP, and Qthreads.

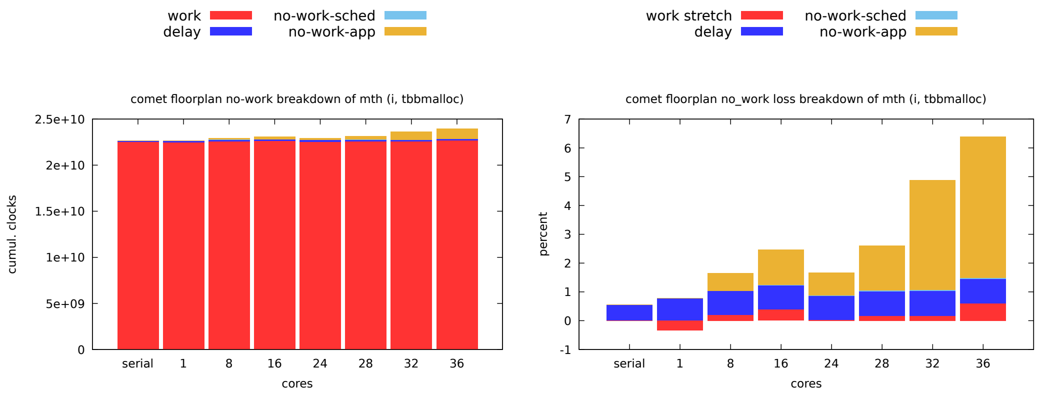


Figure 8.11: Floorplan: performance loss breakdown of 36-core executions run by MassiveThreads, Cilk Plus, TBB, OpenMP, and Qthreads.

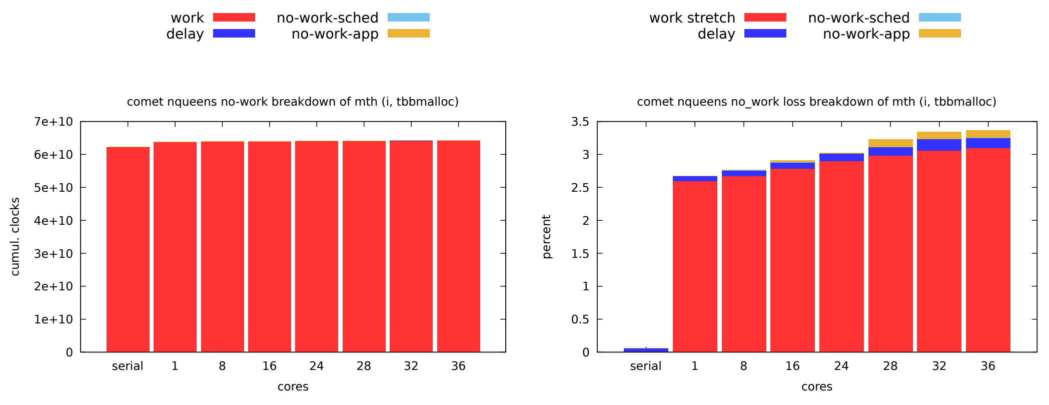


Figure 8.12: NQueens: performance loss breakdown of 36-core executions run by MassiveThreads, Cilk Plus, TBB, OpenMP, and Qthreads.

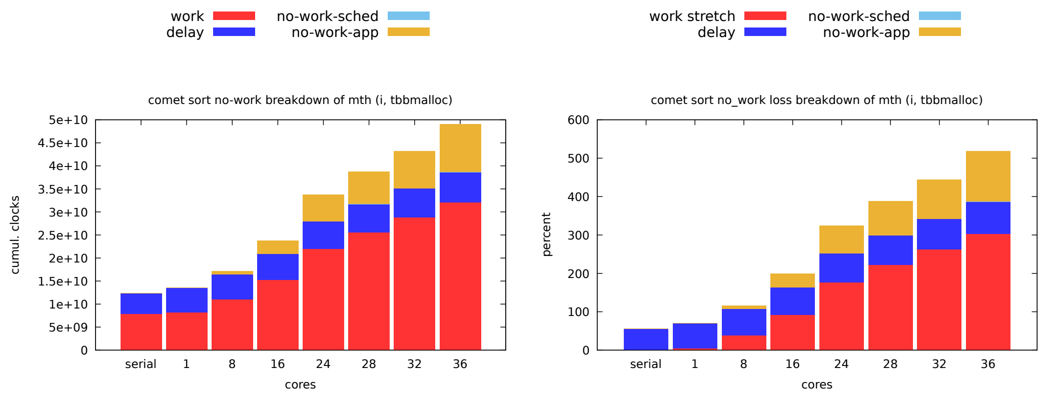


Figure 8.13: Sort: performance loss breakdown of 36-core executions run by MassiveThreads, Cilk Plus, TBB, OpenMP, and Qthreads.

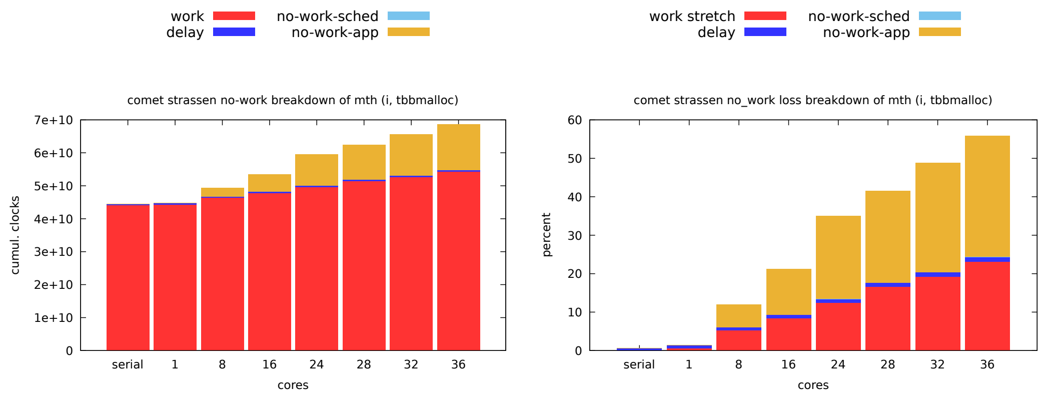


Figure 8.14: Strassen: performance loss breakdown of 36-core executions run by MassiveThreads, Cilk Plus, TBB, OpenMP, and Qthreads.

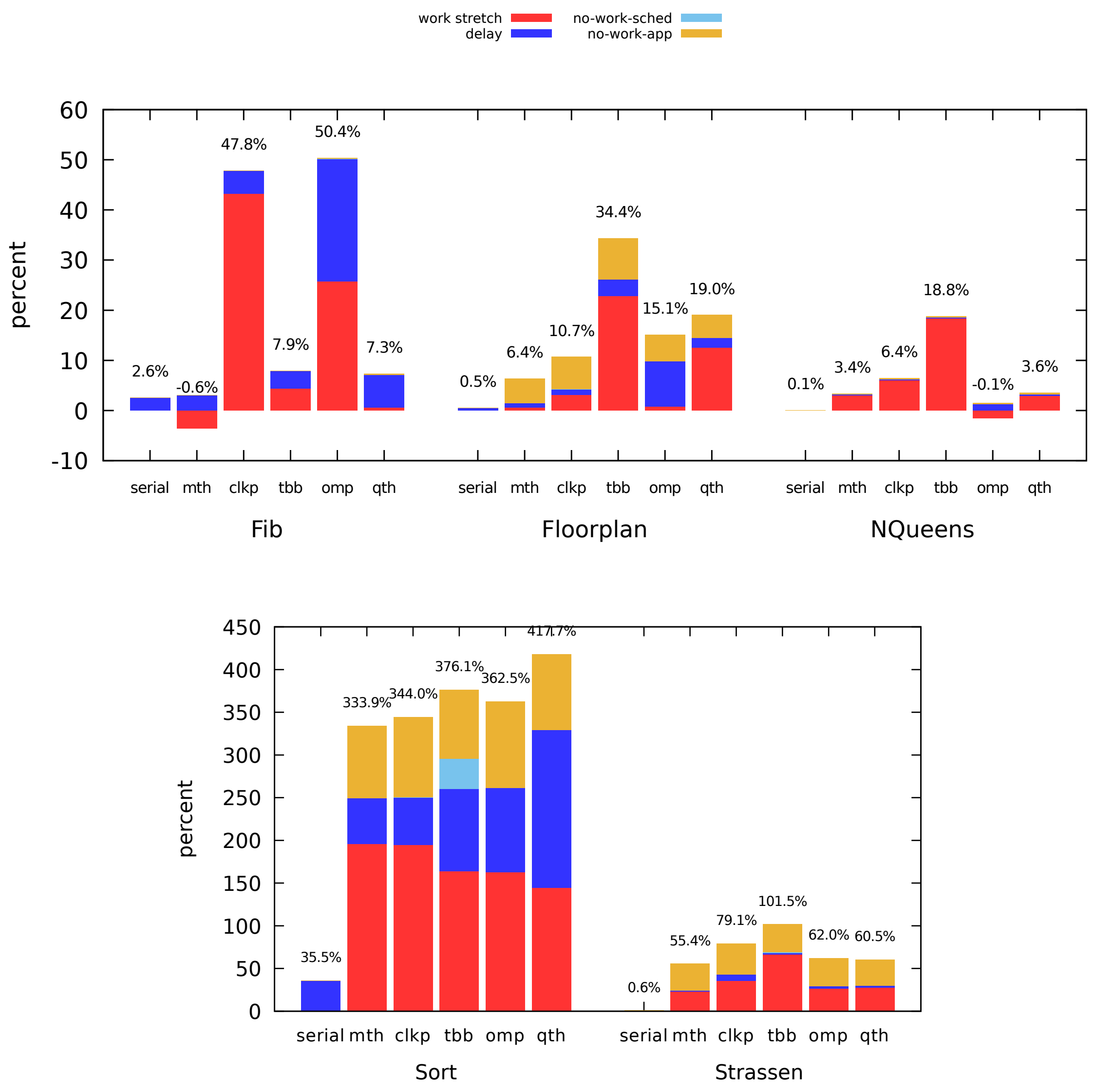
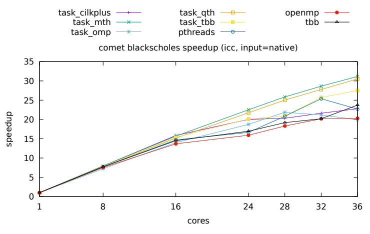


Figure 8.15: Fib, Floorplan, NQueens, Sort, Strassen: performance loss breakdowns of executions on 36 cores by MassiveThreads, Cilk Plus, TBB, OpenMP, and Qthreads.

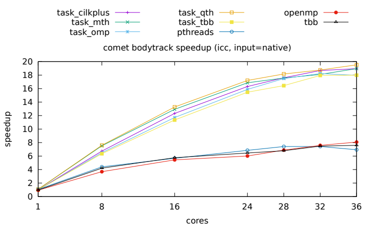
8.2 TP-PARSEC

We have evaluated TP-PARSEC on a 36-core dual-socket Haswell system equipped with two Intel Xeon E5-2699 v3 2.30 GHz. It has 768 GB of memory and runs Ubuntu 16.04.2 with kernel version 4.40-64. We use Intel C++ Compiler (icc) 17.0.1, MassiveThreads 0.97, Qthreads 1.11, TBB (2017 Update 1) in this evaluation. We measure times of only the region of interest (ROI) in each benchmark, excluding the uninteresting initialization and finalization at the beginning and the end of each one. These regions of interest which are the actual parallelized parts of each benchmark are predefined in PARSEC. All benchmarks are executed using the largest input set (*native*), partly in order to avoid chances of bottlenecks caused by the lack of work which usually happen when running on a large number of cores. The speedup results of 11 benchmarks with all original versions and task versions are shown in Fig. 8.16. In general, the task versions perform equivalently and sometimes better than the original versions.

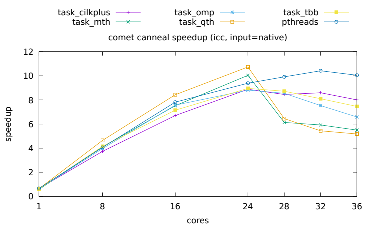
We have adjusted actual threads used in dedup and ferret. With the input number of threads n, dedup and ferret actually deploy n threads for each of their pipeline stages (except first and last ones, dedup: $1 \rightarrow n \rightarrow n \rightarrow n \rightarrow 1$, ferret: $1 \rightarrow n \rightarrow n \rightarrow n \rightarrow n \rightarrow 1$). In their speedup figures (Fig. 8.16d, Fig. 8.16f), we have adjusted their thread counts to the actual number of threads created, i.e., $3 \times n + 2$ for dedup, and $4 \times n + 2$ for ferret. Following we discuss some of the performance details we have observed in the benchmarks.



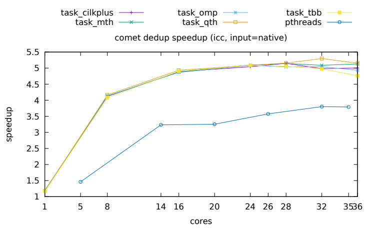
(a) blackscholes



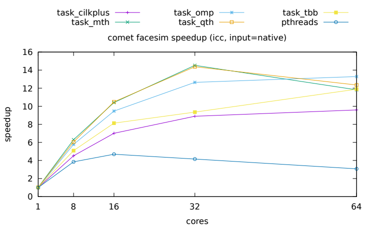
(b) bodytrack



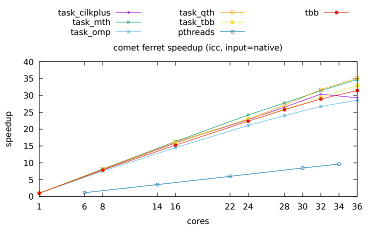
(c) canneal



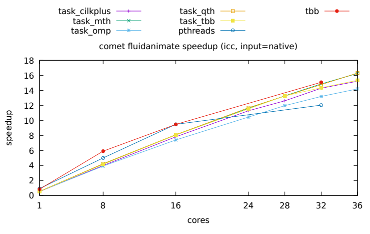
(d) dedup



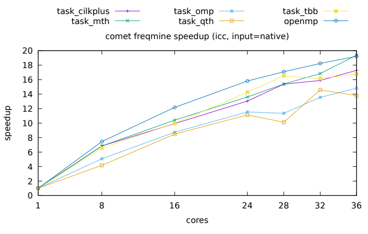
(e) facesim



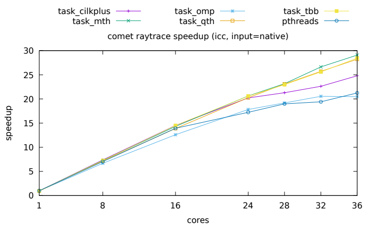
(f) ferret



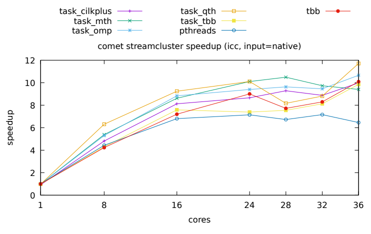
(g) fluidanimate



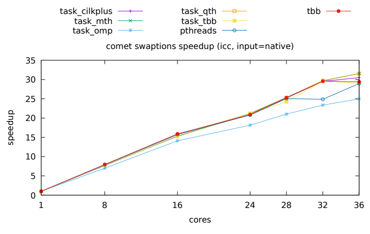
(h) freqmine



(i) raytrace



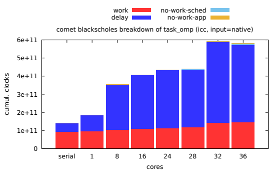
(j) streamcluster



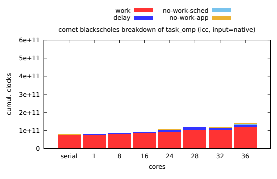
(k) swaptions

Figure 8.16: Speedups of all versions of all benchmarks in TP-PARSEC

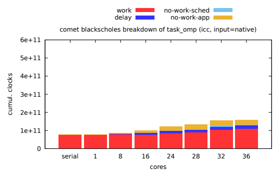
8.2.1 Setting a good grain size with the delay metric (blackscholes)



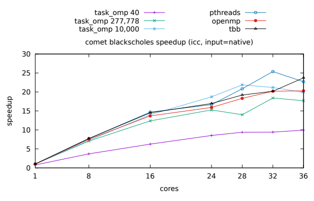
(a) grain size = 40



(b) grain size = 10000



(c) grain size = 277778



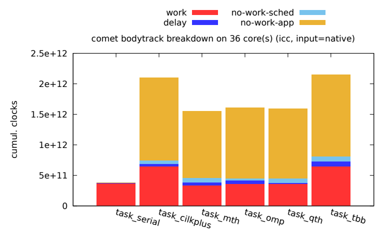
(d) with multiple grain sizes

Figure 8.17: Blackscholes: task_omp's breakdowns and speedups with multiple grain sizes

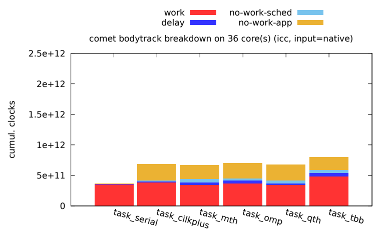
One pitfall of task parallel programming models is that creating too many fine-grained tasks will incur a very large overhead. When first translating blackscholes, we were not very aware of the number of iterations of the loop and workload of each iteration, we set the grain size of `pfor` at a random value 40. It turned out 40 was too tiny for blackscholes, causing the benchmark to perform poorly. At first we had no clue to explain this bad speedup, then the cumulative execution time breakdown produced by the performance tool (Fig. 8.17a) revealed the reason: a huge delay incurred in task versions (we show the results of task_omp because it has the largest delay, other systems incur around a half of it). We right away noticed about the grain size and tried to adjust it to a better value. We first changed it to the same value as in Pthreads version's SPMD model: 277778 ($= \lceil \frac{10^7}{36} \rceil$) which was iterations divided by the number of threads (N/P). Fig. 8.17c shows the breakdown with this grain size; delay has reduced considerably; however, no-work-app has instead increased. Anticipating this no-work-app increase is the result of coarse-grained tasks, we have decreased the grain size. After trying with many values, we got the best at around 10000, whose breakdown was shown in Fig. 8.17b: minimal delay, minimal no-work-app. Fig. 8.17d shows the speedups of original versions together with task_omp version at three different grain sizes. This is a demonstration for the intense affect that task granularity may have on the performance, and our performance tool, specifically the scheduling delay metrics, helps effectively in signaling it.

8.2.2 Overlapping I/O and computation easily with tasks (bodytrack)

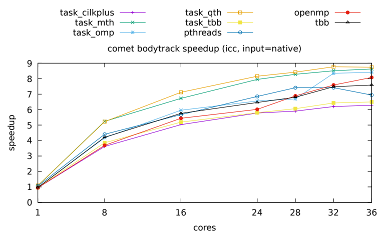
We first implemented task versions similarly to OpenMP version: no parallelism other than the five parallel loops (ver. 1). The speedup results were similar to that of the original versions, at around 8x (Fig. 8.18c); the execution time breakdown had large no-work-app (Fig. 8.18a) due to long serial execution intervals which can be observed in the timelines and parallelism profile visualizations in Fig. 8.18d. Realizing the critical impact of serial *read* and *write* stages, we have tried to overlap them with the computation (*process* stages), as described in the previous section (ver. 2). After overlapping, the results are fantastic; speedups increase 2.5 times up to around 20x (Fig. 8.16b); no-work-app reduces considerably (Fig. 8.18b); the overlapped *read* and *write* stages can even be seen visibly in the timelines visualization (Fig. 8.18e).



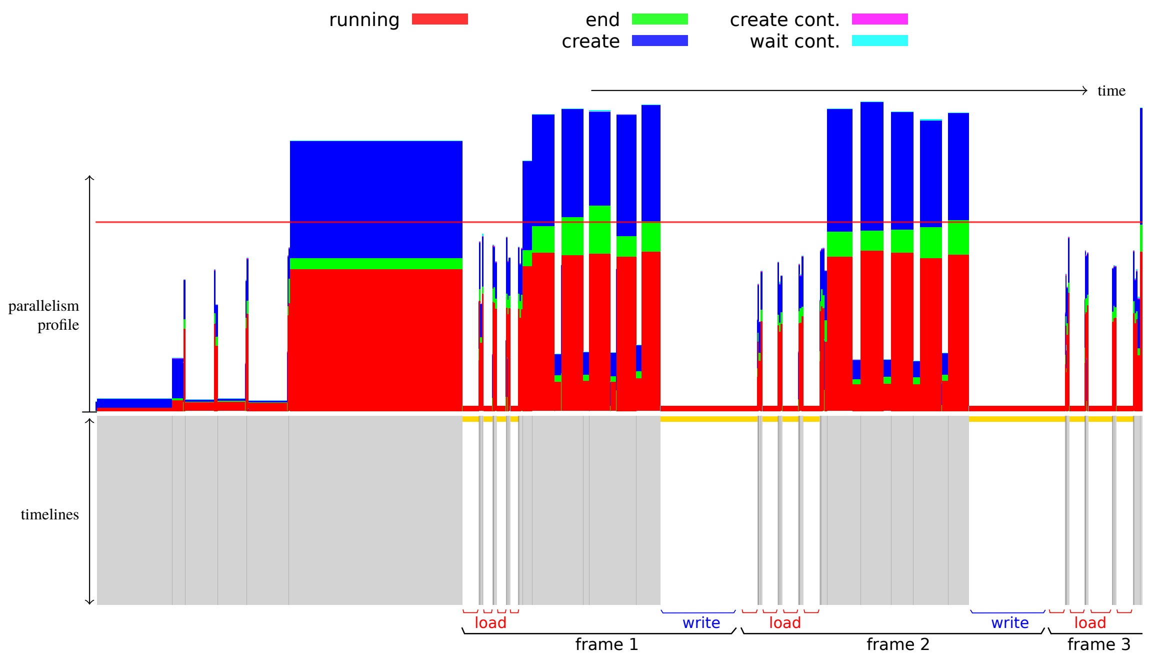
(a) bodytrack ver. 1



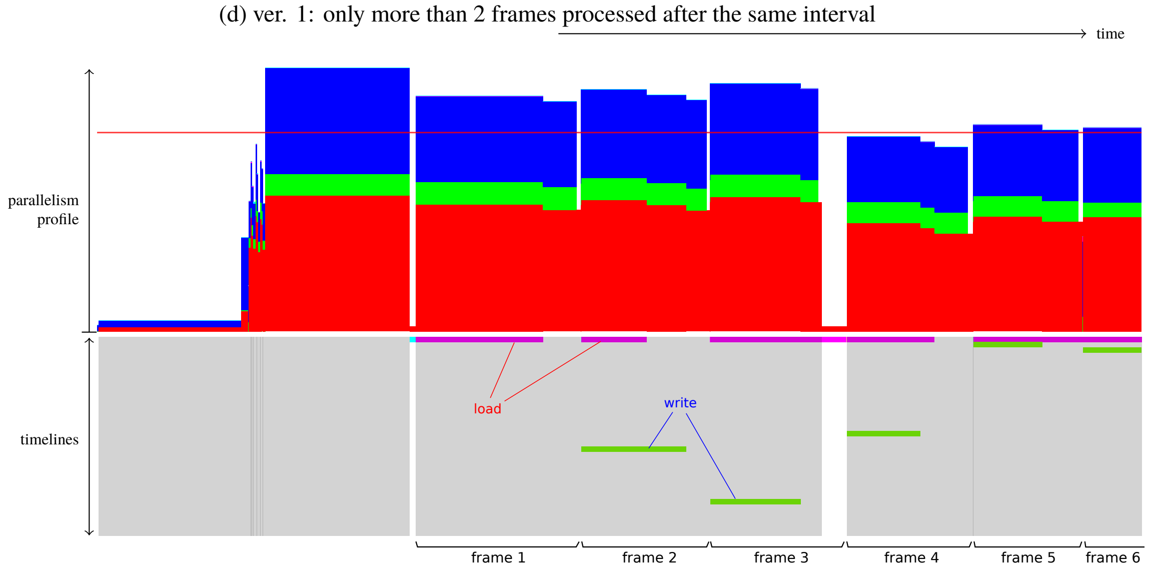
(b) bodytrack ver. 2



(c) speedup of ver. 1 (~8x) as opposed to ver. 2 (~20x) in Fig. 8.16b



(d) ver. 1: only more than 2 frames processed after the same interval

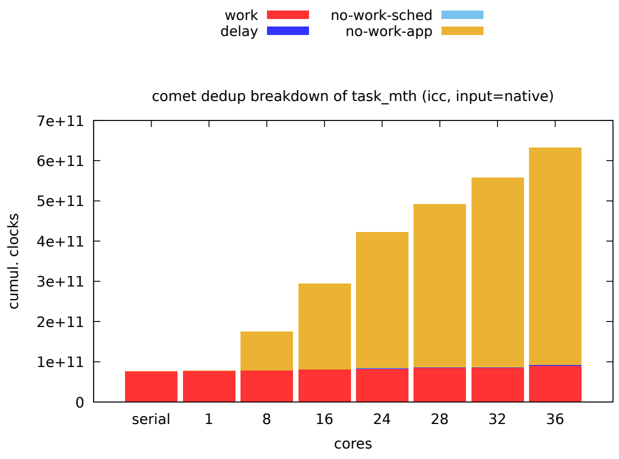


(e) ver. 2: more than 5 frames processed after the same interval

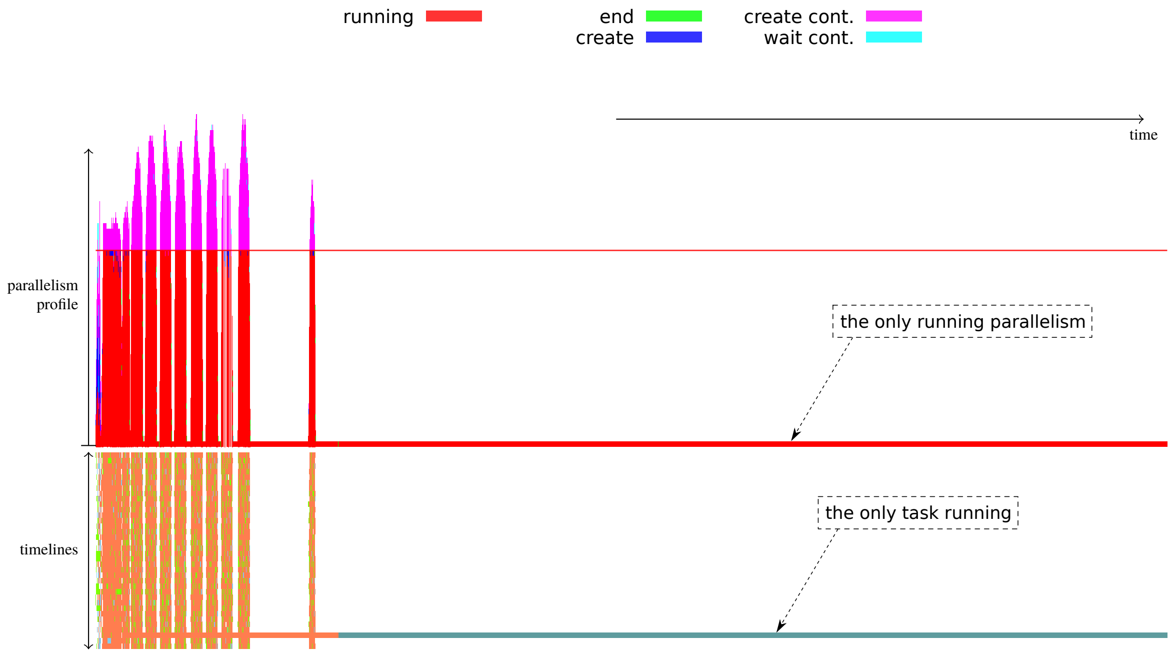
Figure 8.18: Bodytrack: ver. 2 improves substantially over ver. 1 by overlapping I/O tasks with computation tasks

Dedup suffers from the same problem with large no-work-app (Fig. 8.19a), it does not scale above

5.5x even when executed on full 36 cores (Fig. 8.16d). Its timelines visualization in Fig. 8.19b shows a very long serial interval at the end of its execution. An inspection into the code region according to the task in that serial interval has revealed the responsible instruction: file-closing function *close()*. When a file descriptor is closed, its buffer in memory gets actually flushed to disk. Dedup modified a very large buffer, so the flush takes long time. In this situation it can be said that dedup's performance is bound by disk's bandwidth.



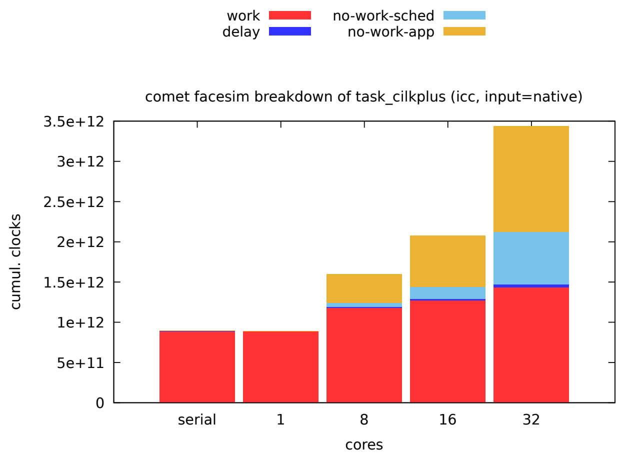
(a) large no-work-app sabotages speedup



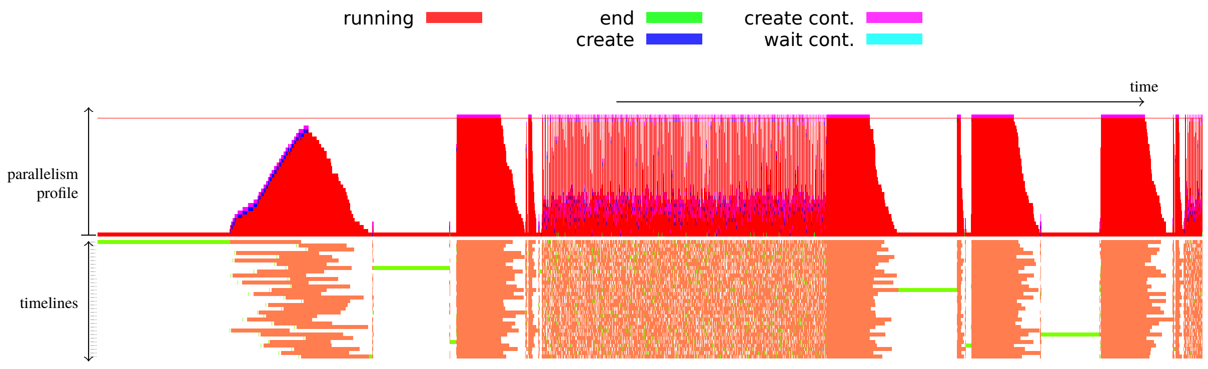
(b) a long task is noticeable on dedup's timelines

Figure 8.19: Dedup has a long serial interval at the end of the execution due to file I/O (flushing memory buffer to file).

Facesim is also having the same issue. Its timelines visualization in Fig. 8.20b shows a lot of serial not-parallelized intervals remaining in its code. These serial codes are reason for facesim's poor speedup across all versions.



(a) large no-work-app sabotages speedup

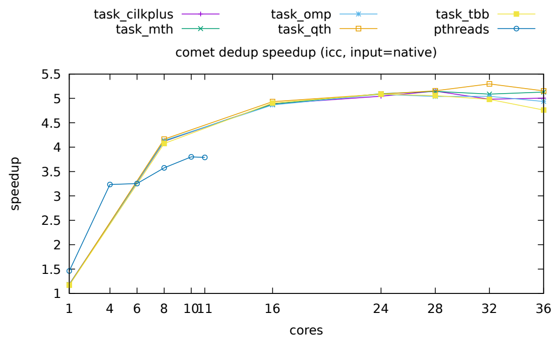


(b) many long serial intervals interleaving parallel sections in timeines of facesim

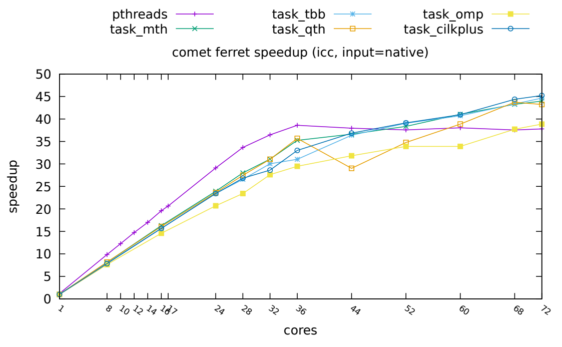
Figure 8.20: Facesim: a tiny head path of its full timelines (1.7% of the full length)

8.2.3 Adjusting actual threads used in dedup & ferret

With the input number of threads n, dedup and ferret actually deploy n threads for each of its pipeline stages (except first and last ones); dedup: $1 \to n \to n \to n \to 1$; ferret: $1 \to n \to n \to n \to n \to 1$. In their speedup figures (Fig. 8.16d, 8.16f), we have adjusted their thread counts to the actual number of threads created, i.e., $3 \times n + 2$ for dedup, and $4 \times n + 2$ for ferret. The earlier (before adjustment) speedup graphs are shown in Fig. 8.21.



(a) dedup



(b) ferret

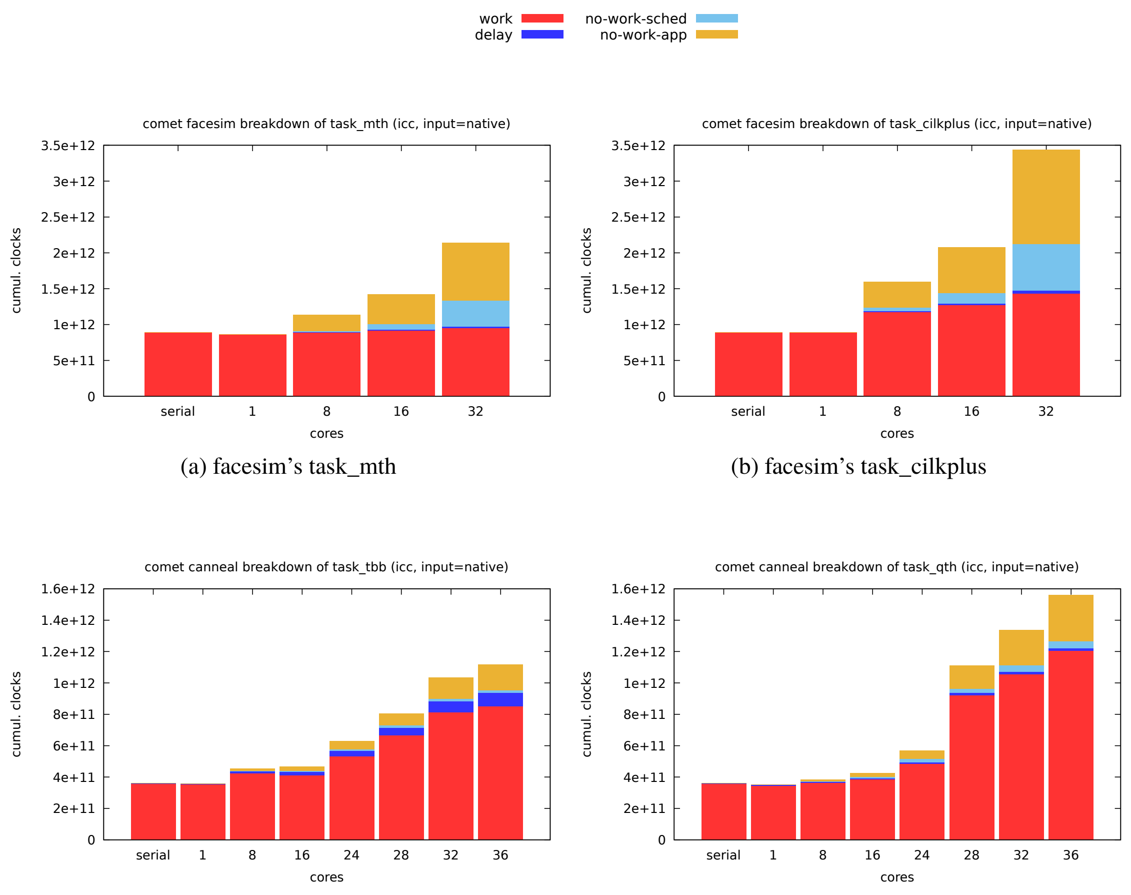
Figure 8.21: Ferret's original speedup, before fixing core counts of pthreads version.

8.2.4 Genuine task parallel schedulers are better than manual task queues (bodytrack, facesim, raytrace)

Pthreads versions of bodytrack, facesim, and raytrace bundle manually implemented task queues which basically pool a number of worker threads and assign any computation work dispatched from the main thread to them. These manual task queues simply (1) make worker threads compete via a mutex lock to get an available work (bodytrack's WorkerGroup, raytrace's MultiThreadedTaskQueue), (2) or assign work to worker threads through a basic round-robin manner (facesim's TaskQ). In task versions, we have replaced these manual task queues with the specialized task schedulers in the proper task parallel programming models. Therefore, these three benchmarks are direct showcases for demonstrating the efficiency of task parallel programming models; in bodytrack, Qthreads and MassiveThreads-based task versions perform better than the original versions (Fig. 8.18c); in facesim, all task versions perform better than the original version (Fig. 8.16e); in raytrace, all task versions except OpenMP perform better than the original version (Fig. 8.16i). A genuine task parallel runtime system usually uses work stealing technique to balance work among workers. Each worker stores ready tasks in a double-ended queue (deque) of which the local worker pushes and pops from one end, and the remote workers try stealing from the other end, hence reducing thread contentions that interfere computation progress. Recursive task creation in these task versions may also contribute partly to the efficiency thanks to its possibly better locality.

8.2.5 Characterizing performance differences with the scheduling delay-based breakdown

In some benchmarks, task versions perform similarly, speedup differences are just around 10-20% (e.g., dedup, bodytrack, fluidanimate). However, in some other benchmarks, task versions perform very differently, e.g., 42% difference in raytrace, 56% difference in blackscholes, or up to 63% difference in facesim. Especially in canneal, Qthreads performs (~22%) better than TBB does until 24 cores, but from 28 cores, its speedup suddenly degrades, falling to ~42% slower than TBB's (Fig. 8.16). Fig. 8.22 contrasts the differences between MassiveThreads vs. Cilk Plus in facesim, and TBB vs. Qthreads in canneal. Cilk Plus incurs larger no-work-sched and no-work-app compared with MassiveThreads most likely because of its slow work stealing speed which was pointed out in SparseLU benchmark in [30]. Qthreads incurs much larger work (work stretch) compared with TBB, which indicates a worsen locality of the computation execution. It is possibly because the locality-aware Qthreads scheduler has misinterpreted something when executing on larger core counts in this canneal benchmark.



(a) facesim's task_mth

(b) facesim's task_cilkplus

(c) canneal's task_tbb

(d) canneal's task_qth

Figure 8.22: Performance variation between task versions in facesim and canneal

Chapter 9

Conclusion

We introduced a useful decomposition method with ready path analysis for dividing the cumulative execution time into four components of work, delay, no-work-sched, and no-work-app. Work is the useful computation executing the application code. Delay and no-work-sched are caused by the scheduler, and no-work-app is caused by the application. Our implemented tool Delay Spotter helps users zoom into any spots of interest which cause a component to be significant so that they can pinpoint their root causes in the scheduling policy. We have successfully understood some interesting inefficiencies caused by different scheduling policies of the runtime systems in BOTS benchmarks.

Cilk Plus and TBB suffer from slow work-stealing when there is only one worker holding all the ready task(s), e.g., a parallel for loop which creates a task for each iteration.

Qthreads sometimes suffers from its delayed task scheduling approach which deliberately delays the start of children until the parent synchronizes. This scheduling policy suffers in two situations: (1) the parent is a long for loop creating a large number of children, all these children cannot start until the parent finishes the task-generating loop (SparseLU); (2) binary recursions that, rightly, spawn only the first recursive call as a task (FFT).

Carefully designed, OpenMP scheduler turned out to have performance ramifications in our experiments. It imposes an upper bound for the ready task queue of a worker; a worker is discouraged to create too many tasks ready on the queue. This approach hurts when there is only one parent creating all available children, and the worker to which the parent is tied switches to a long child, making the parent unable to be resumed. This is what was happening in the Alignment. Another characteristic of OpenMP is the stack-overflow prevention. In order to avoid stack-overflow, at deep recursions a worker restrains from stealing tasks from sub-graphs not originating from itself; instead it tends to prioritize executing current tied sub-graph and stealing only from workers who are executing a part of the same sub-graph. This is what we have observed in FFT.

Acknowledgement

本研究の一部は、文部科学省博士課程教育リーディングプログラム「東京大学ソーシャル ICT グローバル・クリエイティブリーダー育成プログラム」の支援により行われました。

Publications

International:

- (under review) A. Huynh, C. Helm, S. Iwasaki, W. Endo, B. Namsraijav, K. Taura, "**TP-PARSEC: A Task Parallel PARSEC Benchmark Suite**", IEEE International Parallel and Distributed Processing Symposium (IPDPS' 18)
- A. Huynh, K. Taura, "**Delay Spotter: A Tool for Spotting Scheduler-Caused Delays in Task Parallel Runtime Systems**", IEEE International Conference on Cluster Computing (CLUSTER'17)
- (poster) A. Huynh, K. Taura, "**Critical Path Analysis for Characterizing Parallel Runtime Systems**", ACM International Symposium on High-Performance Parallel and Distributed Computing (HPDC'16)
- A. Huynh, D. Thain, M. Pericas, K. Taura, "**DAGViz: A DAG Visualization Tool for Analyzing Task-Parallel Program Traces**", International Workshop on Visual Performance Analysis, held in conjunction with SC15 (VPA'15)

Domestic:

- (unrefereed) A. Huynh, K. Taura, "Critical Path Analysis for Characterizing Parallel Runtime Systems", Summer United Workshops on Parallel, Distributed and Cooperative Processing (SWoPP'16)
- (unrefereed) A. Huynh, D. Thain, M. Pericas, K. Taura, "Analyzing Task Parallel Program Traces based on DAG Visualization", Summer United Workshops on Parallel, Distributed and Cooperative Processing (SWoPP'15)
- (unrefereed) A. Huynh, J. Nakashima, K. Taura, "A Performance Analyzer for Task Parallel Applications based on Execution Time Stretches", Summer United Workshops on Parallel, Distributed and Cooperative Processing (SWoPP'13)
- (poster) A. Huynh, J. Nakashima, K. Taura, "A Performance Analyzer for Task Parallel Applications based on Execution Time Stretches", Symposium on Advanced Computing Systems and Infrastructures (SACSIS'13)

Bibliography

[1] Intel cilk plus. `https://www.cilkplus.org/`.

[2] Massivethreads. `https://github.com/massivethreads/massivethreads`.

[3] Umut A. Acar, Guy E. Blelloch, and Robert D. Blumofe. The data locality of work stealing. In *Proceedings of the Twelfth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '00, pages 1–12. ACM, 2000.

[4] 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. *Concurr. Comput. : Pract. Exper.*, 22(6):685–701, April 2010.

[5] Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. Thread scheduling for multiprogrammed multiprocessors. In *Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '98, pages 119–129, New York, NY, USA, 1998. ACM.

[6] Eduard Ayguadé, Nawal Copty, Alejandro Duran, Jay Hoeflinger, Yuan Lin, Federico Massaioli, Xavier Teruel, Priya Unnikrishnan, and Guansong Zhang. The design of openmp tasks. *Parallel and Distributed Systems, IEEE Transactions on*, 20(3):404–418, 2009.

[7] Eduard Ayguade and et al. A proposal for task parallelism in openmp. In Barbara Chapman and et al., editors, *A Practical Programming Model for the Multi-Core Era*, volume 4935 of *Lecture Notes in Computer Science*, pages 1–12. Springer Berlin Heidelberg, 2008.

[8] Nick Barrow-Williams, Christian Fensch, and Simon Moore. A communication characterisation of splash-2 and parsec. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pages 86–97. IEEE, 2009.

[9] Major Bhadauria, Vincent M Weaver, and Sally A McKee. Understanding parsec performance on contemporary cmps. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pages 98–107. IEEE, 2009.

[10] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The parsec benchmark suite: characterization and architectural implications. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 72–81. ACM, 2008.

[11] Arif Bilgin. Graphviz - graph visualization software, 1988.

[12] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. In *Proceedings 35th Annual Symposium on Foundations of Computer Science*, pages 356–368, Nov 1994.

[13] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '95, pages 207–216, New York, NY, USA, 1995. ACM.

[14] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, September 1999.

[15] F. Warren Burton and M. Ronan Sleep. Executing functional programs on a virtual tree of processors. In *Proceedings of the 1981 Conference on Functional Programming Languages and Computer Architecture*, FPCA '81, pages 187–194, New York, NY, USA, 1981. ACM.

[16] Juan M Cebrian, Magnus Jahre, and Lasse Natvig. Parvec: vectorizing the parsec benchmark suite. *Computing*, 97(11):1077–1100, 2015.

[17] Dimitrios Chasapis, Marc Casas, Miquel Moretó, Raul Vidal, Eduard Ayguadé, Jesús Labarta, and Mateo Valero. Parsecss: Evaluating the impact of task parallelism in the parsec benchmark suite. *ACM Transactions on Architecture and Code Optimization (TACO)*, 12(4):41, 2016.

[18] David Chase and Yossi Lev. Dynamic circular work-stealing deque. In *Proceedings of the Seventeenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '05, pages 21–28, New York, NY, USA, 2005. ACM.

[19] Leonardo Dagum and Ramesh Menon. Openmp: an industry standard api for shared-memory programming. *IEEE computational science and engineering*, 5(1):46–55, 1998.

[20] Andrew Danowitz, Kyle Kelley, James Mao, John P. Stevenson, and Mark Horowitz. Cpu db: Recording microprocessor history. *Commun. ACM*, 55(4):55–63, April 2012.

[21] Andi Drebes, Antoniu Pop, Karine Heydemann, Albert Cohen, and Nathalie Drach-Temam. Aftermath: A graphical tool for performance analysis and debugging of fine-grained task-parallel programs and run-time systems. In *Proceedings of 7th Workshop on Programmability Issues for Heterogeneous Multicores*, MULTIPROG '14, 2014.

[22] Alejandro Duran, Xavier Teruel, Roger Ferrer, Xavier Martorell, and Eduard Ayguade. Barcelona OpenMP Tasks Suite: A Set of Benchmarks Targeting the Exploitation of Task Parallelism in OpenMP. In *2009 International Conference on Parallel Processing*, pages 124–131. IEEE, September 2009.

[23] Alejandro Duran González, Xavier Teruel, Roger Ferrer, Xavier Martorell Bofill, and Eduard Ayguadé Parra. Barcelona openmp tasks suite: A set of benchmarks targeting the exploitation of task parallelism in openmp. In *38th International Conference on Parallel Processing*, pages 124–131, 2009.

[24] Matteo Frigo, Charles E. Leiserson, and Keith 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 '98, pages 212–223, New York, NY, USA, 1998. ACM.

[25] Felix Garcia and Javier Fernandez. Posix thread libraries. *Linux J.*, 2000(70es), February 2000.

[26] Robert H. Halstead, Jr. Implementation of multilisp: Lisp on a multiprocessor. In *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*, LFP '84, pages 9–17, New York, NY, USA, 1984. ACM.

[27] Robert H. Halstead, Jr. Multilisp: A language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst.*, 7(4):501–538, October 1985.

[28] Yuxiong He, Charles E. Leiserson, and William M. Leiserson. The cilkview scalability analyzer. In *Proceedings of the Twenty-second Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '10, pages 145–156, New York, NY, USA, 2010. ACM.

[29] S. Hunold, R. Hoffmann, and F. Suter. Jedule: A tool for visualizing schedules of parallel applications. In *Parallel Processing Workshops (ICPPW), 2010 39th International Conference on*, pages 169–178, Sept 2010.

[30] A. Huynh and K. Taura. Delay spotter: A tool for spotting scheduler-caused delays in task parallel runtime systems. In *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 114–125, Sept 2017.

[31] An Huynh, Douglas Thain, Miquel Pericàs, and Kenjiro Taura. DAGViz: A dag visualization tool for analyzing task-parallel program traces. In *Proceedings of the 2nd Workshop on Visual Performance Analysis*, VPA '15, pages 3:1–3:8. ACM, 2015.

[32] Intel. Intel cilk plus homepage.

[33] Intel. Intel vtune amplifier, 2015. [Online; last accessed July 5, 2015].

[34] Intel. Intel(r) threading building blocks reference manual, 2015.

[35] Christian Iwainsky, Thomas Reichstein, Christopher Dahnken, Dieteran Mey, Christian Terboven, Andrey Semin, and Christian Bischof. An approach to visualize remote socket traffic on the intel nehalem-ex. In *Euro-Par 2010 Parallel Processing Workshops*, volume 6586 of *Lecture Notes in Computer Science*, pages 523–530. Springer Berlin Heidelberg, 2011.

[36] Jacques Chassin De Kergommeaux, Benhur De Oliveira Stein, and Montbonnot Saint Martin. Paje: An extensible environment for visualizing multi-threaded program executions. In *Proc. Euro-Par 2000, Springer-Verlag, LNCS*, pages 133–144, 1900.

[37] D. A. Kranz, R. H. Halstead, Jr., and E. Mohr. Mul-t: A high-performance parallel lisp. In *Proceedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation*, PLDI '89, pages 81–90, New York, NY, USA, 1989. ACM.

[38] A.G. Landge, J.A. Levine, A. Bhatele, K.E. Isaacs, T. Gamblin, M. Schulz, S.H. Langer, P.-T. Bremer, and V. Pascucci. Visualizing network traffic to understand the performance of massively parallel simulations. *Visualization and Computer Graphics, IEEE Transactions on*, 18(12):2467–2476, Dec 2012.

[39] Doug Lea. A java fork/join framework. In *Proceedings of the ACM 2000 Conference on Java Grande*, JAVA '00, pages 36–43, New York, NY, USA, 2000. ACM.

[40] I-Ting Angelina Lee, Charles E. Leiserson, Tao B. Schardl, Zhunping Zhang, and Jim Sukha. On-the-fly pipeline parallelism. *ACM Trans. Parallel Comput.*, 2(3):17:1–17:42, September 2015.

[41] Daan Leijen, Wolfram Schulte, and Sebastian 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 '09, pages 227–242, New York, NY, USA, 2009. ACM.

[42] Charles E. Leiserson. The Cilk++ concurrency platform. In *Proceedings of the 46th Annual Design Automation Conference DAC '09*. ACM Press, July 2009.

[43] Xu Liu and John 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 '14, pages 259–272. ACM, 2014.

[44] Germán Llort, Harald Servat, Juan González, Judit Giménez, and Jesús Labarta. On the usefulness of object tracking techniques in performance analysis. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 29:1–29:11, New York, NY, USA, 2013. ACM.

[45] Zoltan Majo and Thomas R Gross. (mis) understanding the numa memory system performance of multithreaded workloads. In *Workload Characterization (IISWC), 2013 IEEE International Symposium on*, pages 11–22. IEEE, 2013.

[46] Zoltan Majo and Thomas R. Gross. A library for portable and composable data locality optimizations for numa systems. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP 2015, pages 227–238. ACM, 2015.

[47] E. Mohr, D. A. Kranz, and R. H. Halstead. Lazy task creation: a technique for increasing the granularity of parallel programs. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):264–280, Jul 1991.

[48] Eric Mohr, David A. Kranz, and Robert H. Halstead, Jr. Lazy task creation: A technique for increasing the granularity of parallel programs. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, LFP '90, pages 185–197, New York, NY, USA, 1990. ACM.

[49] Ananya Muddukrishna, Peter A. Jonsson, Artur Podobas, and Mats Brorsson. Grain graphs: Openmp performance analysis made easy. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '16, pages 28:1–28:13. ACM, 2016.

[50] W. E. Nagel, A. Arnold, M. Weber, H.-Ch. Hoppe, and K. Solchenbach. Vampir: Visualization and analysis of mpi resources. *Supercomputer*, 12:69–80, 1996.

[51] Jun Nakashima, Sho Nakatani, and Kenjiro Taura. Design and implementation of a customizable work stealing scheduler. In *Proceedings of the 3rd International Workshop on Runtime and Operating Systems for Supercomputers - ROSS '13*. ACM Press, June 2013.

[52] Jun Nakashima and Kenjiro Taura. Massivethreads: A thread library for high productivity languages. In Gul Agha, Atsushi Igarashi, Naoki Kobayashi, Hidehiko Masuhara, Satoshi Matsuoka, Etsuya Shibayama, and Kenjiro Taura, editors, *Concurrent Objects and Beyond: Papers dedicated to Akinori Yonezawa on the Occasion of His 65th Birthday*, pages 222–238, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[53] Bradford Nichols, Dick Buttlar, and Jacqueline Farrell. *Pthreads programming: A POSIX standard for better multiprocessing*. " O'Reilly Media, Inc.", 1996.

[54] Stephen L. Olivier, Bronis R. de Supinski, Martin Schulz, and Jan F. Prins. Characterizing and mitigating work time inflation in task parallel programs. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 65:1–65:12. IEEE Computer Society Press, 2012.

[55] Stephen L Olivier, Allan K Porterfield, Kyle B Wheeler, Michael Spiegel, and Jan F Prins. Openmp task scheduling strategies for multicore numa systems. *Int. J. High Perform. Comput. Appl.*, 26(2):110–124, May 2012.

[56] OpenMP Architecture Review Board. OpenMP application program interface version 4.0, 2013.

[57] Chuck Pheatt. Intel(R) Threading Building Blocks. *J. Comput. Sci. Coll.*, 23(4):298–298, April 2008.

[58] Antoniu Pop and Albert Cohen. Openstream: Expressiveness and data-flow compilation of openmp streaming programs. *ACM Trans. Archit. Code Optim.*, 9(4):53:1–53:25, January 2013.

[59] Cairo Graphics Project. Cairo. `http://cairographics.org/`, 2016. [Online; last accessed June 7, 2016].

[60] The GTK+ Project. Gtk+ 3. `http://www.gtk.org/`, 2016. [Online; last accessed June 7, 2016].

[61] Arch D. Robison. Composable parallel patterns with intel cilk plus. *Computing in Science and Engineering*, 15(2):66–71, 2013.

[62] Tao B. Schardl, Bradley C. Kuszmaul, I-Ting Angelina Lee, William M. Leiserson, and Charles E. Leiserson. The cilkprof scalability profiler. In *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '15, pages 89–100, New York, NY, USA, 2015. ACM.

[63] Sameer S. Shende and Allen D. Malony. The tau parallel performance system. *Int. J. High Perform. Comput. Appl.*, 20(2):287–311, May 2006.

[64] B. Shneiderman. The eyes have it: a task by data type taxonomy for information visualizations. In *Visual Languages, 1996. Proceedings., IEEE Symposium on*, pages 336–343, Sep 1996.

[65] Gabriel Southern and Jose Renau. Analysis of parsec workload scalability. In *Performance Analysis of Systems and Software (ISPASS), 2016 IEEE International Symposium on*, pages 133–142. IEEE, 2016.

[66] Kozo Sugiyama, Shojiro Tagawa, and Mitsuhiko Toda. Methods for visual understanding of hierarchical system structures. *Systems, Man and Cybernetics, IEEE Transactions on*, 11(2):109–125, 1981.

[67] Warut Suksompong, Charles E. Leiserson, and Tao B. Schardl. On the efficiency of localized work stealing. *Information Processing Letters*, 116(2):100 – 106, 2016.

[68] Nathan R. Tallent and John 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 '09, pages 229–240. ACM, 2009.

[69] Kenjiro Taura and Jun Nakashima. A Comparative Study of Six Task Parallel Programming Systems (in Japanese). In *IPSJ SIG Technical Report HPC*, volume 140(16), pages 1–10. IPSJ, 2013.

[70] Vasanth Tovinkere and Michael Voss. Flow graph designer: A tool for designing and analyzing intel®threading building blocks flow graphs. In *Proceedings of the 2014 43rd International Conference on Parallel Processing Workshops*, ICPPW '14, pages 149–158, Washington, DC, USA, 2014. IEEE Computer Society.

[71] Kyle B. Wheeler, Richard C. Murphy, and Douglas Thain. Qthreads: An API for programming with millions of lightweight threads. In *2008 IEEE IPDPS*, pages 1–8. IEEE, April 2008.

[72] Kyle B. Wheeler and Douglas Thain. Visualizing massively multithreaded applications with ThreadScope. *Concurrency and Computation: Practice and Experience*, 22(1):45–67, January 2010.

[73] Jieting Wu, Jianping Zeng, Hongfeng Yu, and Joseph P. Kenny. Commgram: A new visual analytics tool for large communication trace data. In *Proceedings of the First Workshop on Visual Performance Analysis*, VPA '14, pages 28–35, Piscataway, NJ, USA, 2014. IEEE Press.

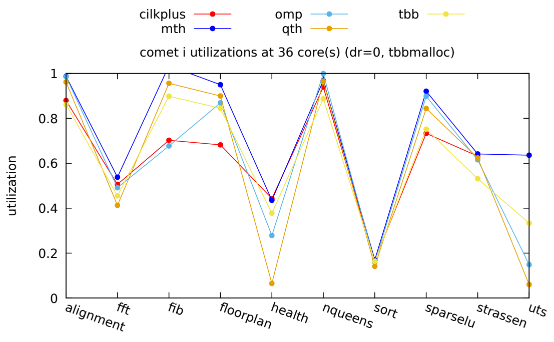
[74] Omer Zaki, Ewing Lusk, and Deborah Swider. Toward scalable performance visualization with jumpshot. *High Performance Computing Applications*, 13:277–288, 1999.

Appendices

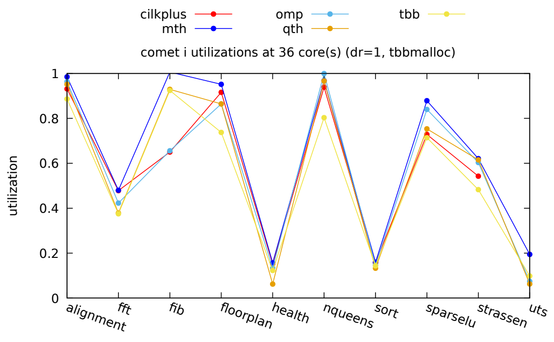
Appendix A

BOTS on Xeon E5-2699 v3

A.1 Overview



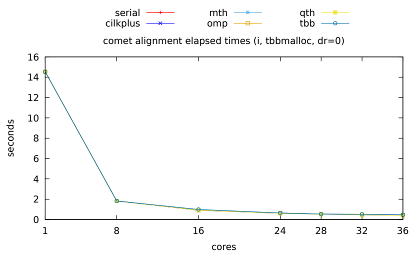
(a) with DR=0



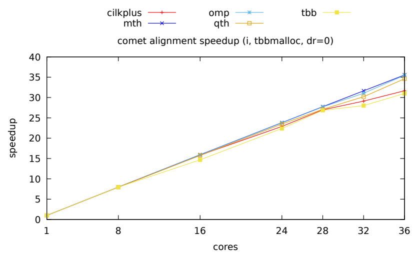
(b) with DR=1

Figure A.1: utilizations on 36 core(s)

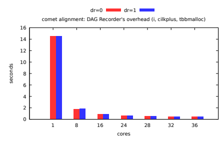
A.2 Alignment



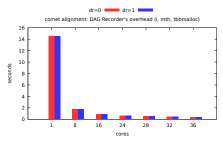
(a) elapsed times



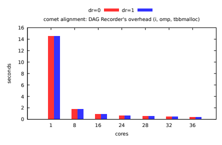
(b) speedups



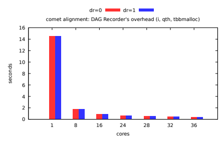
(c) cilkplus



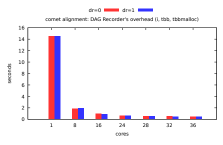
(d) mth



(e) omp

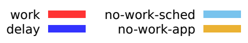


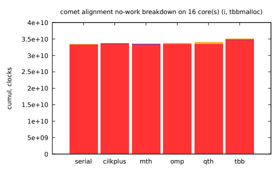
(f) qth



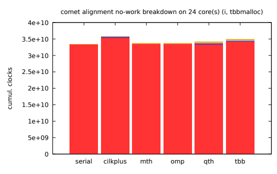
(g) tbb

Figure A.2: Elapsed times, speedups, and profiling overheads

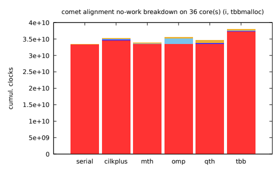




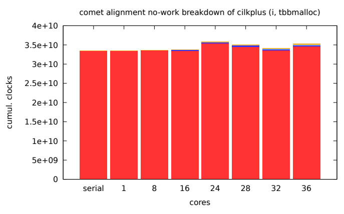
(a) on 16 cores



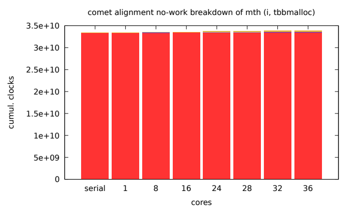
(b) on 24 cores



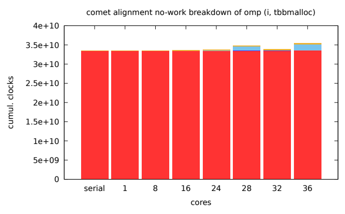
(c) on 36 cores



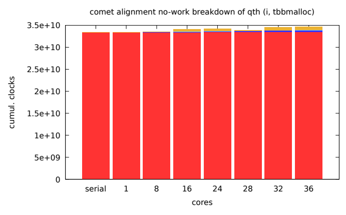
(d) cilkplus



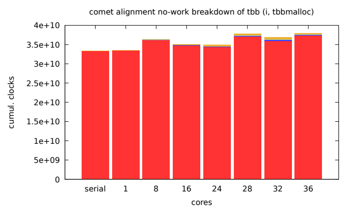
(e) mth



(f) omp



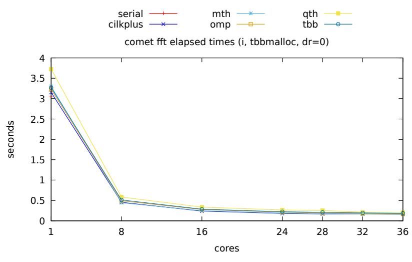
(g) qth

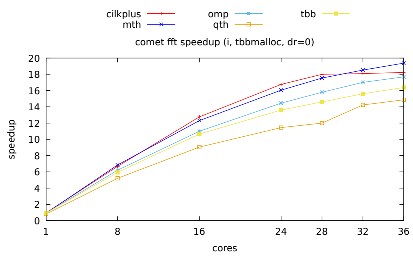


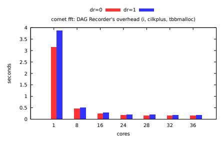
(h) tbb

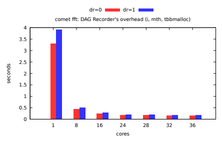
Figure A.3: Breakdown of cumulative execution times (elapsed time × cores)

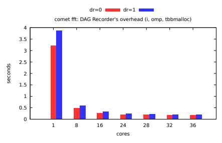
A.3 FFT

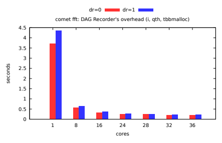

(a) elapsed times


(b) speedups


(c) cilkplus


(d) mth


(e) omp


(f) qth

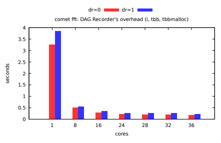
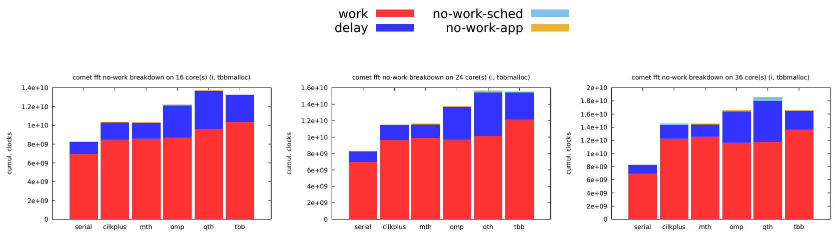
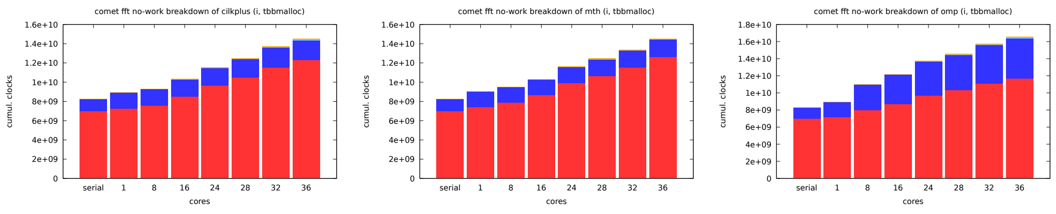

(g) tbb

Figure A.4: Elapsed times, speedups, and profiling overheads


(a) on 16 cores

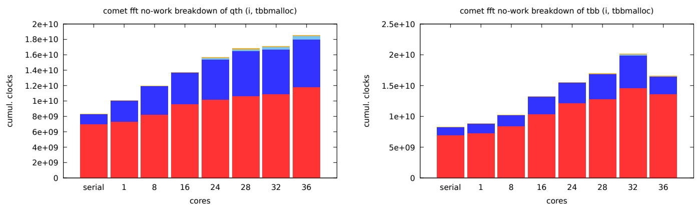
(b) on 24 cores

(c) on 36 cores


(d) cilkplus

(e) mth

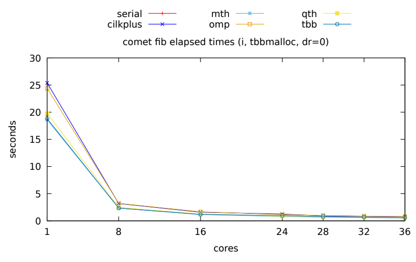
(f) omp


(g) qth

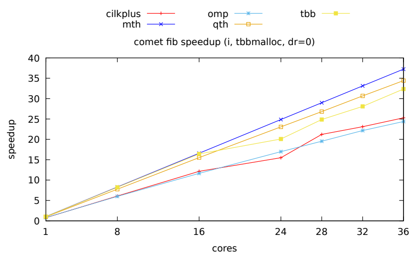
(h) tbb

Figure A.5: Breakdown of cumulative execution times (elapsed time × cores)

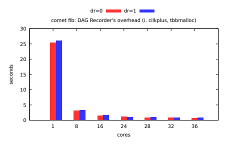
A.4 Fib



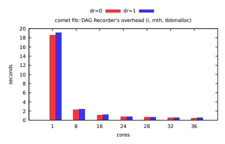
(a) elapsed times



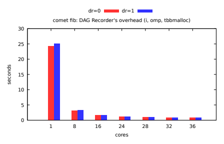
(b) speedups



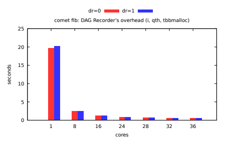
(c) cilkplus



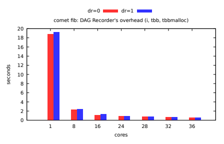
(d) mth



(e) omp



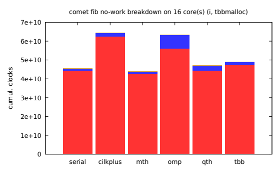
(f) qth



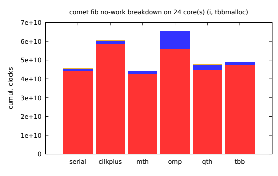
(g) tbb

Figure A.6: Elapsed times, speedups, and profiling overheads

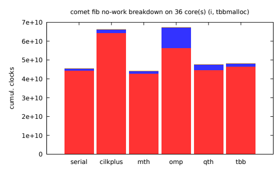
work 🟥 no-work-sched 🟦
delay 🟦 no-work-app 🟧



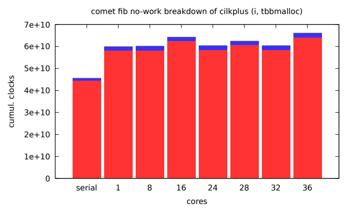
(a) on 16 cores



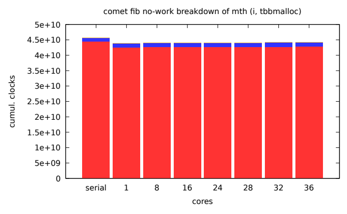
(b) on 24 cores



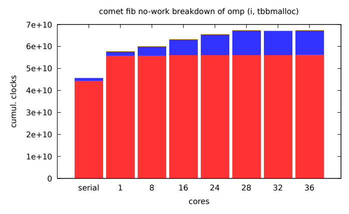
(c) on 36 cores



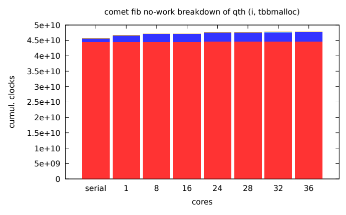
(d) cilkplus



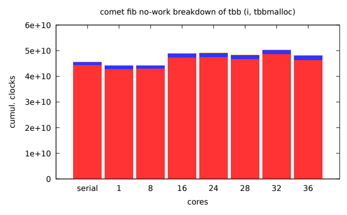
(e) mth



(f) omp



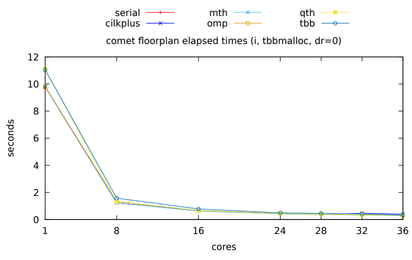
(g) qth



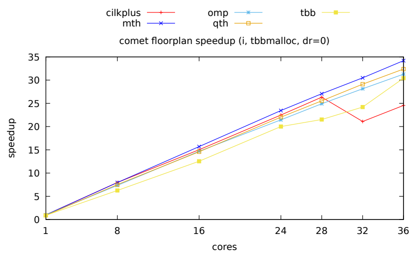
(h) tbb

Figure A.7: Breakdown of cumulative execution times (elapsed time × cores)

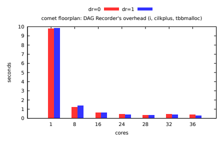
A.5 Floorplan

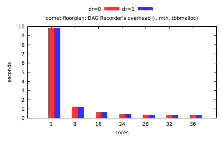


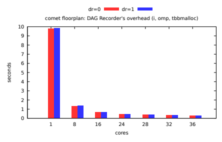
(a) elapsed times

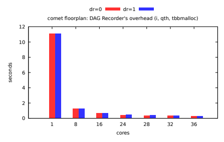


(b) speedups


(c) cilkplus


(d) mth


(e) omp


(f) qth

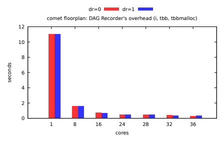
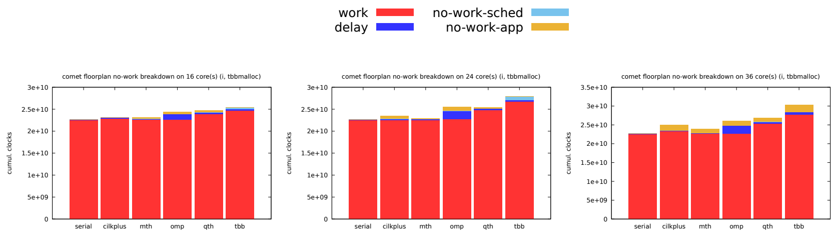
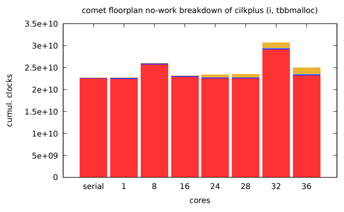

(g) tbb

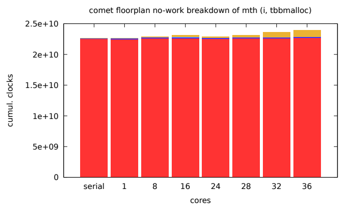
Figure A.8: Elapsed times, speedups, and profiling overheads



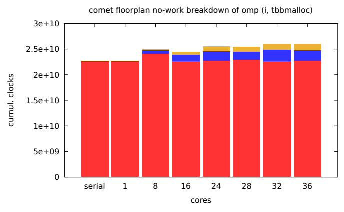
(a) on 16 cores
(b) on 24 cores
(c) on 36 cores



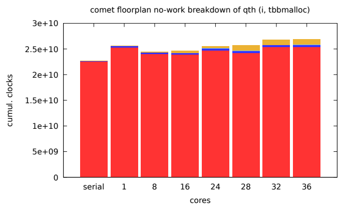
(d) cilkplus



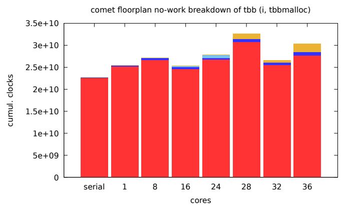
(e) mth



(f) omp



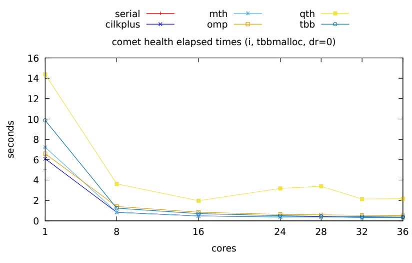
(g) qth



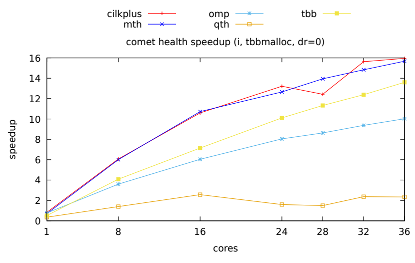
(h) tbb

Figure A.9: Breakdown of cumulative execution times (elapsed time × cores)

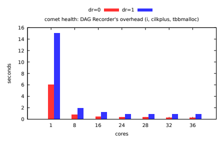
A.6 Health



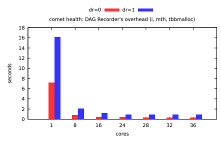
(a) elapsed times



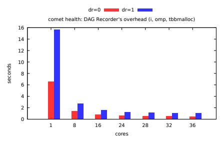
(b) speedups



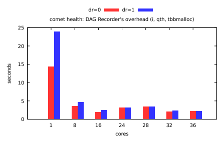
(c) cilkplus



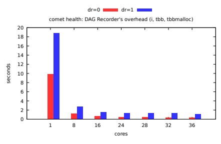
(d) mth



(e) omp

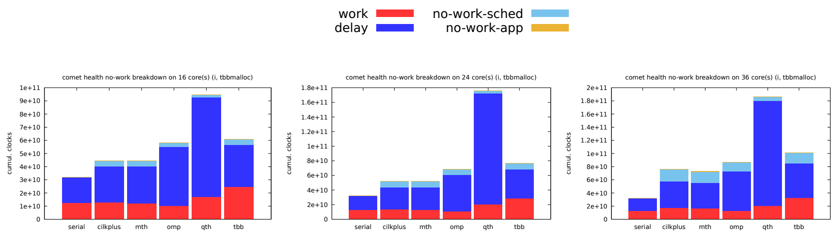


(f) qth

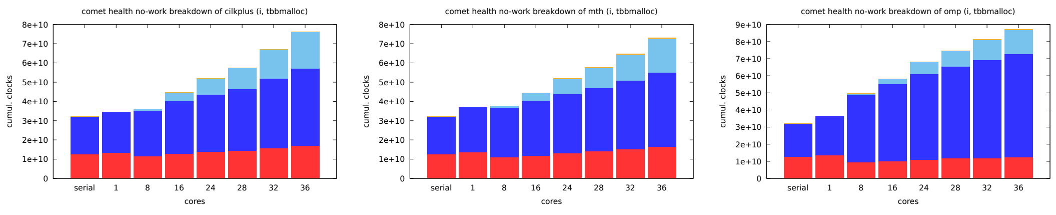


(g) tbb

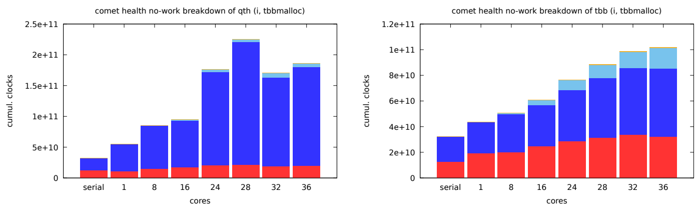
Figure A.10: Elapsed times, speedups, and profiling overheads



(a) on 16 cores (b) on 24 cores (c) on 36 cores



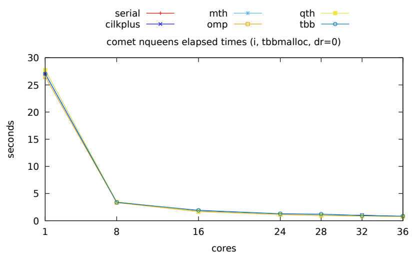
(d) cilkplus (e) mth (f) omp



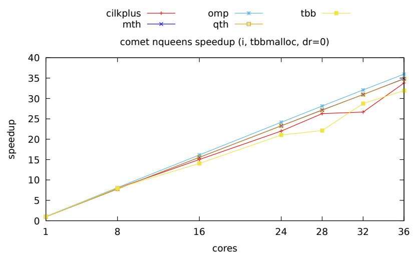
(g) qth (h) tbb

Figure A.11: Breakdown of cumulative execution times (elapsed time × cores)

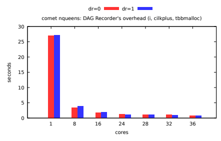
A.7 NQueens



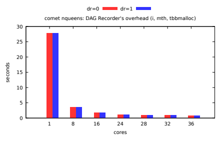
(a) elapsed times



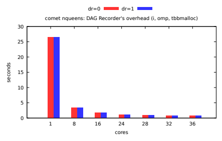
(b) speedups



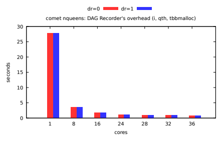
(c) cilkplus



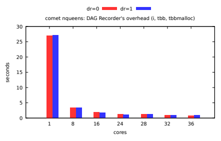
(d) mth



(e) omp

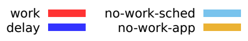


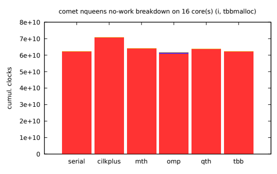
(f) qth



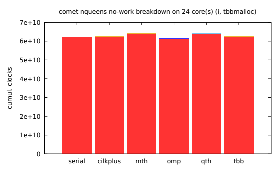
(g) tbb

Figure A.12: Elapsed times, speedups, and profiling overheads

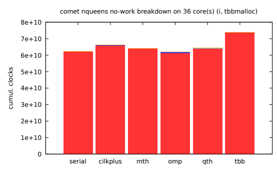




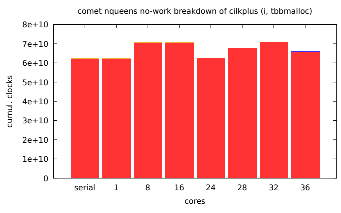
(a) on 16 cores



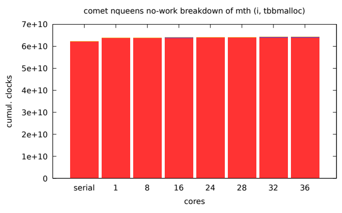
(b) on 24 cores



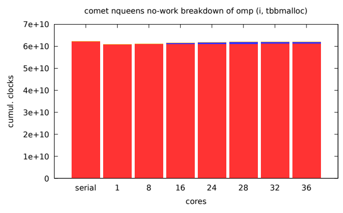
(c) on 36 cores



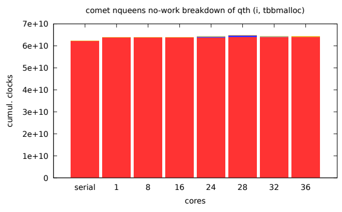
(d) cilkplus



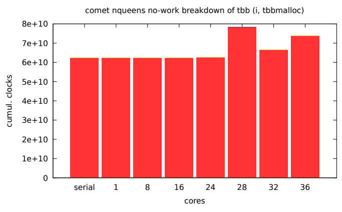
(e) mth



(f) omp



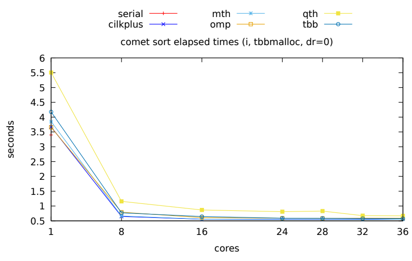
(g) qth



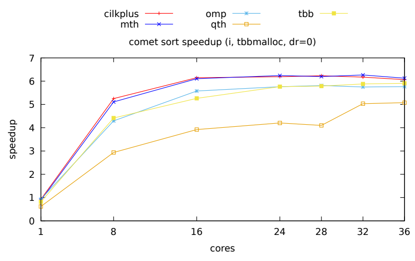
(h) tbb

Figure A.13: Breakdown of cumulative execution times (elapsed time × cores)

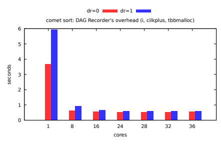
A.8 Sort



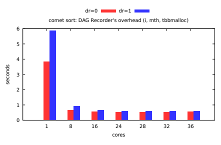
(a) elapsed times



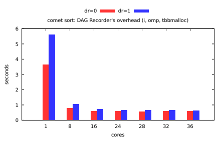
(b) speedups



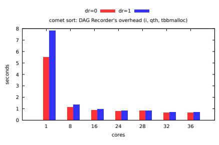
(c) cilkplus



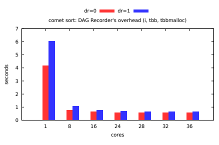
(d) mth



(e) omp

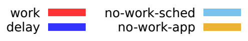


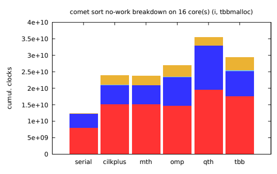
(f) qth



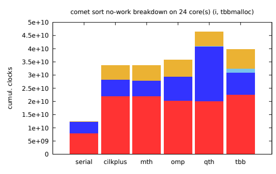
(g) tbb

Figure A.14: Elapsed times, speedups, and profiling overheads

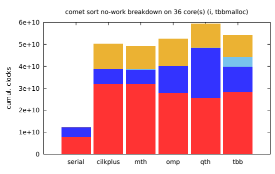




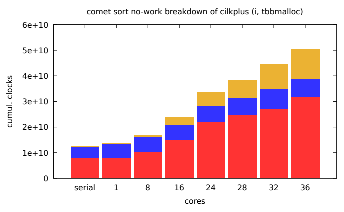
(a) on 16 cores



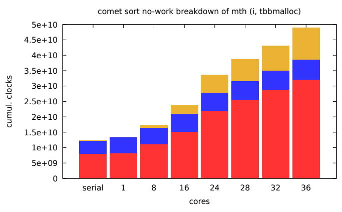
(b) on 24 cores



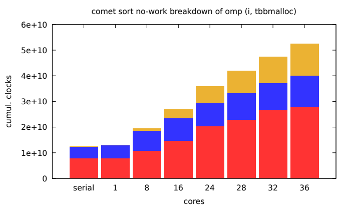
(c) on 36 cores



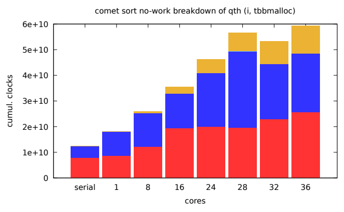
(d) cilkplus



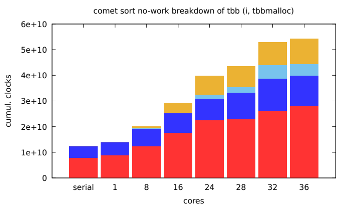
(e) mth



(f) omp



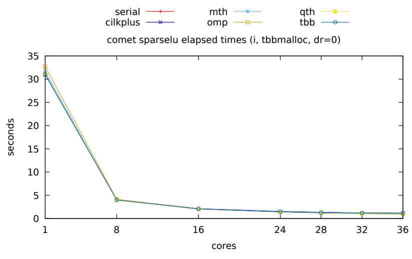
(g) qth



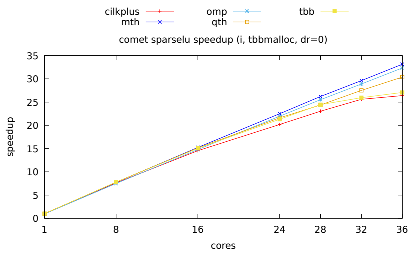
(h) tbb

Figure A.15: Breakdown of cumulative execution times (elapsed time × cores)

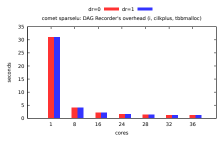
A.9 Sparselu



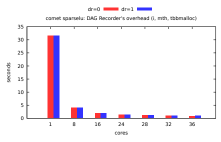
(a) elapsed times



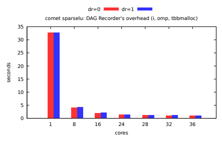
(b) speedups



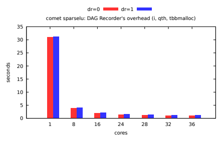
(c) cilkplus



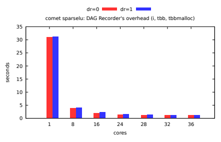
(d) mth



(e) omp

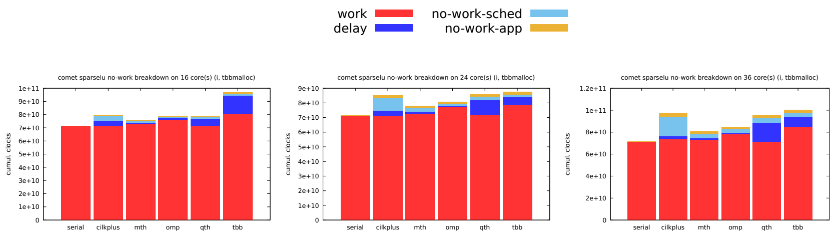


(f) qth

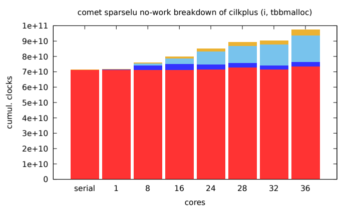


(g) tbb

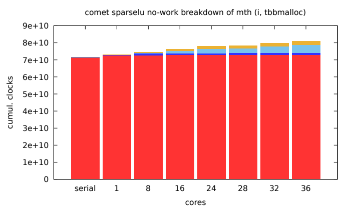
Figure A.16: Elapsed times, speedups, and profiling overheads



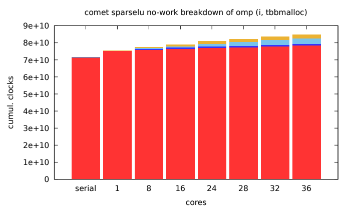
(a) on 16 cores (b) on 24 cores (c) on 36 cores



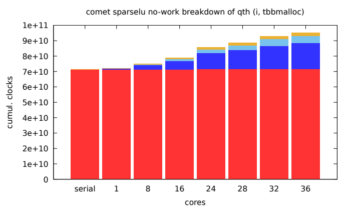
(d) cilkplus



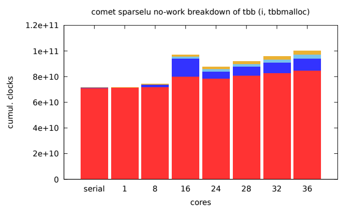
(e) mth



(f) omp



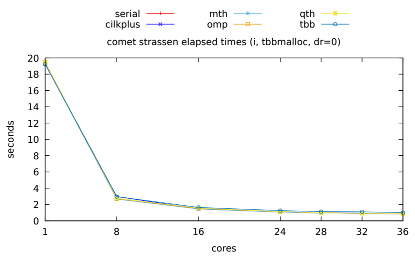
(g) qth



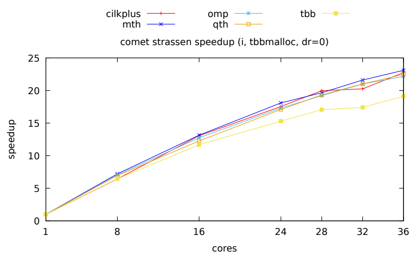
(h) tbb

Figure A.17: Breakdown of cumulative execution times (elapsed time × cores)

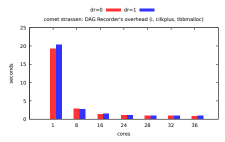
A.10 Strassen



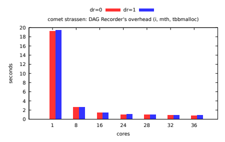
(a) elapsed times



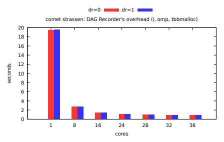
(b) speedups



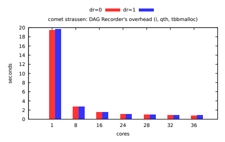
(c) cilkplus



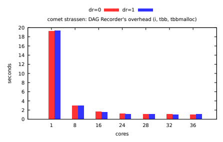
(d) mth



(e) omp

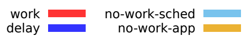


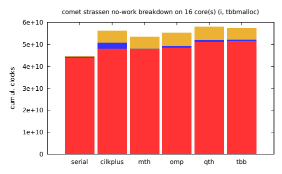
(f) qth



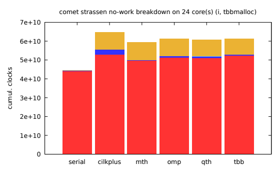
(g) tbb

Figure A.18: Elapsed times, speedups, and profiling overheads

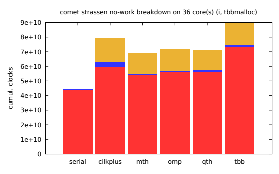




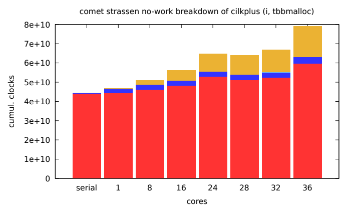
(a) on 16 cores



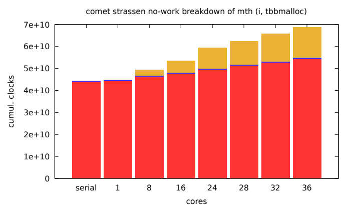
(b) on 24 cores



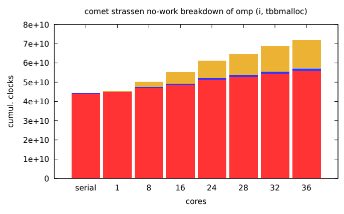
(c) on 36 cores



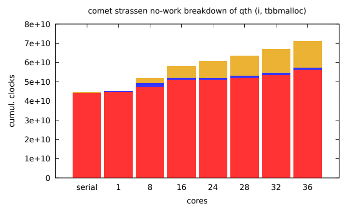
(d) cilkplus



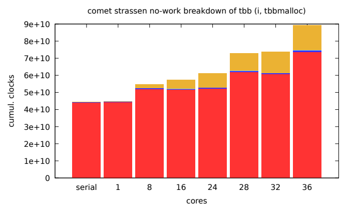
(e) mth



(f) omp



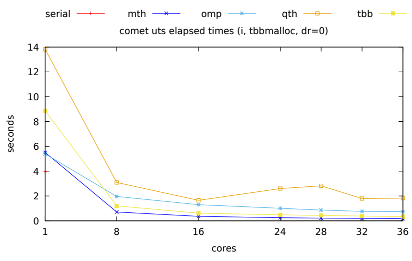
(g) qth



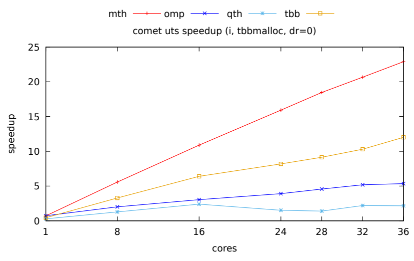
(h) tbb

Figure A.19: Breakdown of cumulative execution times (elapsed time × cores)

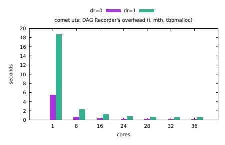
A.11 UTS



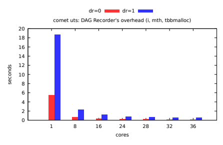
(a) elapsed times



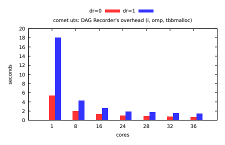
(b) speedups



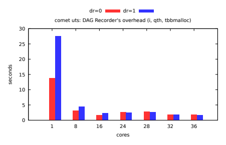
(c) cilkplus



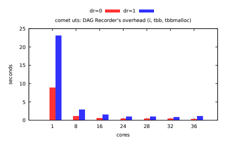
(d) mth



(e) omp

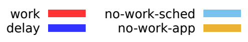


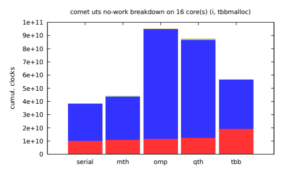
(f) qth



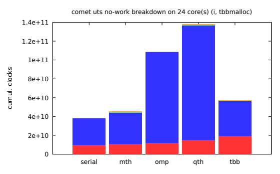
(g) tbb

Figure A.20: Elapsed times, speedups, and profiling overheads

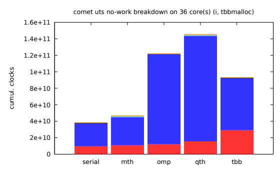




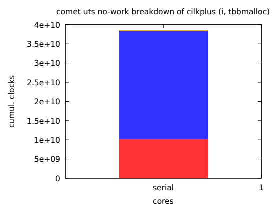
(a) on 16 cores



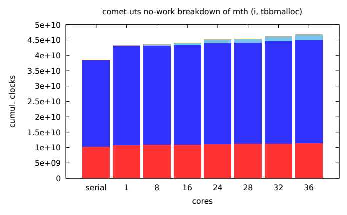
(b) on 24 cores



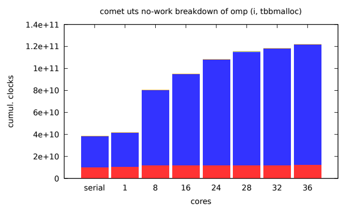
(c) on 36 cores



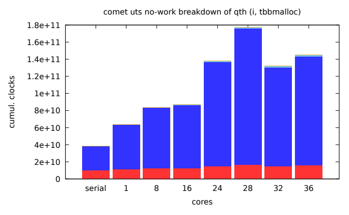
(d) cilkplus



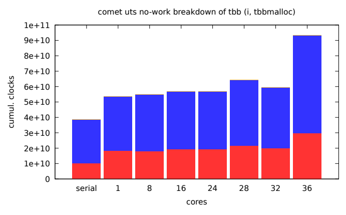
(e) mth



(f) omp



(g) qth



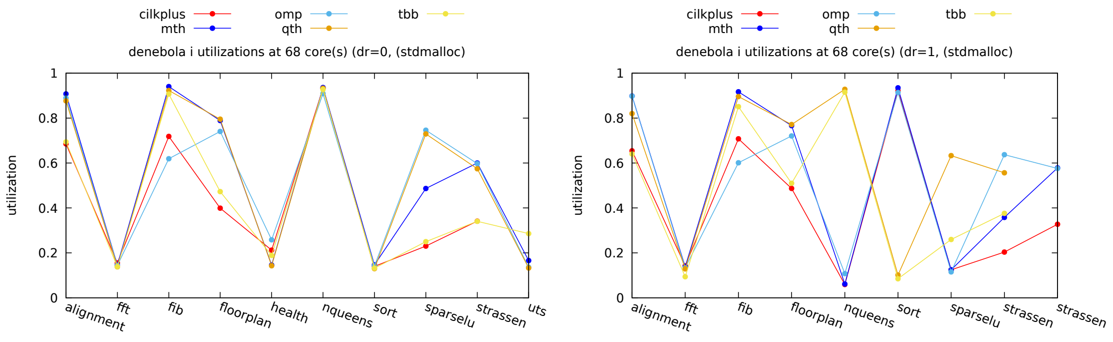
(h) tbb

Figure A.21: Breakdown of cumulative execution times (elapsed time × cores)

Appendix B

BOTS on Xeon Phi 7250 (Knights Landing)

B.1 Overview

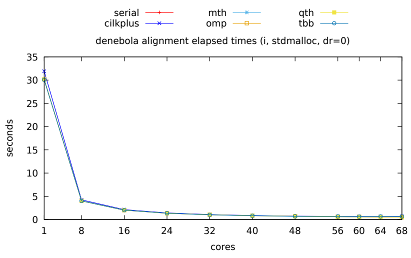


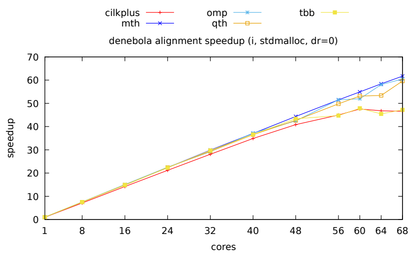
(a) with DR=0

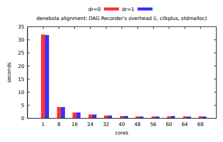
(b) with DR=1

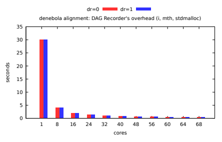
Figure B.1: utilizations on 68 core(s)

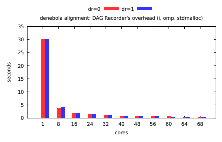
B.2 Alignment

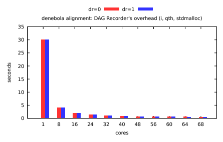

(a) elapsed times


(b) speedups


(c) cilkplus


(d) mth


(e) omp


(f) qth

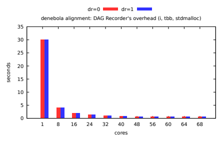
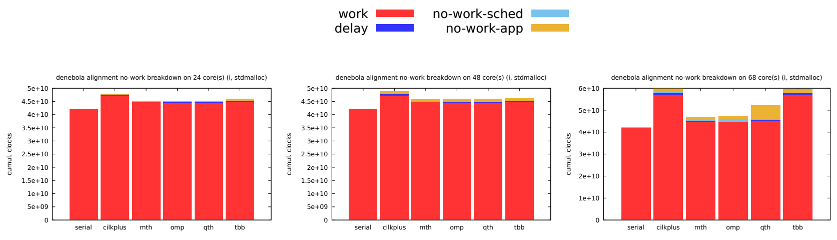
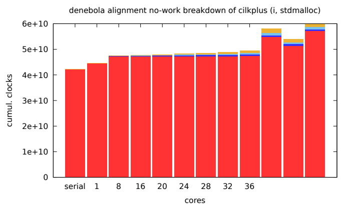
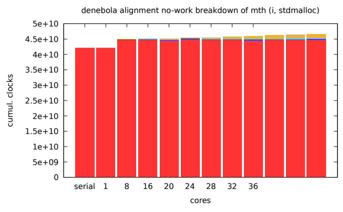
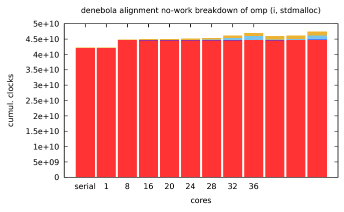

(g) tbb

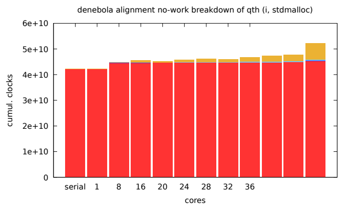
Figure B.2: Elapsed times, speedups, and profiling overheads


(a) on 24 cores (b) on 48 cores (c) on 68 cores


(d) cilkplus


(e) mth


(f) omp


(g) qth

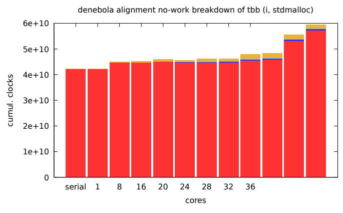
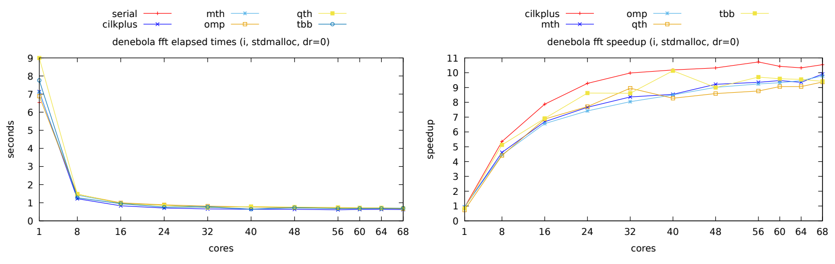

(h) tbb

Figure B.3: Breakdown of cumulative execution times (elapsed time × cores)

B.3 FFT


(a) elapsed times (b) speedups

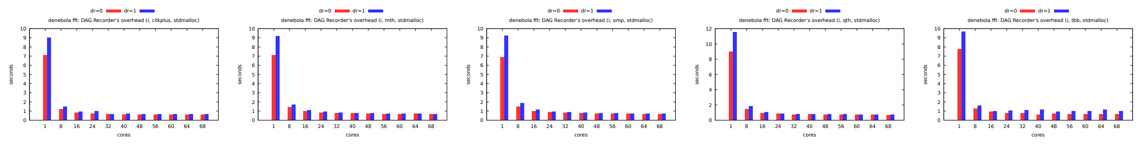
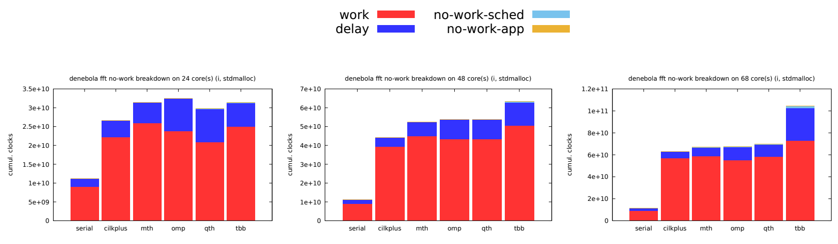
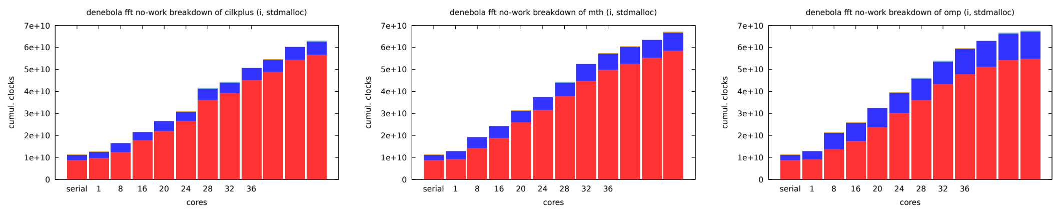

(c) cilkplus (d) mth (e) omp (f) qth (g) tbb

Figure B.4: Elapsed times, speedups, and profiling overheads


(a) on 24 cores (b) on 48 cores (c) on 68 cores


(d) cilkplus (e) mth (f) omp

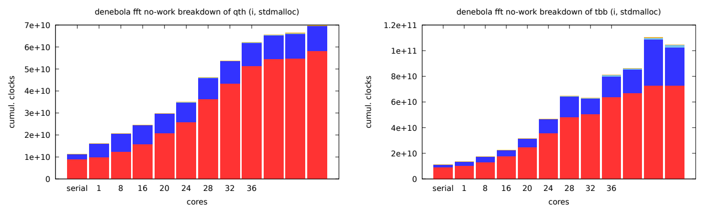
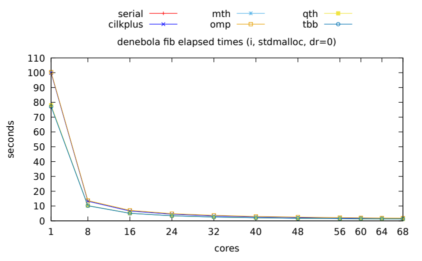
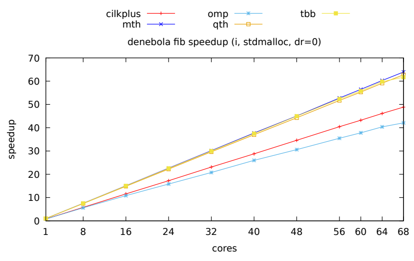

(g) qth (h) tbb

Figure B.5: Breakdown of cumulative execution times (elapsed time × cores)

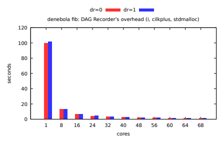
B.4 Fib

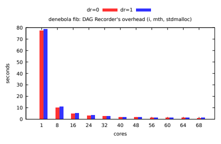


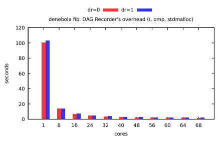
(a) elapsed times

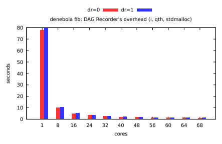


(b) speedups


(c) cilkplus


(d) mth


(e) omp


(f) qth

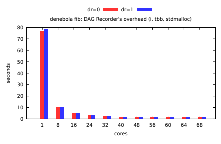
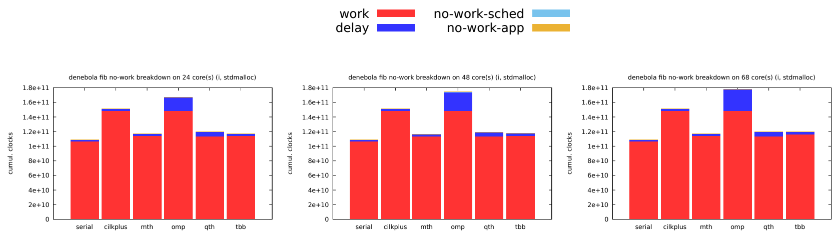
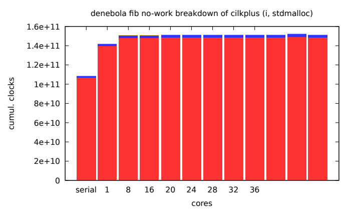

(g) tbb

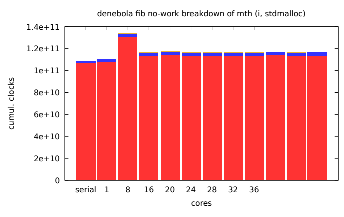
Figure B.6: Elapsed times, speedups, and profiling overheads



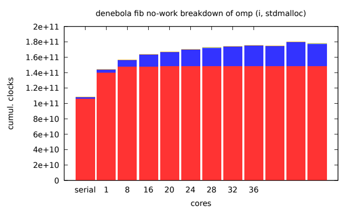
(a) on 24 cores (b) on 48 cores (c) on 68 cores



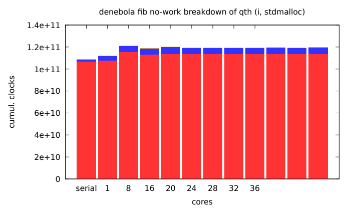
(d) cilkplus



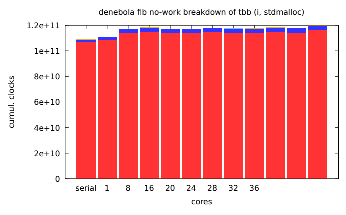
(e) mth



(f) omp



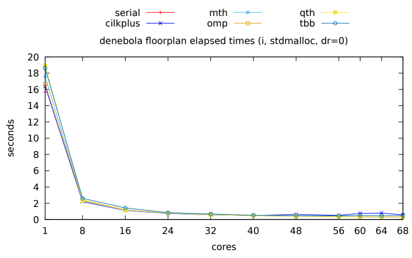
(g) qth



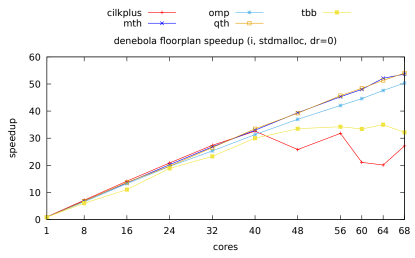
(h) tbb

Figure B.7: Breakdown of cumulative execution times (elapsed time × cores)

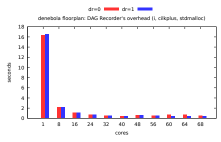
B.5 Floorplan

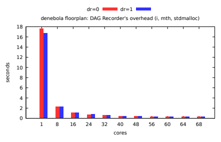


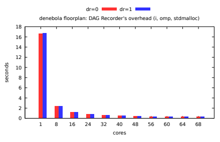
(a) elapsed times

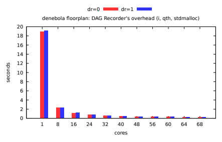


(b) speedups


(c) cilkplus


(d) mth


(e) omp


(f) qth

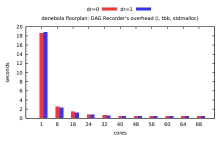
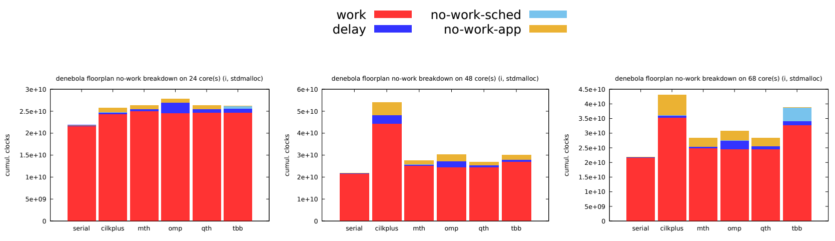
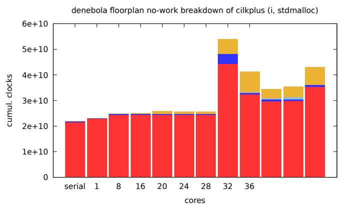
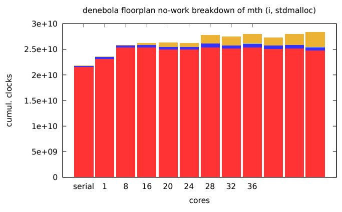

(g) tbb

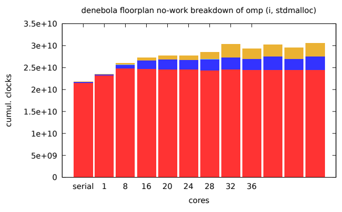
Figure B.8: Elapsed times, speedups, and profiling overheads

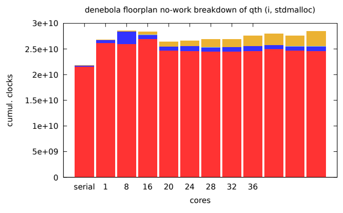


(a) on 24 cores (b) on 48 cores (c) on 68 cores


(d) cilkplus


(e) mth


(f) omp


(g) qth

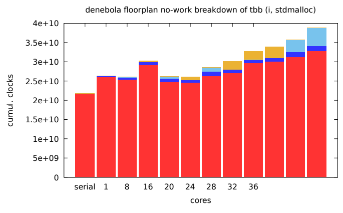
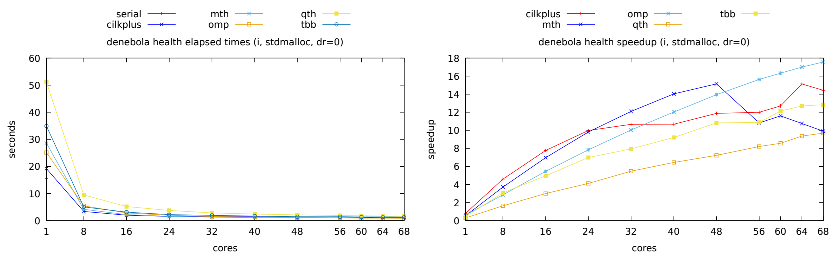
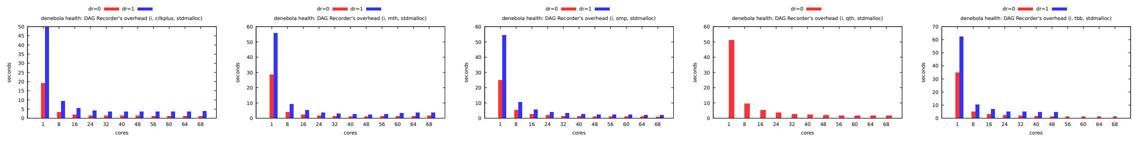

(h) tbb

Figure B.9: Breakdown of cumulative execution times (elapsed time × cores)

B.6 Health

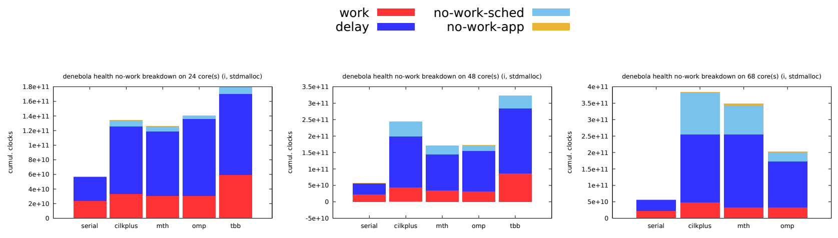


(a) elapsed times (b) speedups

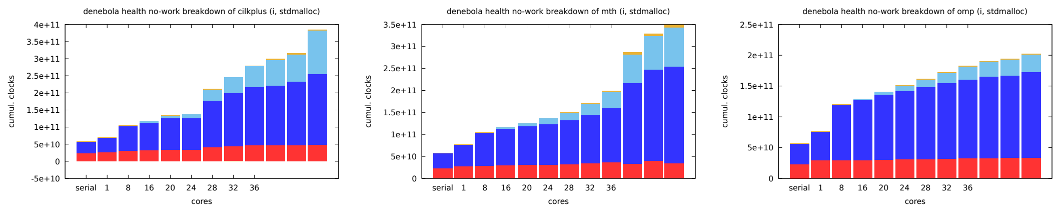


(c) cilkplus (d) mth (e) omp (f) qth (g) tbb

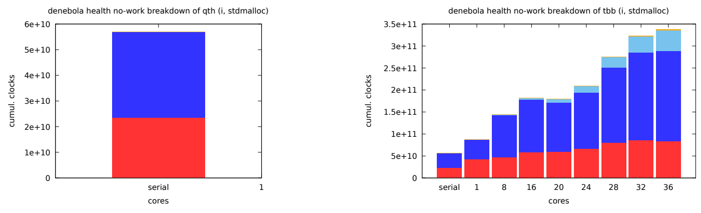
Figure B.10: Elapsed times, speedups, and profiling overheads



(a) on 24 cores (b) on 48 cores (c) on 68 cores



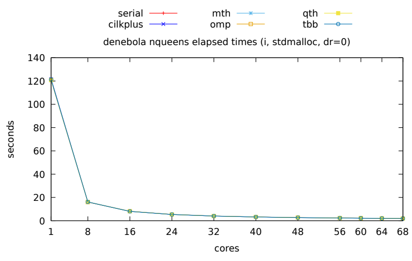
(d) cilkplus (e) mth (f) omp



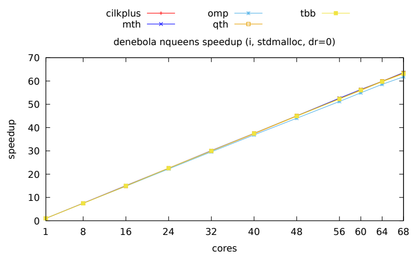
(g) qth (h) tbb

Figure B.11: Breakdown of cumulative execution times (elapsed time × cores)

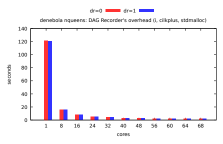
B.7 NQueens

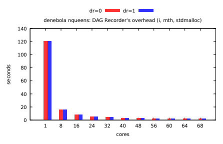


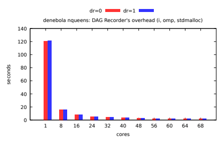
(a) elapsed times

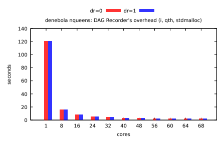


(b) speedups


(c) cilkplus


(d) mth


(e) omp


(f) qth

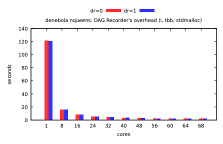
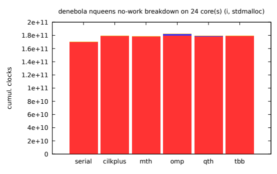
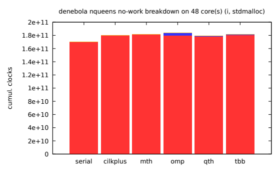

(g) tbb

Figure B.12: Elapsed times, speedups, and profiling overheads

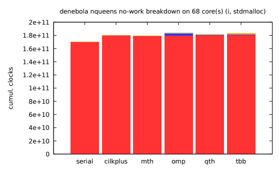
work 🟥 no-work-sched 🟦
delay 🟦 no-work-app 🟧



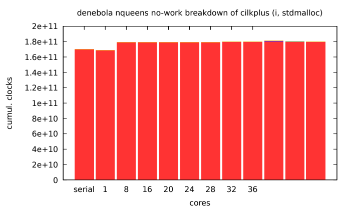
(a) on 24 cores



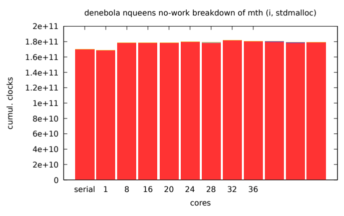
(b) on 48 cores



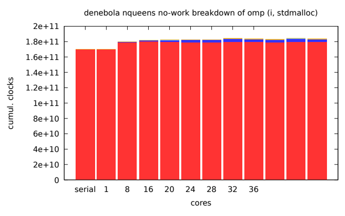
(c) on 68 cores



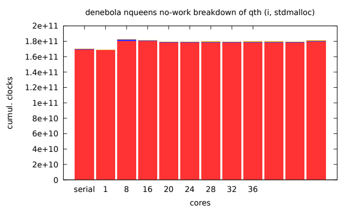
(d) cilkplus



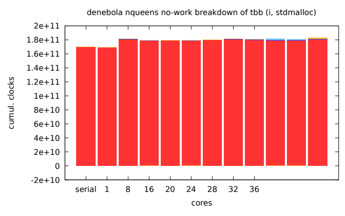
(e) mth



(f) omp



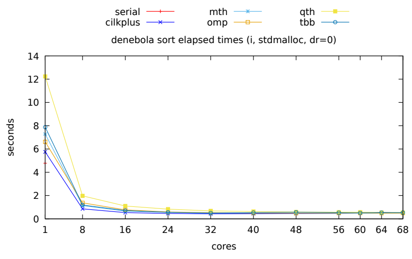
(g) qth



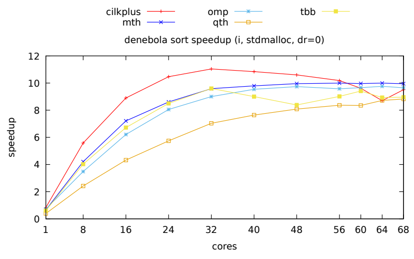
(h) tbb

Figure B.13: Breakdown of cumulative execution times (elapsed time × cores)

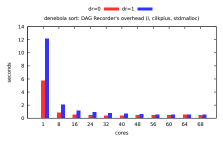
B.8 Sort



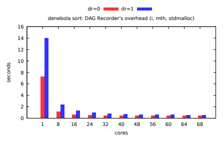
(a) elapsed times



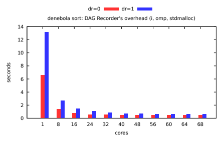
(b) speedups



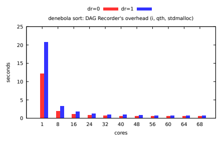
(c) cilkplus



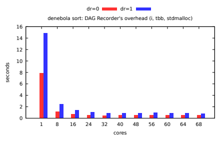
(d) mth



(e) omp

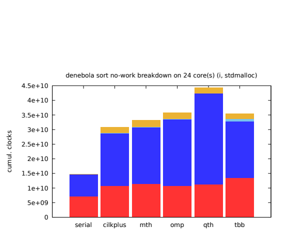


(f) qth

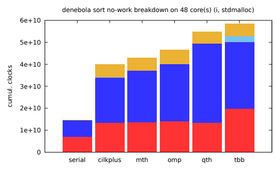


(g) tbb

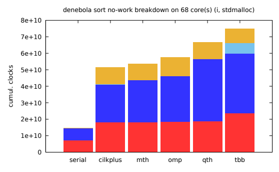
Figure B.14: Elapsed times, speedups, and profiling overheads



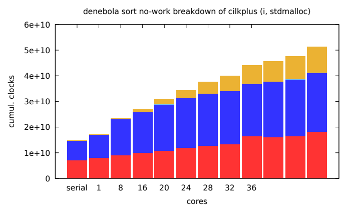
(a) on 24 cores



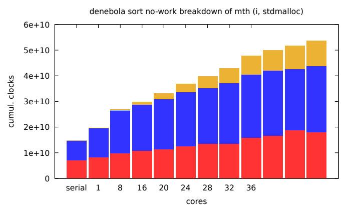
(b) on 48 cores



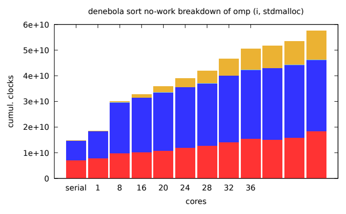
(c) on 68 cores



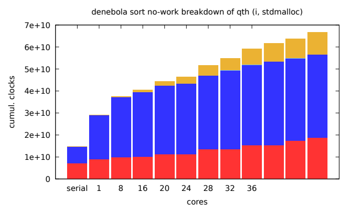
(d) cilkplus



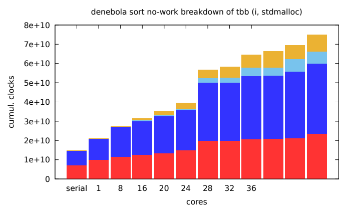
(e) mth



(f) omp



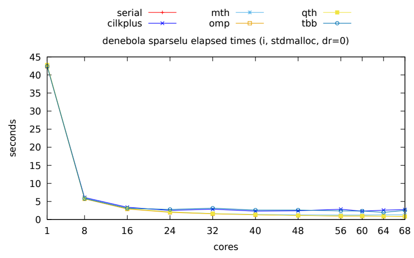
(g) qth



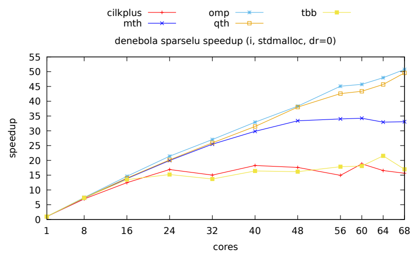
(h) tbb

Figure B.15: Breakdown of cumulative execution times (elapsed time × cores)

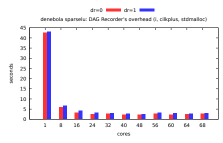
B.9 Sparselu



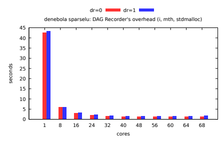
(a) elapsed times



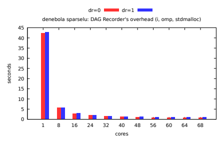
(b) speedups



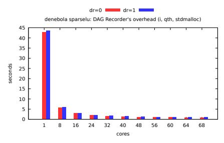
(c) cilkplus



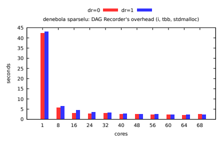
(d) mth



(e) omp



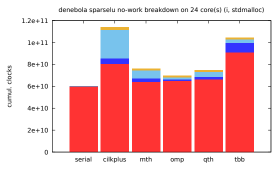
(f) qth



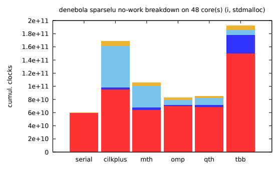
(g) tbb

Figure B.16: Elapsed times, speedups, and profiling overheads

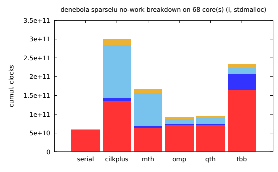
work 🟥 no-work-sched 🟦
delay 🟦 no-work-app 🟨



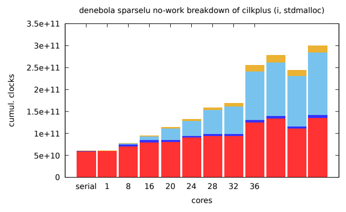
(a) on 24 cores



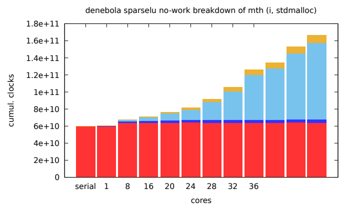
(b) on 48 cores



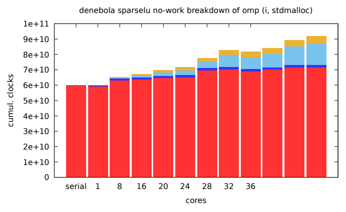
(c) on 68 cores



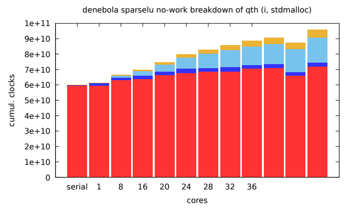
(d) cilkplus



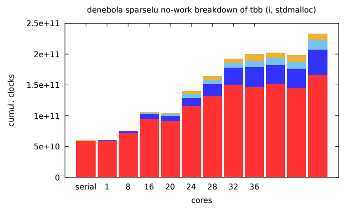
(e) mth



(f) omp



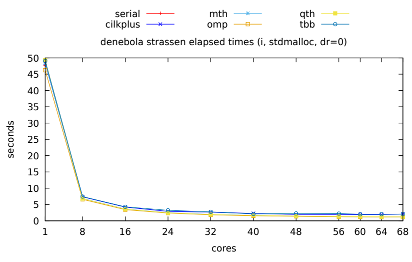
(g) qth

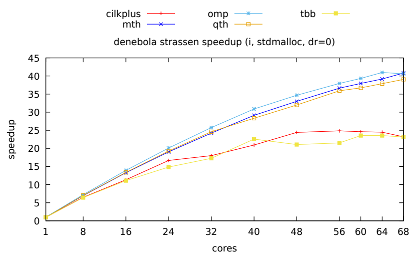


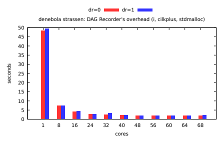
(h) tbb

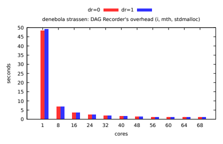
Figure B.17: Breakdown of cumulative execution times (elapsed time × cores)

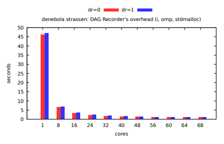
B.10 Strassen

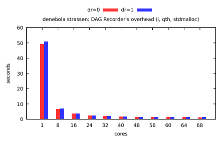

(a) elapsed times


(b) speedups


(c) cilkplus


(d) mth


(e) omp


(f) qth

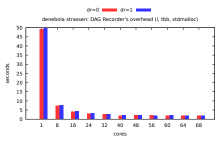
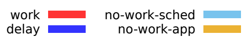
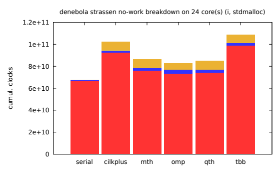
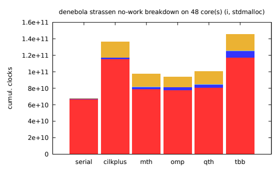
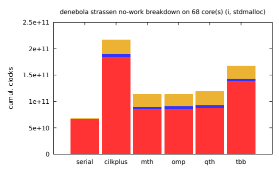

(g) tbb

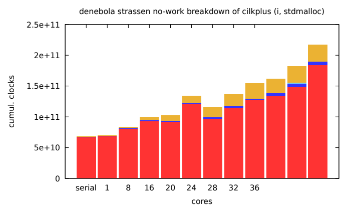
Figure B.18: Elapsed times, speedups, and profiling overheads

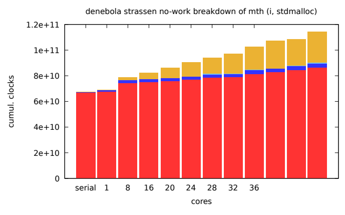


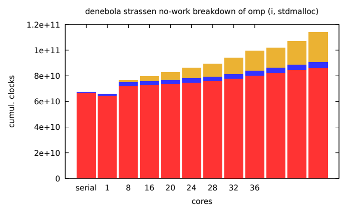

(a) on 24 cores

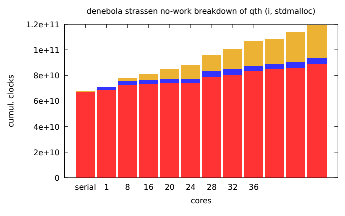

(b) on 48 cores


(c) on 68 cores


(d) cilkplus


(e) mth


(f) omp


(g) qth

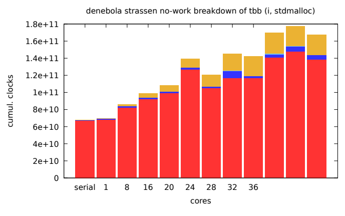
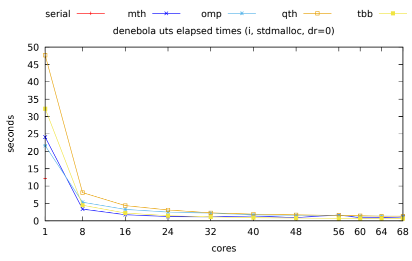
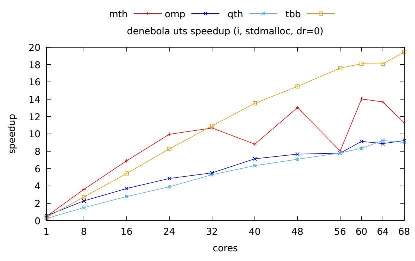

(h) tbb

Figure B.19: Breakdown of cumulative execution times (elapsed time × cores)

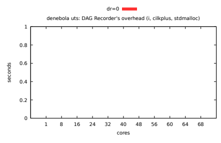
B.11 UTS

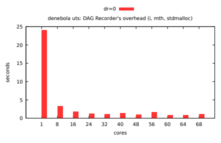


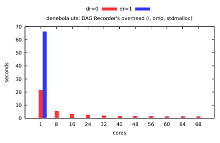
(a) elapsed times

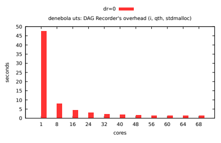


(b) speedups


(c) cilkplus


(d) mth


(e) omp


(f) qth

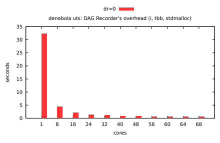
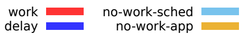
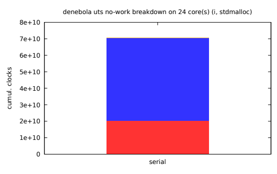
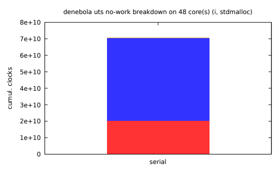

(g) tbb

Figure B.20: Elapsed times, speedups, and profiling overheads

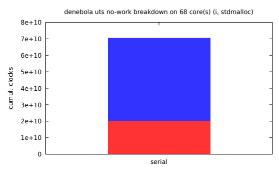




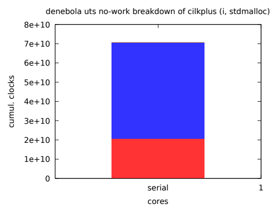
(a) on 24 cores



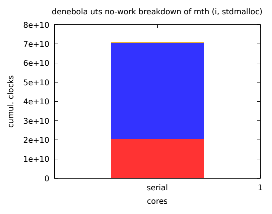
(b) on 48 cores



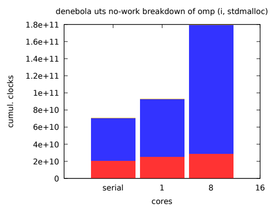
(c) on 68 cores



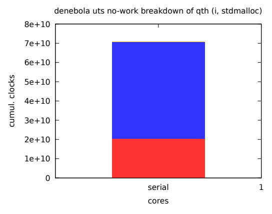
(d) cilkplus



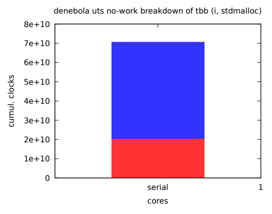
(e) mth



(f) omp



(g) qth



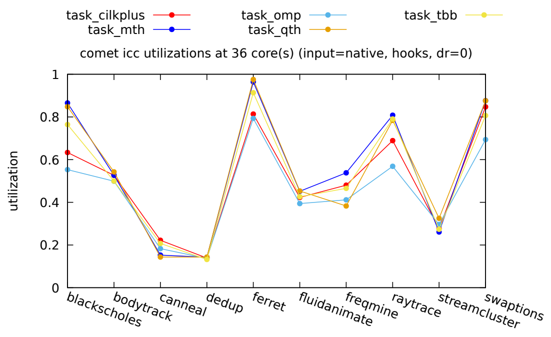
(h) tbb

Figure B.21: Breakdown of cumulative execution times (elapsed time × cores)

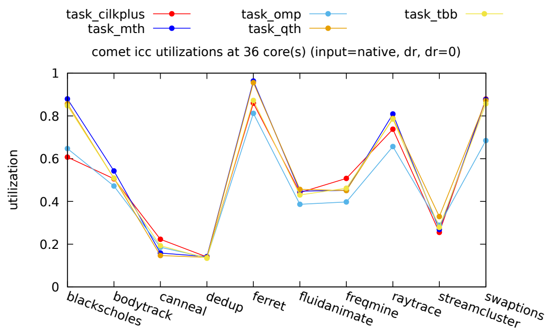
Appendix C

TP-PARSEC on Xeon E5-2699 v3

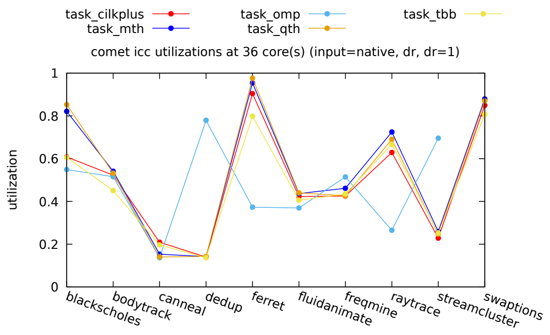
C.1 Overview



(a) compiled without DR



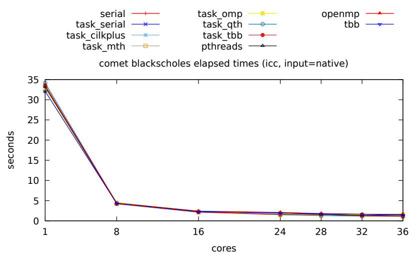
(b) DR off at runtime (DR=0)

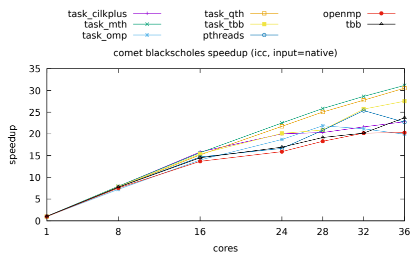


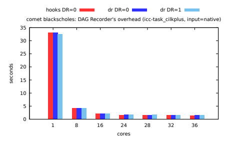
(c) DR on at runtime (DR=1)

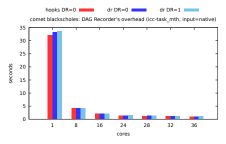
Figure C.1: utilizations on 36 core(s)

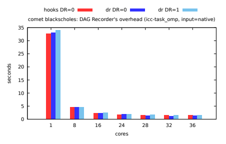
C.2 Blackscholes

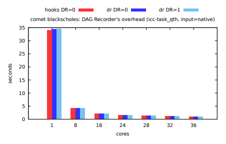

(a) elapsed times


(b) speedups


(c) cilkplus


(d) mth


(e) omp


(f) qth

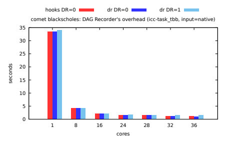
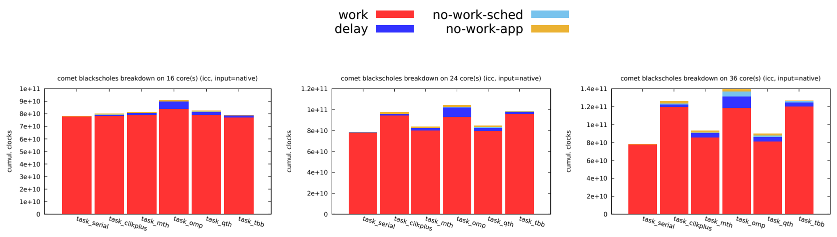
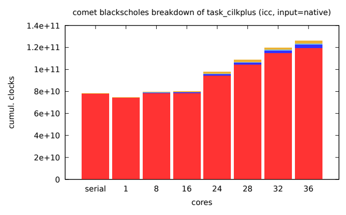

(g) tbb

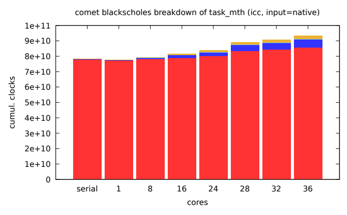
Figure C.2: Elapsed times, speedups, and profiling overheads

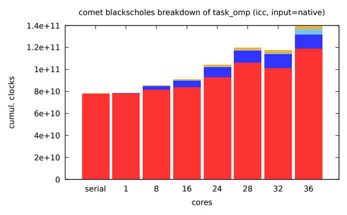

(a) on 16 cores

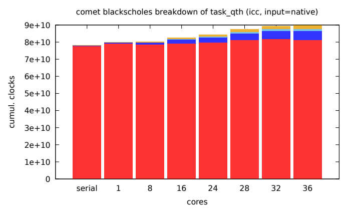
(b) on 24 cores

(c) on 36 cores


(d) cilkplus


(e) mth


(f) omp


(g) qth

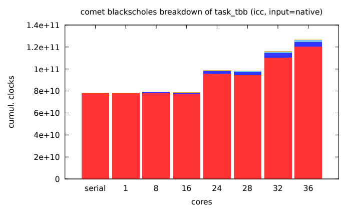
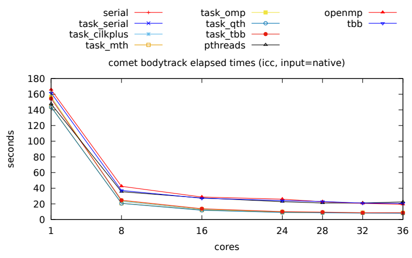
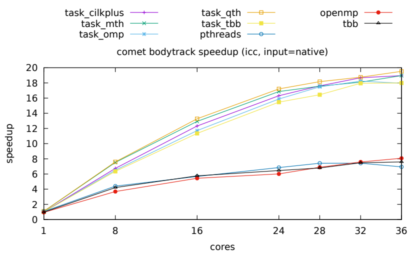

(h) tbb

Figure C.3: Breakdown of cumulative execution times (elapsed time × cores)

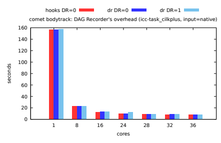
C.3 Bodytrack



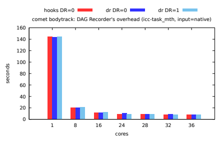
(a) elapsed times



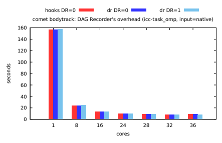
(b) speedups



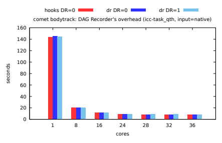
(c) cilkplus



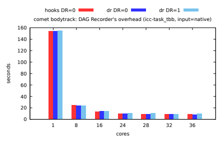
(d) mth



(e) omp

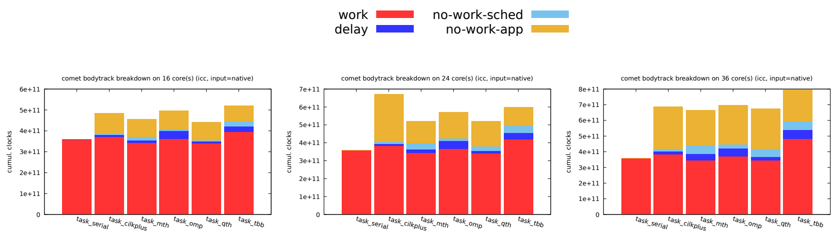


(f) qth

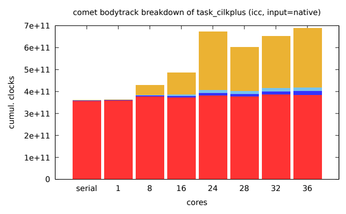


(g) tbb

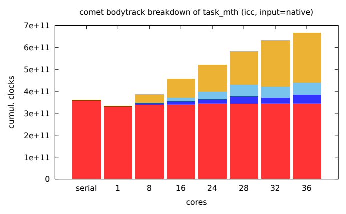
Figure C.4: Elapsed times, speedups, and profiling overheads



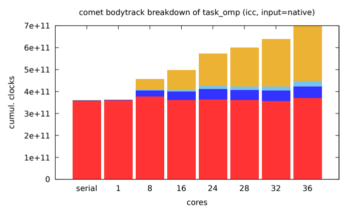
(a) on 16 cores (b) on 24 cores (c) on 36 cores



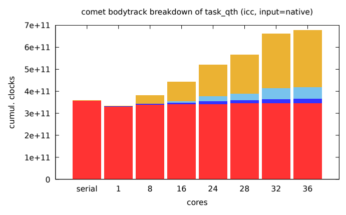
(d) cilkplus



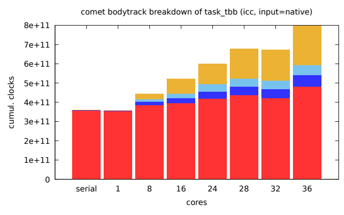
(e) mth



(f) omp



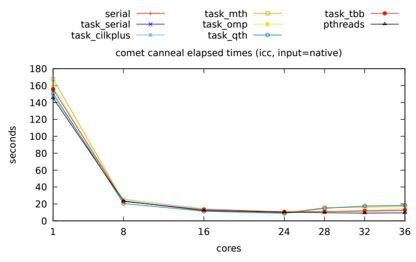
(g) qth



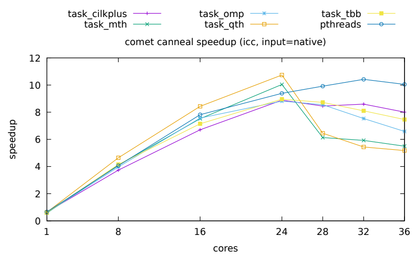
(h) tbb

Figure C.5: Breakdown of cumulative execution times (elapsed time × cores)

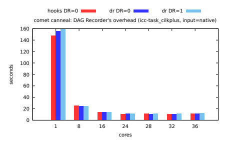
C.4 Canneal



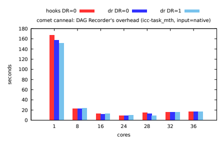
(a) elapsed times



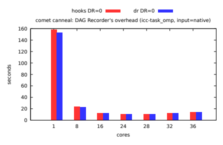
(b) speedups



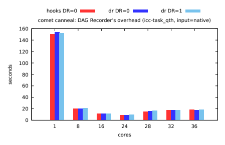
(c) cilkplus



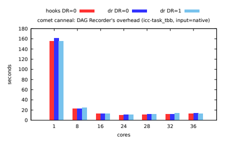
(d) mth



(e) omp

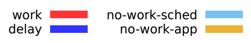


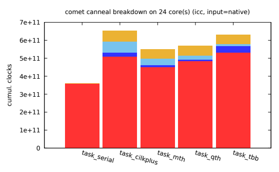
(f) qth



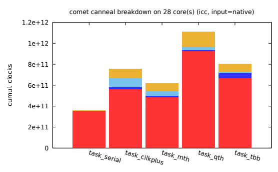
(g) tbb

Figure C.6: Elapsed times, speedups, and profiling overheads

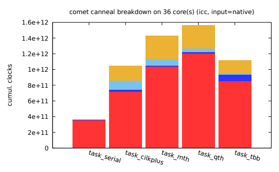




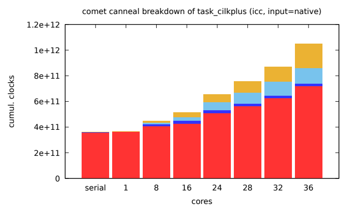
(a) on 24 cores



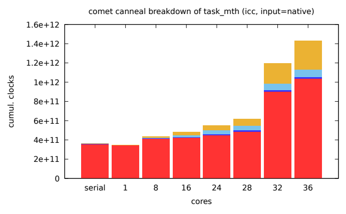
(b) on 28 cores



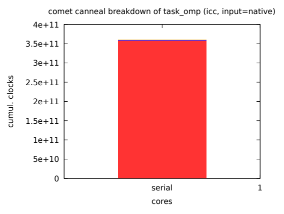
(c) on 36 cores



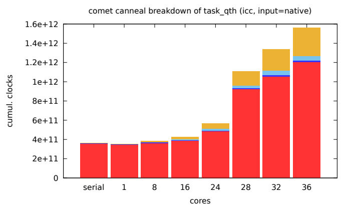
(d) cilkplus



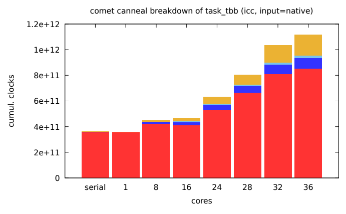
(e) mth



(f) omp



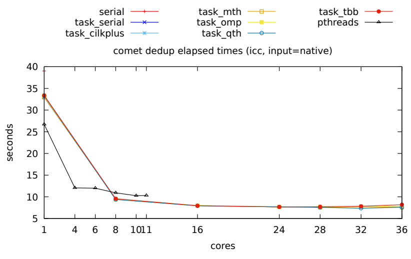
(g) qth



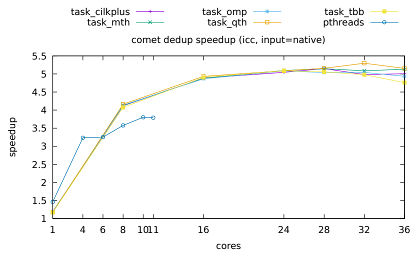
(h) tbb

Figure C.7: Breakdown of cumulative execution times (elapsed time × cores)

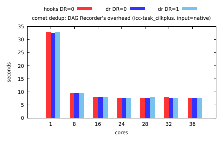
C.5 Dedup



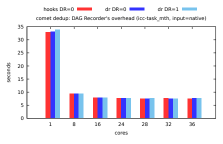
(a) elapsed times



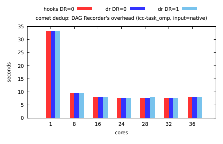
(b) speedups



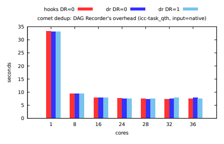
(c) cilkplus



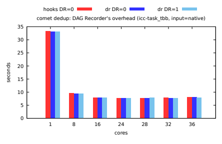
(d) mth



(e) omp

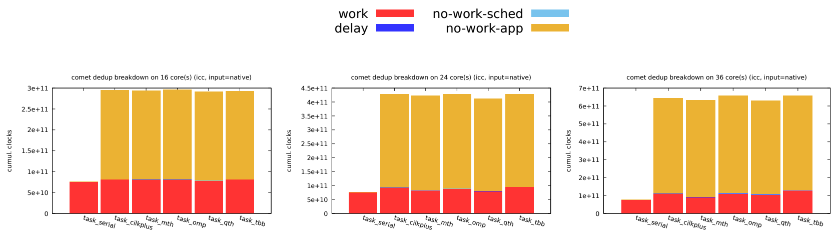


(f) qth

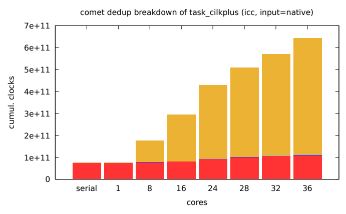


(g) tbb

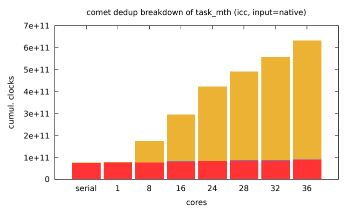
Figure C.8: Elapsed times, speedups, and profiling overheads



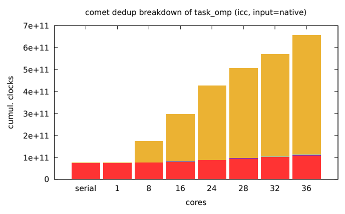
(a) on 16 cores (b) on 24 cores (c) on 36 cores



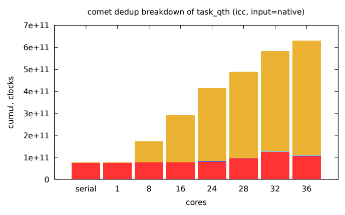
(d) cilkplus



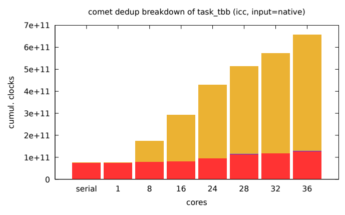
(e) mth



(f) omp



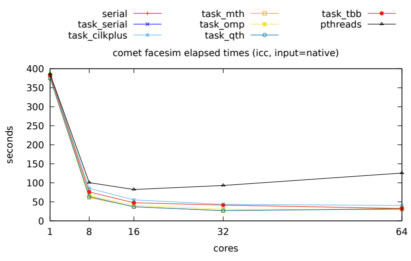
(g) qth

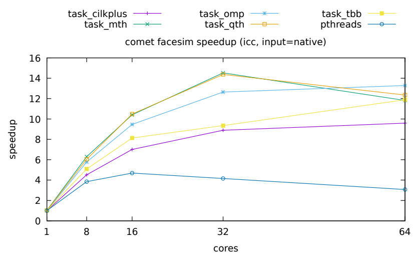


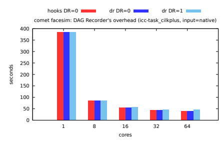
(h) tbb

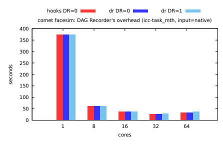
Figure C.9: Breakdown of cumulative execution times (elapsed time × cores)

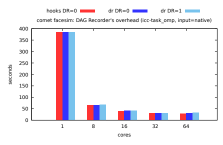
C.6 Facesim

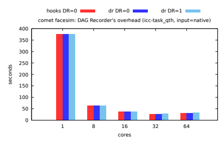

(a) elapsed times


(b) speedups


(c) cilkplus


(d) mth


(e) omp


(f) qth

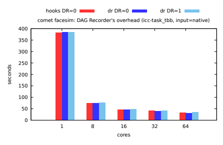
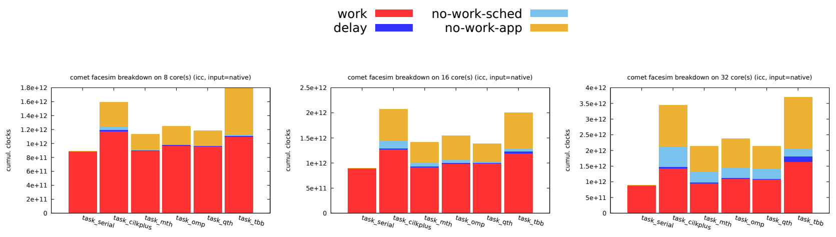
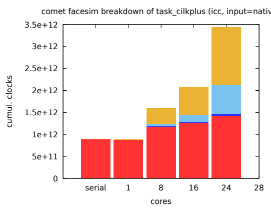
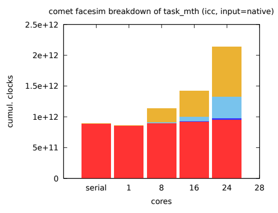
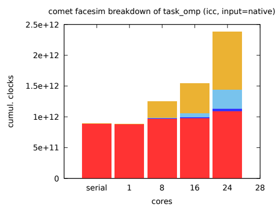

(g) tbb

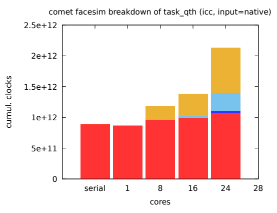
Figure C.10: Elapsed times, speedups, and profiling overheads


(a) on 8 cores (b) on 16 cores (c) on 32 cores


(d) cilkplus


(e) mth


(f) omp


(g) qth

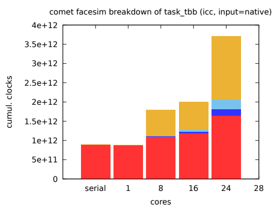
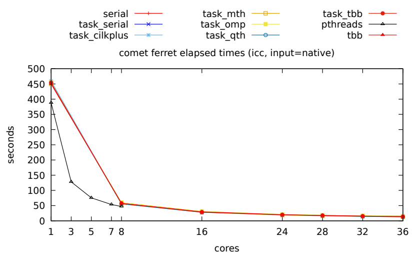
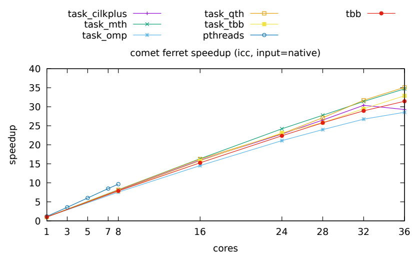

(h) tbb

Figure C.11: Breakdown of cumulative execution times (elapsed time × cores)

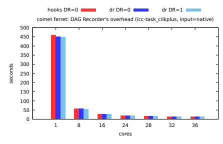
C.7 Ferret



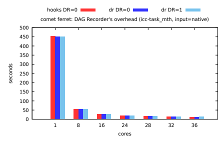
(a) elapsed times



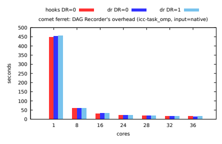
(b) speedups



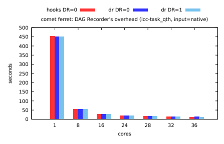
(c) cilkplus



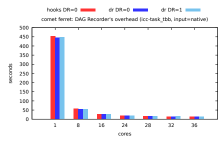
(d) mth



(e) omp

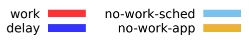


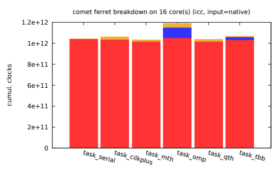
(f) qth



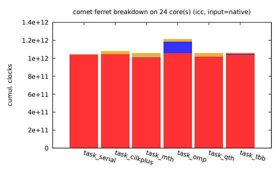
(g) tbb

Figure C.12: Elapsed times, speedups, and profiling overheads

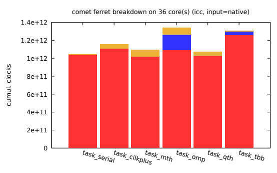




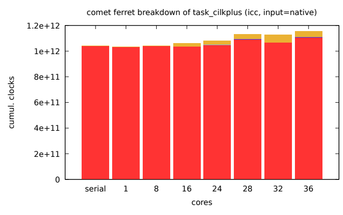
(a) on 16 cores



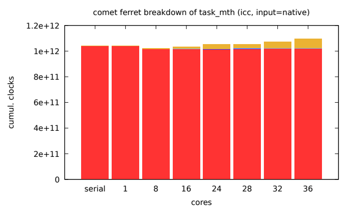
(b) on 24 cores



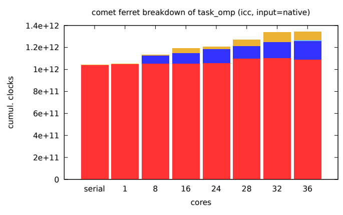
(c) on 36 cores



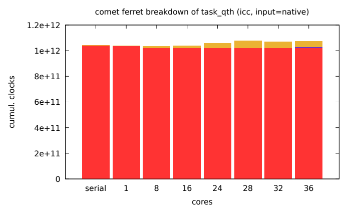
(d) cilkplus



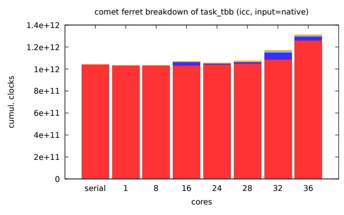
(e) mth



(f) omp



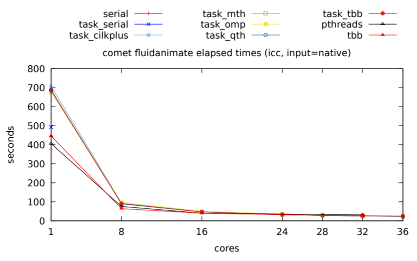
(g) qth



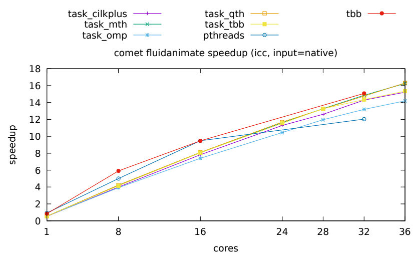
(h) tbb

Figure C.13: Breakdown of cumulative execution times (elapsed time × cores)

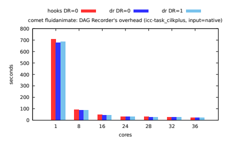
C.8 Fluidanimate



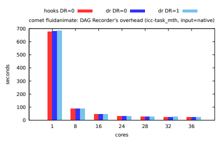
(a) elapsed times



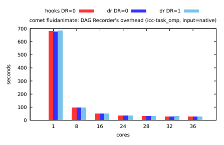
(b) speedups



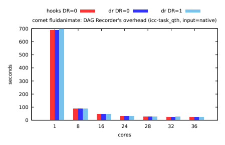
(c) cilkplus



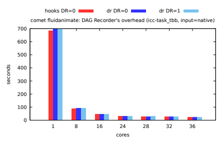
(d) mth



(e) omp

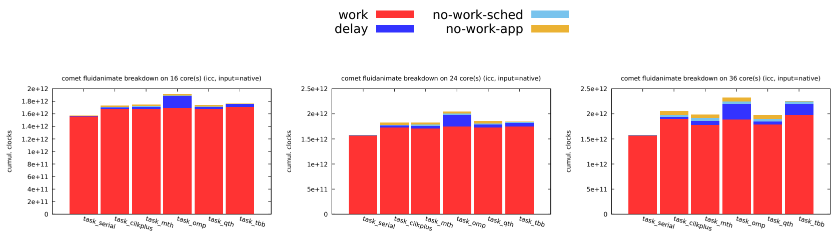


(f) qth

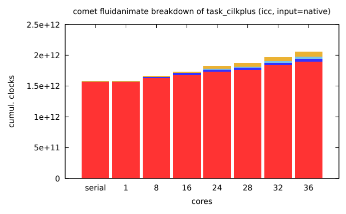


(g) tbb

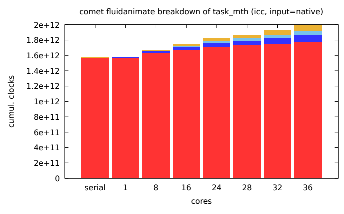
Figure C.14: Elapsed times, speedups, and profiling overheads



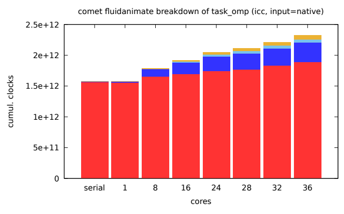
(a) on 16 cores (b) on 24 cores (c) on 36 cores



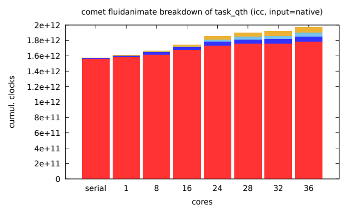
(d) cilkplus



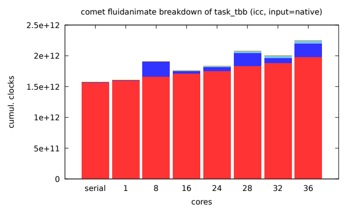
(e) mth



(f) omp



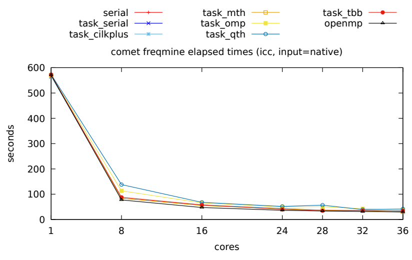
(g) qth



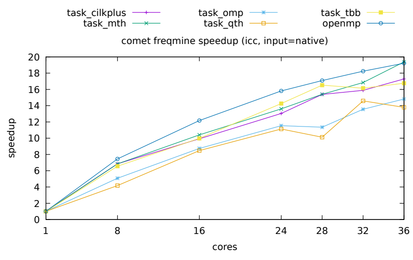
(h) tbb

Figure C.15: Breakdown of cumulative execution times (elapsed time × cores)

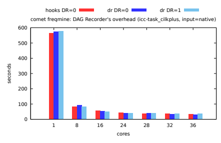
C.9 Freqmine

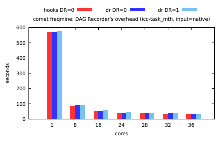


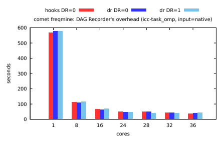
(a) elapsed times

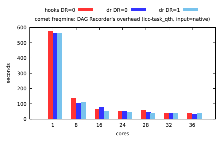


(b) speedups


(c) cilkplus


(d) mth


(e) omp


(f) qth

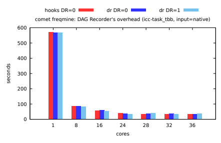
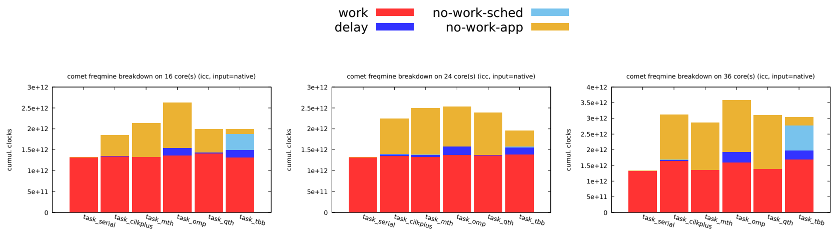
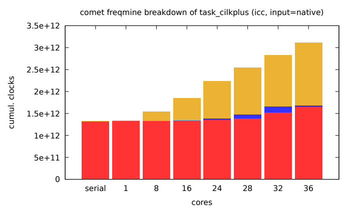

(g) tbb

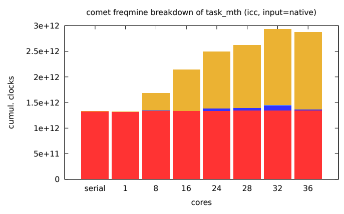
Figure C.16: Elapsed times, speedups, and profiling overheads



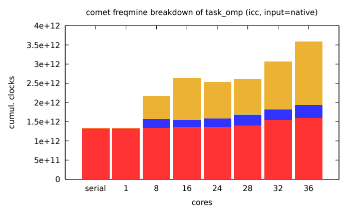
(a) on 16 cores (b) on 24 cores (c) on 36 cores



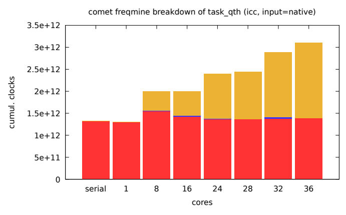
(d) cilkplus



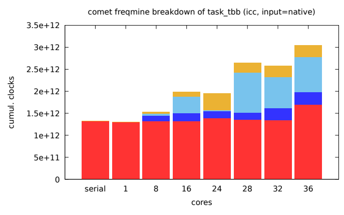
(e) mth



(f) omp



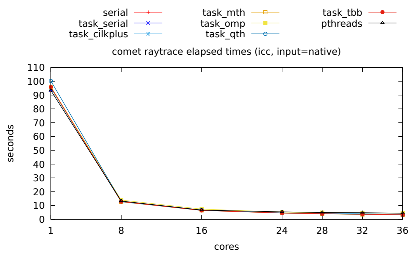
(g) qth



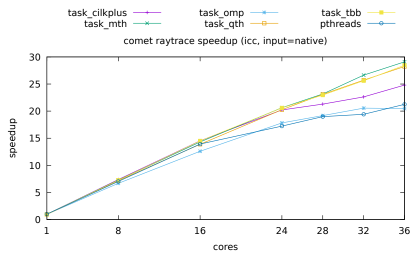
(h) tbb

Figure C.17: Breakdown of cumulative execution times (elapsed time × cores)

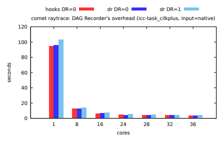
C.10 Raytrace



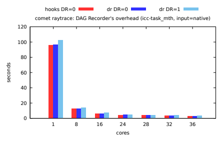
(a) elapsed times



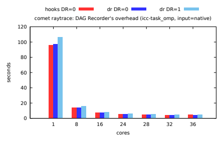
(b) speedups



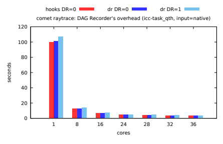
(c) cilkplus



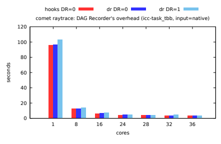
(d) mth



(e) omp

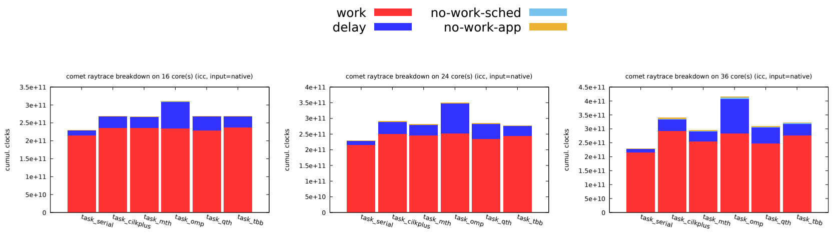


(f) qth

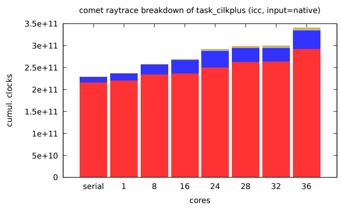


(g) tbb

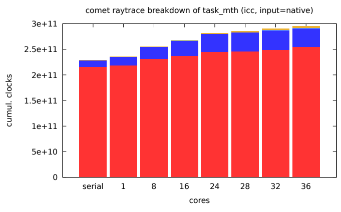
Figure C.18: Elapsed times, speedups, and profiling overheads



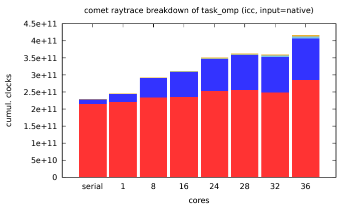
(a) on 16 cores (b) on 24 cores (c) on 36 cores



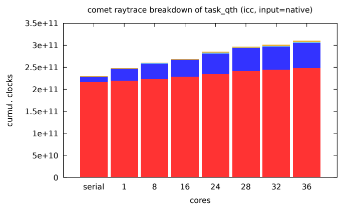
(d) cilkplus



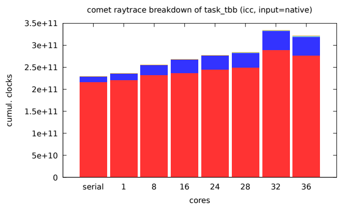
(e) mth



(f) omp



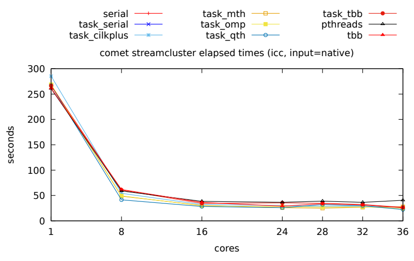
(g) qth



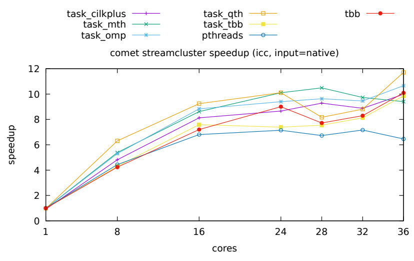
(h) tbb

Figure C.19: Breakdown of cumulative execution times (elapsed time × cores)

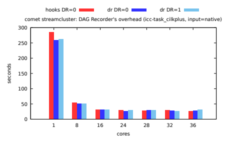
C.11 Streamcluster



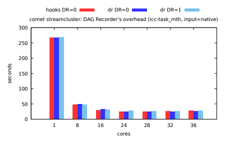
(a) elapsed times



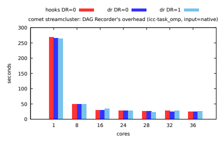
(b) speedups



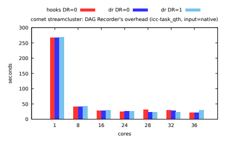
(c) cilkplus



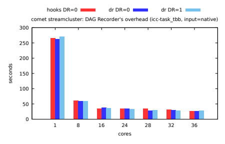
(d) mth



(e) omp

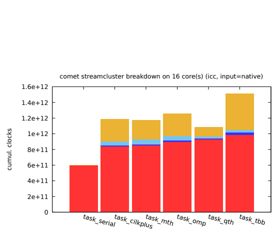


(f) qth

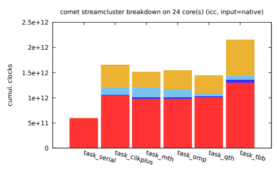


(g) tbb

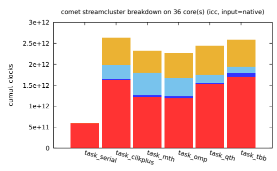
Figure C.20: Elapsed times, speedups, and profiling overheads



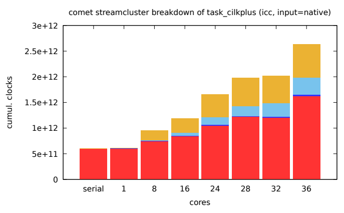
(a) on 16 cores



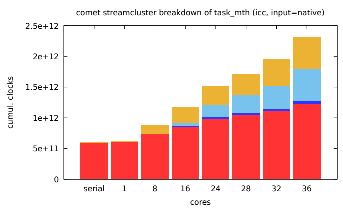
(b) on 24 cores



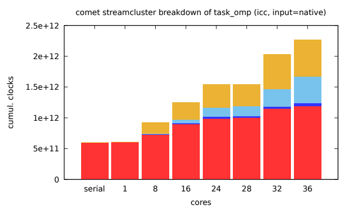
(c) on 36 cores



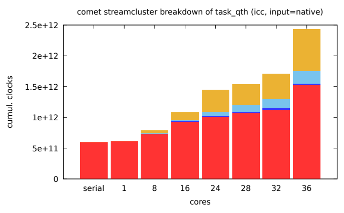
(d) cilkplus



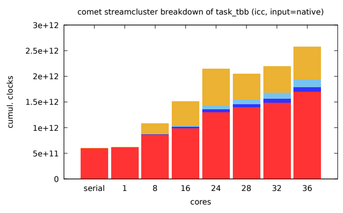
(e) mth



(f) omp



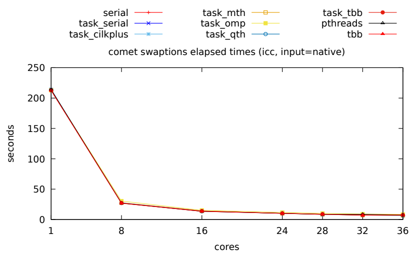
(g) qth



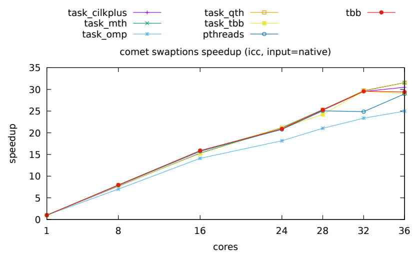
(h) tbb

Figure C.21: Breakdown of cumulative execution times (elapsed time × cores)

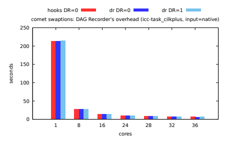
C.12 Swaptions



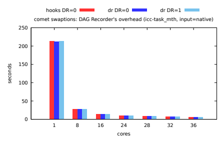
(a) elapsed times



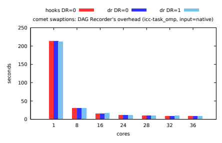
(b) speedups



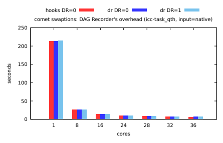
(c) cilkplus



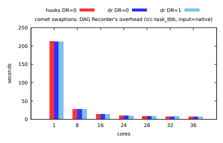
(d) mth



(e) omp

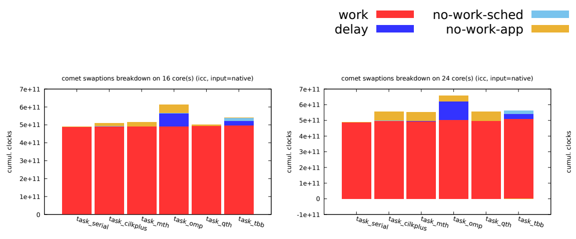


(f) qth

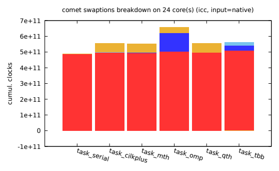


(g) tbb

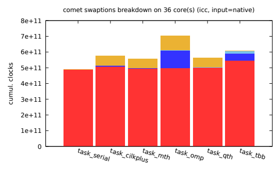
Figure C.22: Elapsed times, speedups, and profiling overheads



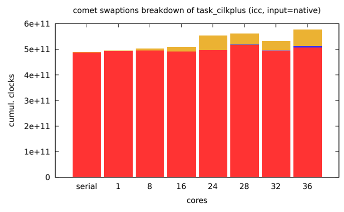
(a) on 16 cores



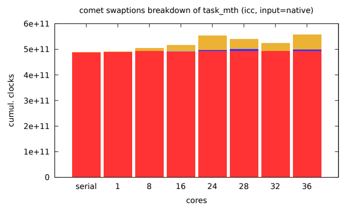
(b) on 24 cores



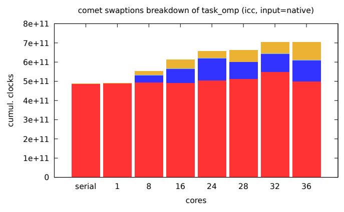
(c) on 36 cores



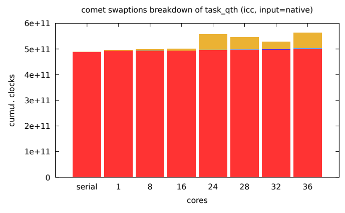
(d) cilkplus



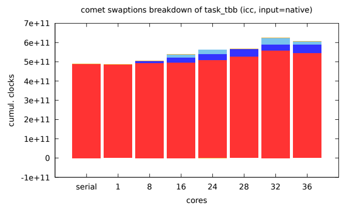
(e) mth



(f) omp



(g) qth



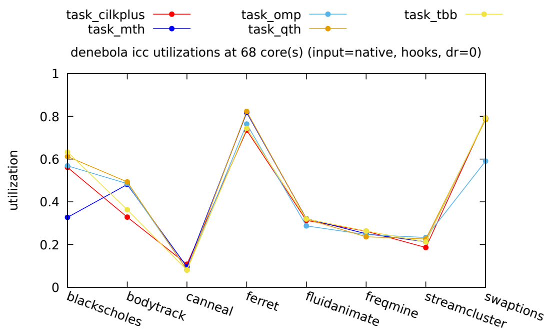
(h) tbb

Figure C.23: Breakdown of cumulative execution times (elapsed time × cores)

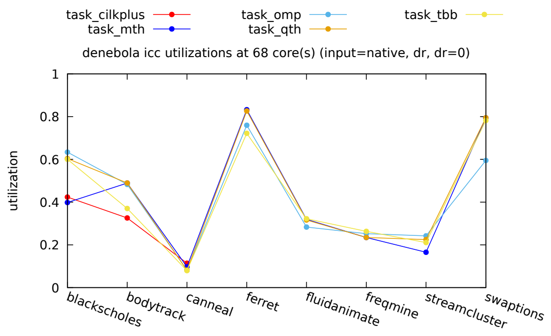
Appendix D

TP-PARSEC on Xeon Phi 7250 (Knights Landing)

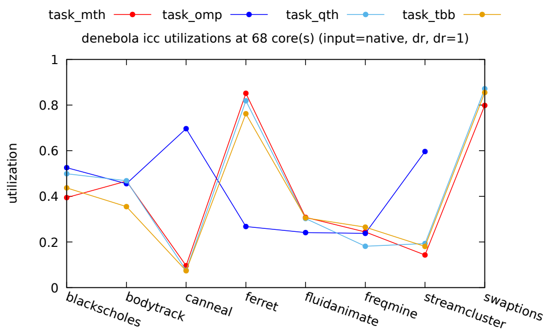
D.1 Overview



(a) compiled without DR



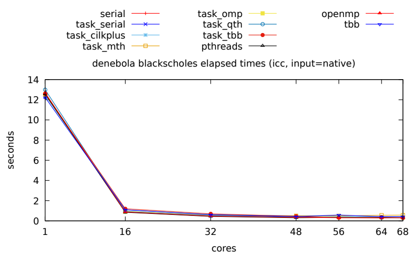
(b) DR off at runtime (DR=0)



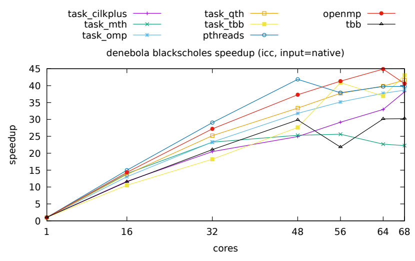
(c) DR on at runtime (DR=1)

Figure D.1: utilizations on 68 core(s)

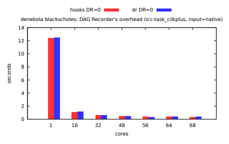
D.2 Blackscholes



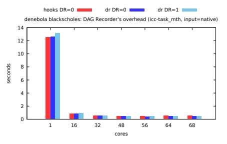
(a) elapsed times



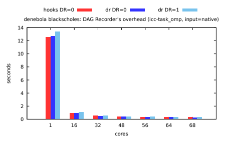
(b) speedups



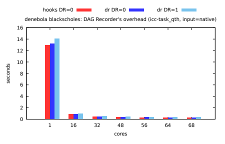
(c) cilkplus



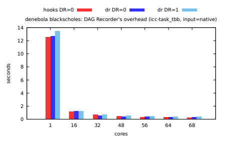
(d) mth



(e) omp

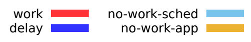


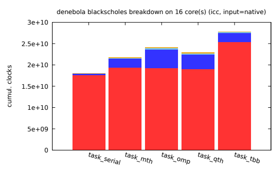
(f) qth



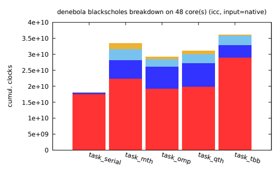
(g) tbb

Figure D.2: Elapsed times, speedups, and profiling overheads

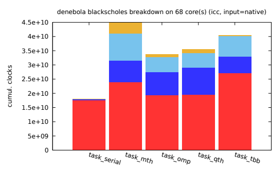




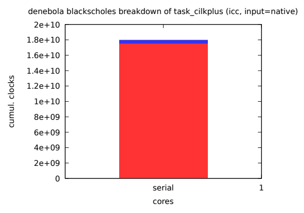
(a) on 16 cores



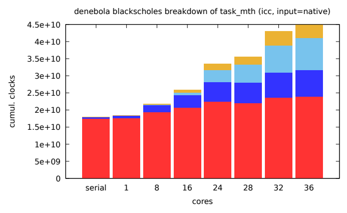
(b) on 48 cores



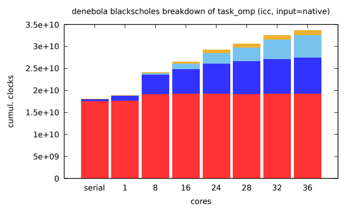
(c) on 68 cores



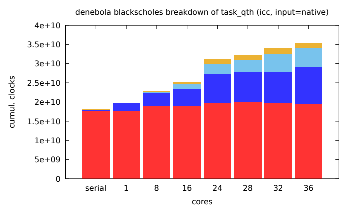
(d) cilkplus



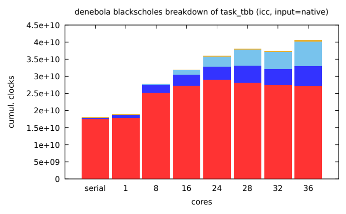
(e) mth



(f) omp



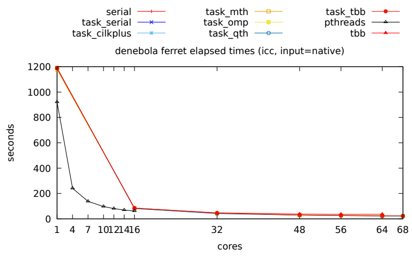
(g) qth

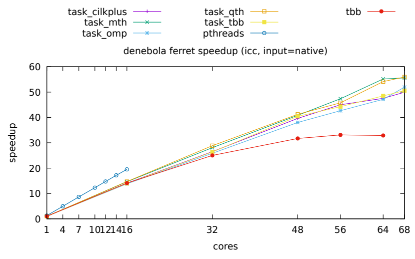


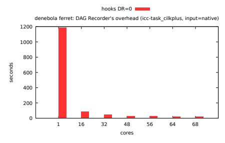
(h) tbb

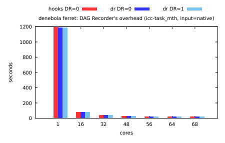
Figure D.3: Breakdown of cumulative execution times (elapsed time × cores)

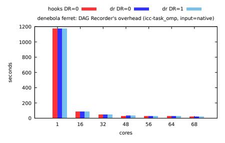
D.3 Ferret

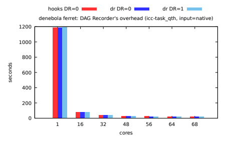

(a) elapsed times


(b) speedups


(c) cilkplus


(d) mth


(e) omp


(f) qth

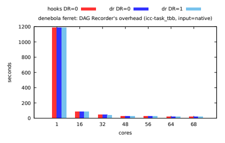
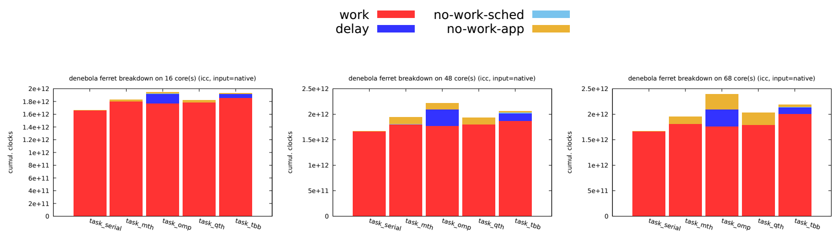
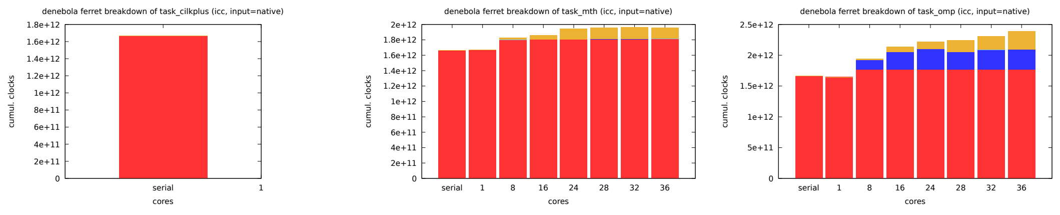

(g) tbb

Figure D.4: Elapsed times, speedups, and profiling overheads


(a) on 16 cores (b) on 48 cores (c) on 68 cores


(d) cilkplus (e) mth (f) omp

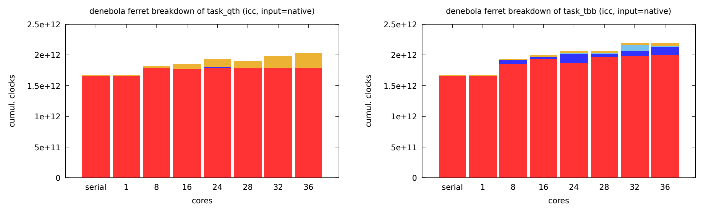
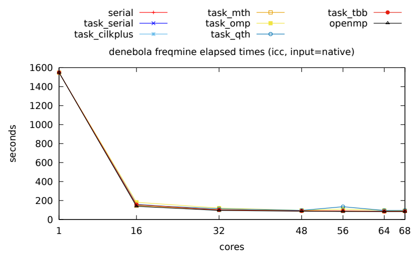
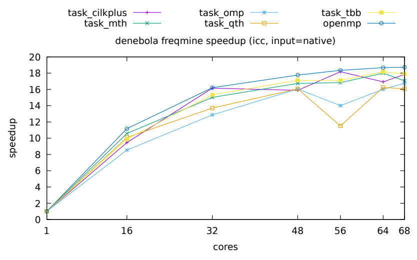
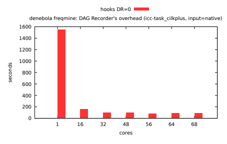

(g) qth (h) tbb

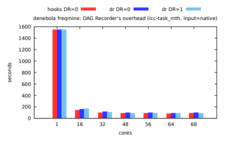
Figure D.5: Breakdown of cumulative execution times (elapsed time × cores)

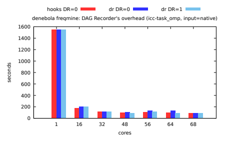
D.4 Freqmine

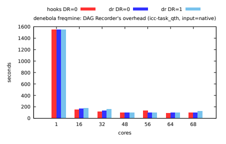

(a) elapsed times


(b) speedups


(c) cilkplus


(d) mth


(e) omp


(f) qth

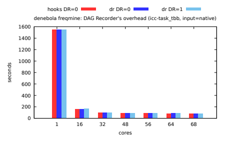
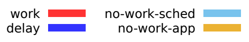
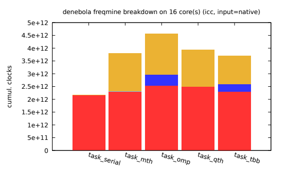
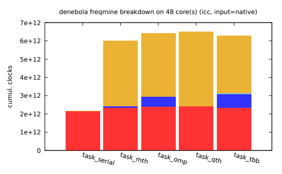
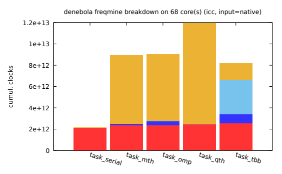

(g) tbb

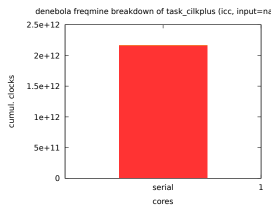
Figure D.6: Elapsed times, speedups, and profiling overheads

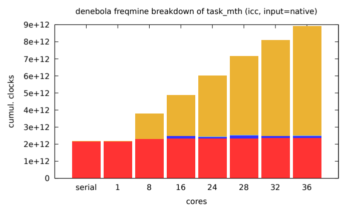


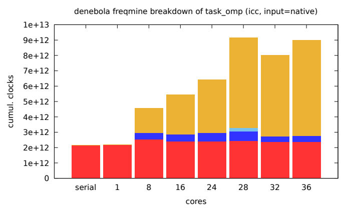

(a) on 16 cores

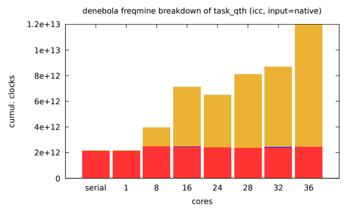

(b) on 48 cores


(c) on 68 cores


(d) cilkplus


(e) mth


(f) omp


(g) qth

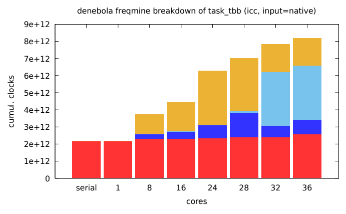
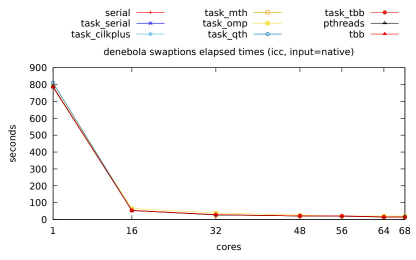
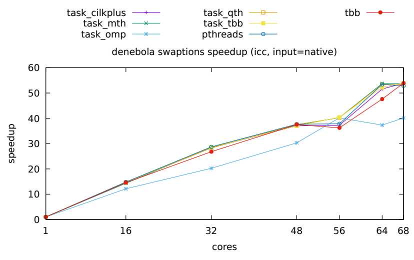

(h) tbb

Figure D.7: Breakdown of cumulative execution times (elapsed time × cores)

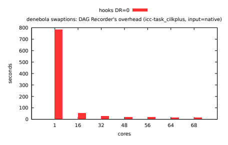
D.5 Swaptions



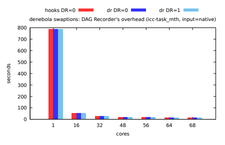
(a) elapsed times



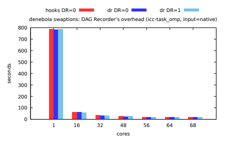
(b) speedups



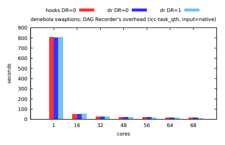
(c) cilkplus



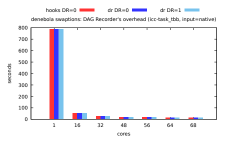
(d) mth



(e) omp

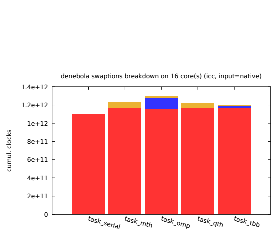


(f) qth

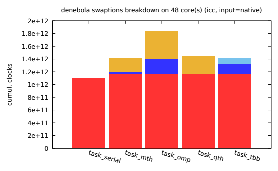


(g) tbb

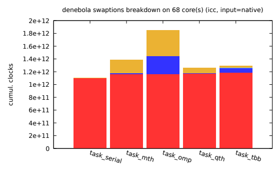
Figure D.8: Elapsed times, speedups, and profiling overheads



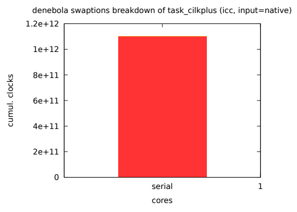
(a) on 16 cores



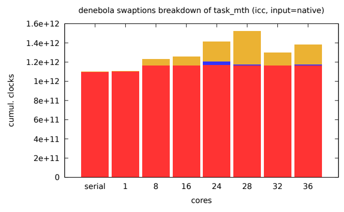
(b) on 48 cores



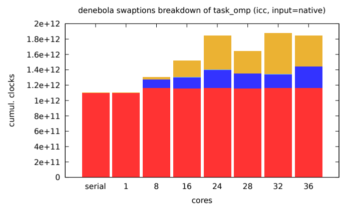
(c) on 68 cores



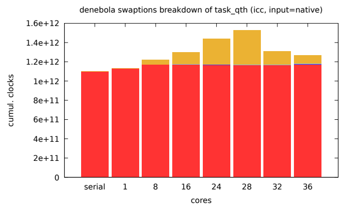
(d) cilkplus



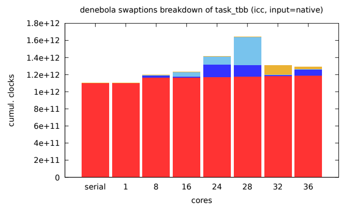
(e) mth



(f) omp



(g) qth



(h) tbb

Figure D.9: Breakdown of cumulative execution times (elapsed time × cores)