Journal of Information Processing Vol.22 No.4 1–10 (July 2014)

[DOI: 10.2197/ipsjjip.22.1]

Regular Paper

TP-PARSEC: A Task Parallel PARSEC Benchmark Suite

An Huynh[1,a) Christian Helm[1,b) Shintaro Iwasaki[1,c) Wataru Endo[1,d)
Byambajav Namsraijav[1,e) Kenjiro Taura[1,f)

Received: April 1, 2018, Accepted: August 24, 2018

Abstract: The original PARSEC benchmark suite consists of a diverse and representative set of benchmark applications which are useful in evaluating shared-memory multicore architectures. However, it supports only three programming models: Pthreads (SPMD), OpenMP (parallel for), TBB (parallel for, pipeline), lacking support for emerging and widespread 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, and ensures a better load balance on a large number of processor cores with the proven work stealing scheduling technique. TP-PARSEC is not only useful for task parallel system developers to analyze their runtime systems with a wide range of workloads from diverse areas, but also enables them to compare performance differences between 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.

Keywords: benchmark suite, task parallelism, programming models, performance analysis, scheduling delay

1. Introduction

Multicore processors have been widespread, with increasingly many cores integrated on a processor chip. These higher core counts have put pressure on the software layer; more threads mean more contentions, more synchronizations, more inter-chip traffic, longer memory accesses, etc.; programmers need to be more careful in order to keep their parallel programs efficient. Task parallel programming models are a popular approach in shared-memory parallel programming. With task parallelism, programmers do not need to be aware of low-level details in the systems, e.g., how many threads there are, then manually crafting and scheduling the program's workload to the number of 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 task, they just need to focus on the program's logics to extract parallelism and denote them as tasks. Task parallel runtime systems will map these logical tasks onto available processor cores automatically and dynamically at runtime. This dynamic scheduling is the basis for hiding latencies and tolerating noise.

Task parallel programming models are promising to deliver both programmability and performance to a wider audience. However, there is still lots of work to do to improve their performance. They need good benchmarks in order to be developed in proper directions. A popular benchmark for task parallelism is the Barcelona OpenMP Tasks Suite (BOTS) [1], 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-suited to 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) [2] is a popular benchmark suite that contains representative workloads from a wide range of areas such as image recognition, financial analytics, physics simulation, and data mining. It has been extensively used in researches of multicore shared-memory systems. PARSEC is shipped with support for POSIX Threads (Pthreads), OpenMP, and Intel Threading Building Blocks (TBB); benchmarks in PARSEC are mainly programmed with the SPMD (single program multiple data) model based on Pthreads, parallel for loop models based on OpenMP, TBB, and pipeline models based on Pthreads, TBB. They lack the support for task parallel programming models. That is why we have task-parallelized PARSEC, and presented a new benchmark suite TP-PARSEC (Task Parallel PARSEC[*1]) which adds support for *task parallelism* based on five task parallel programming models (Cilk Plus, MassiveThreads, OpenMP Tasks, Qthreads, TBB). On one hand, TP-PARSEC extends the original PARSEC with emerging task 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 implementa-

[1] University of Tokyo, Hongo, Bunkyo, Tokyo 113–8654, Japan
[a)] huynh@eidos.ic.i.u-tokyo.ac.jp
[b)] christian@eidos.ic.i.u-tokyo.ac.jp
[c)] iwasaki@eidos.ic.i.u-tokyo.ac.jp
[d)] wendo@eidos.ic.i.u-tokyo.ac.jp
[e)] byambajav@eidos.ic.i.u-tokyo.ac.jp
[f)] tau@eidos.ic.i.u-tokyo.ac.jp

[*1] https://github.com/massivethreads/tp-parsec

© 2014 Information Processing Society of Japan

tions of different task parallel programming models.

The performance of a task parallel programming model depends substantially on its runtime task scheduler. Different runtime systems 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*, MassiveThreads' speedup is ∼63% better than Cilk Plus; in *canneal*, Qthreads performs ∼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 models/runtime systems, TP-PARSEC becomes a useful benchmark suite that enables system developers to compare their system with others, analyze performance differences, and improve their implementation. TP-PARSEC unifies different primitives of multiple models at the preprocessed macro level, which conveniently simplifies the conversion process to many models.

In order to support users effectively in analyzing these performance differences, we have integrated into TP-PARSEC a task-centric performance tool DAGViz [3], [4] which can contrast performance differences between systems with a novel scheduling delay analysis and spot responsible places on DAG visualizations.

The rest of the paper is structured as follows: section 2 will discuss the background of this work, section 3 will describe our proposed TP-PARSEC benchmark suite, section 4 will evaluate speedups and demonstrate some performance tunings using our performance tool, section 5 is related work, and the conclusion and future work are in section 6.

2. Background

2.1 PARSEC

The original PARSEC benchmark suite developed by Princeton University [2] is a large benchmark suite consisting of 13 parallel applications and kernels which 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 [5], OpenMP [6], and TBB [7] (summarized in **Table 1**). Most of Pthreads versions are implemented with the *SPMD* model in which the data space (e.g., loop iterations) is divided equally among available threads; whereas, *dedup* and *ferret* use *manual pipeline* models which are implemented manually upon threads; *facesim* and *raytrace* use *manual task queues* implemented upon threads. Besides a Pthreads version, some benchmarks also have an OpenMP version (*blackscholes*, *bodytrack*, *freqmine*) or a TBB version (*blackscholes*, *bodytrack*, *ferret*, *fluidanimate*, *streamcluster*, *swaptions*). OpenMP versions use OpenMP's parallel loop model (*omp parallel for* directive). TBB versions use either TBB's parallel loop model (*tbb::parallel_for*), pipeline model (*tbb::pipeline*), or its low-level tasking interface (*tbb::task*).

2.2 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, and are dynamically scheduled on available processor cores at runtime by the runtime system's task scheduler. As tasks can be created at arbitrary points, typical parallelism patterns such as for loops and recursions can be expressed easily by tasks. Dynamic and automatic load balancing does not only relieve programmers from manual scheduling burdens, but also is the key to hiding latencies and runtime noise. Thus, task parallelism is promising to deliver both performance and productivity.

Many task parallel programming models exist. They have different concepts for scheduling and load balancing; and they differ substantially in their design and implementation. 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 [8] is a language extension of C/C++ with two simple keywords for expressing task parallelism (`cilk_spawn`, `cilk_sync`); (2) MassiveThreads [9] [10] is a lightweight thread library, and like Cilk Plus, uses the work-first work stealing strategy for scheduling tasks; (3) OpenMP [6] is a widely-used framework for shared-memory parallel programming (task parallelism has been introduced in OpenMP from version 3.0); (4) Qthreads [11] [12] is also a lightweight thread library, with a locality-aware scheduler; (5) TBB [7] is a commercial threading library equipped with a wide range of ready-made parallel patterns and algorithms.

3. TP-PARSEC

TP-PARSEC is based on the core package of PARSEC 3.0 (latest version as of February 2018), excluding input datasets which can be downloaded separately from the PARSEC website.

3.1 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 primitives, then the program can be preprocessed automatically into supported underlying systems. The wrapper includes two basic primitives: `create_task` for creating a task, and `wait_tasks` for synchronizing tasks (of the current scope). These primitives are translated to the corresponding API of specific models (**Table 2**) during the preprocessing stage of the compilation.

In addition, we also introduce the `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.

3.2 Task-Parallelizing PARSEC

In this section, we describe the computation model of each

© 2014 Information Processing Society of Japan

Table 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

Table 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()

Table 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	ø			ø
facesim	N/P			N/P
ferret	ø		ø	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; ø: none)

benchmark, how it is implemented in the original Pthreads, OpenMP, and 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, and P to denote the number of threads. The grain size (work granularity) of the SPMD model is N/P because the SPMD model divides data space (N) into P equal parts for P threads to execute. A summary of programming models used in each benchmark is shown in Table 1. A summary of the grain size set for each benchmark is shown in **Table 3**.

3.2.1 Blackscholes

Blackscholes is a workload in computational finance, and it calculates 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 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 in its OpenMP version and TBB's parallel for template in its TBB version. In the 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 Section 4.

3.2.2 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* recognizes poses of the human body in the input images, marks these poses and returns annotated images.

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, and frame after frame. The 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 available. The worker threads compete with each other through a mutex lock to acquire the next part to execute until all iterations are processed. The OpenMP and TBB versions use their parallel for primitives for all five loops. The TBB version additionally employs a pipeline model on the program's three stages.

In our first implementation of task versions, we only parallelized the five for loops (with `pfor`) just like the OpenMP ver-

© 2014 Information Processing Society of Japan

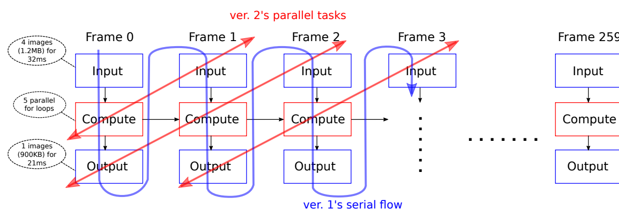


Fig. 1 *Bodytrack*'s general computation flows.

sion, 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. 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 $t+1$. This overlapping could be done easily with task parallelism:

```
for (i = frames [ 0 → 259 ]) {
  create_task( Input(i+1) );
  create_task( Compute(i) );
  create_task( Output(i-1) );
  wait_tasks;
}
```

We have made each stage a separate task and run three tasks *process*(t), *write*(t-1), and *read*(t+1) in parallel (**Fig. 1**). By making this change we could reduce serial intervals considerably. The performance improvement is discussed in Section 4.

3.2.3 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. Only Pthreads version is available for *canneal* in PARSEC, no OpenMP and TBB implementations are provided. Ptheads version divides 15000 moves equally for participating threads. Our task versions divide these moves 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.

3.2.4 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*, each stage is associated with a pool of threads. 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.

3.2.5 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 in total 21 loops in the program code; in the native input run, with 100 frames (time steps), these loops were invoked in total 61601 times. These loops generally apply specific 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.

3.2.6 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

© 2014 Information Processing Society of Japan

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 provided with two versions Pthreads and TBB. In these versions, the benchmark is organized as a pipeline model handling a series of 3500 input images. 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 exactly 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 text to file, so they actually run quickly and do not leave any noticeably long serial interval at the end of the execution.

3.2.7 Fluidanimate

Fluidanimate is a stencil computation which operates on a 3-dimensional 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. 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.

3.2.8 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 as 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).

3.2.9 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 pixel in the image to determine if it can reach back to any of the light sources. In this benchmark, 200 continuous frames are rendered, and 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 in total 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 the x-axis, then the 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 there are in Pthreads version. These leaf tasks are not created all at once, but recursively. At each recursive stage, the 2-dimensional 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. That is why the data parallelism between frames are not exploited in the benchmark.

3.2.10 Streamcluster

Streamcluster is a kernel solving the clustering problem commonly seen in data mining workloads. In this benchmark, there are in total 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* 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 basically uses the SPMD model. In task versions, we follow TBB version's model, and apply `pfor` in place of *tbb::parallel_for*, `create_task` in place of *tbb::task*. However, we set the grain size for `pfor`(s) at a low value 50, in order

© 2014 Information Processing Society of Japan

to make fine-grained tasks.

3.2.11 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 also apply `pfor` with grain size 1. One note is that *swaptions* code is currently not auto-vectorized at compile time by both gcc and icc, whereas *blackscholes* code is auto-vectorized by icc (not gcc). There are three reasons why icc can auto-vectorize *blackscholes* but *swaptions*: (1) *swaptions*' compute functions are scattered in multiple source files, (2) the for loop in *swaptions* has multiple exits, (3) *swaptions*' data arrays are not aligned yet.

3.3 Performance Analysis Tool

TP-PARSEC is integrated with a task-centric performance analysis and visualization tool [3] [4]. 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 performance and pinpoint 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 [4]. 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 large DAG of the execution and inspect in detail any task that caused the problem.

Moreover, the tool provides a novel 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) which is the scheduling delay analysis described in our previous work [3]. 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 also 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 lack of 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.

3.4 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 the 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
```

4. Evaluation

We 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 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 are the actual parallelized parts of each benchmark, and predefined in PARSEC. All benchmarks are executed using the largest input set (*native*). The speedup results of 11 benchmarks with all original versions and task versions are shown in **Fig. 2**. 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 specified number of threads n, *dedup* and *ferret* actually deploy n threads for each of their pipeline stages (except first and last ones). In their speedup figures (**Fig. 2d**, **Fig. 2f**), 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*.

4.0.1 Set grain size with delay metric (*blackscholes*)

One common pitfall of task parallelism is that too many fine-grained tasks created 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, and we

© 2014 Information Processing Society of Japan

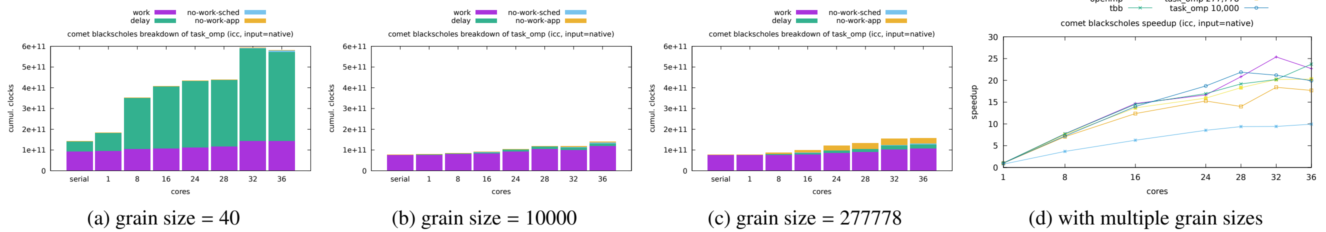
(a) blackscholes (b) bodytrack (c) canneal

(d) dedup (e) facesim (f) ferret

(g) fluidanimate (h) freqmine (i) raytrace

(j) streamcluster (k) swaptions

Fig. 2 Speedups of all versions of all benchmarks in TP-PARSEC



(a) grain size = 40 (b) grain size = 10000 (c) grain size = 277778 (d) with multiple grain sizes

Fig. 3 Blackscholes: task_omp's breakdowns and speedups with multiple grain sizes

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. 3a**) helped reveal the reason clearly: a huge delay incurred in task versions (we show the results of task_omp because it has the largest delay, and other systems incur around a half of it). We right away noticed the grain size and tried to

© 2014 Information Processing Society of Japan

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. 3c** shows the breakdown with this grain size; delay was reduced considerably; however, no-work-app has also increased. This increase of no-work-app is the result of coarse-grained tasks, so we decreased the grain size. After trying with many values, we got the best results at around 10000, whose breakdown is shown in **Fig. 3b**: minimal delay, minimal no-work-app. **Fig. 3d** shows the speedups of original versions together with task_omp version at three different grain sizes. This is a demonstration of the severe affect that task granularity may have on the performance, and our performance tool, specifically the scheduling delay metrics, helps effectively in signaling it.

4.0.2 Overlap I/O and computation 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. 4c**); the execution time breakdown had large no-work-app (**Fig. 4a**) due to long serial execution intervals which can be observed in the timelines and parallelism profile visualizations in **Fig. 4d**. 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 result is fantastic: speedups increase 2.5 times up to around 20x (**Fig. 2b**); no-work-app reduces considerably (**Fig. 4b**); the overlapped *read* and *write* stages can even be seen visibly in the timelines visualization (**Fig. 4e**).

Dedup suffers from the same problem with large no-work-app (**Fig. 5a**). It does not scale above 5.5x even when executed on full 36 cores (Fig. 2d). Its timelines visualization in **Fig. 5b** shows a very long serial interval at the end of its execution. An inspection into the code region according to the task of that serial interval has revealed the responsible instruction: file-closing function *close()*. When a file descriptor is closed, its buffer in memory actually gets flushed to disk. *Dedup* modified a very large buffer, so the flush takes a long time. In this situation it can be said that dedup's performance is bound by the disk's bandwidth.

4.0.3 Genuine task parallel schedulers perform 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 task or (2) assign tasks directly to worker threads based on a simple round-robin manner. In task versions, we have replaced these manual task queues with specialized and optimized 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 over manual self-implementations. In *bodytrack*, Qthreads and MassiveThreads-based task versions perform better than the original versions (Fig. 4c); in *facesim*, all task versions perform better than the original version (**Fig. 2e**); in *raytrace*, all task versions except task_omp perform better than the original version (**Fig. 2i**). A genuine task parallel runtime system usually uses the work stealing technique to balance load 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 remote workers try stealing from the other end, hence reducing thread contentions that interfere with computation progress. Recursive task creations in task versions (e.g., `pfor`) may also have contributed partly to the efficiency thanks to its possibly better locality.

4.0.4 Characterize performance differences with scheduling delay analysis

In some benchmarks, task versions perform similarly with each other, 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. 2). **Fig. 6** contrasts the differences of 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 slower work stealing speed which was also pointed out in SparseLU benchmark in [3]. 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 more detailed discussion on differences between runtime implementations is out of the scope of this paper, but planned in our future work.

5. Related Work

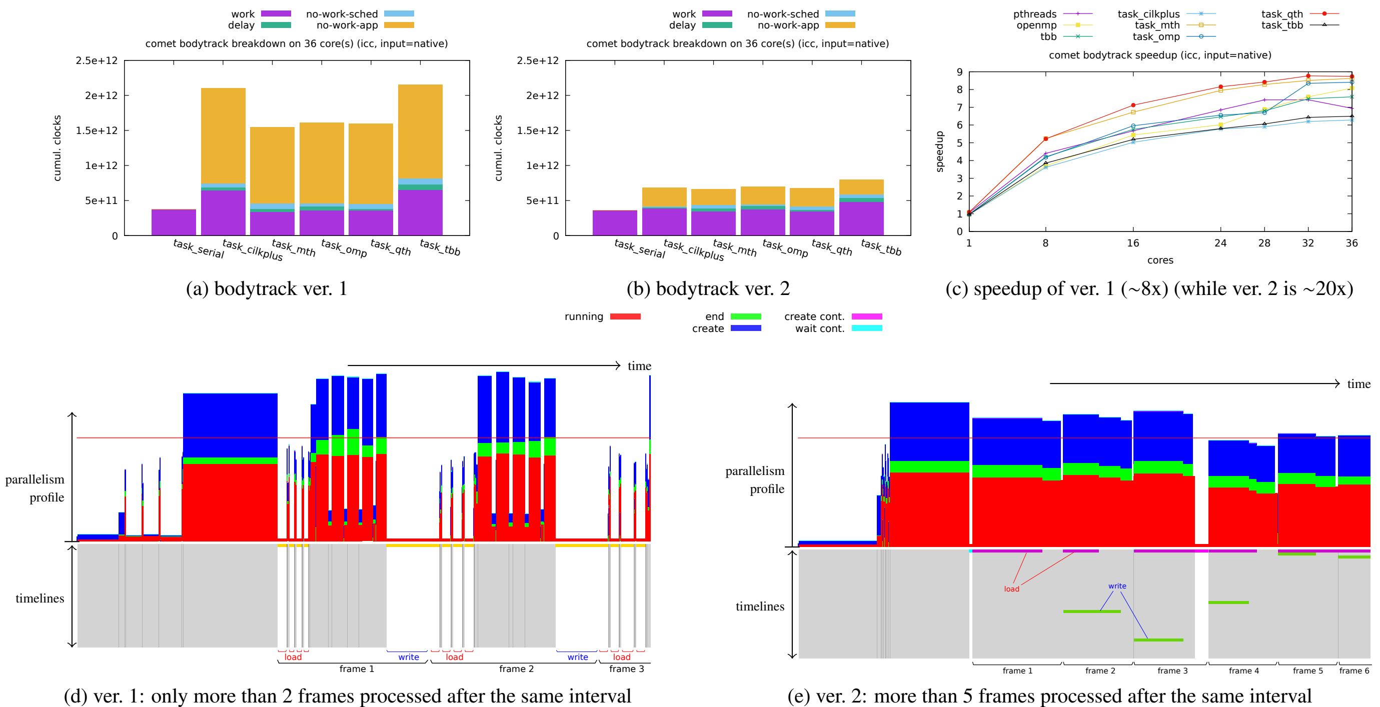
Barcelona OpenMP Tasks Suite (BOTS) [1] is a popular task parallel benchmark suite which consists of 10 applications. Most of the applications are simple divide-and-conquer algorithms parallelized only by OpenMP Tasks. They are not representative of realistic applications and do not support other task parallel programming models.

PARSECSs [13] 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 a reduction in lines of code) which makes it less than a complete benchmark suite. TP-PARSEC is not only a showcase to demonstrate the advantages of task parallelism over SPMD, manual pipelines, and manual task queues; but also is intended as a full-fledged benchmark suite for general usage.

Lee et al. [14] have ported three of the PARSEC applications to a pipelined 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. Majo et al. [15] introduced optimizations regarding NUMA and prefetching for three of the PAR-

© 2014 Information Processing Society of Japan



(a) bodytrack ver. 1

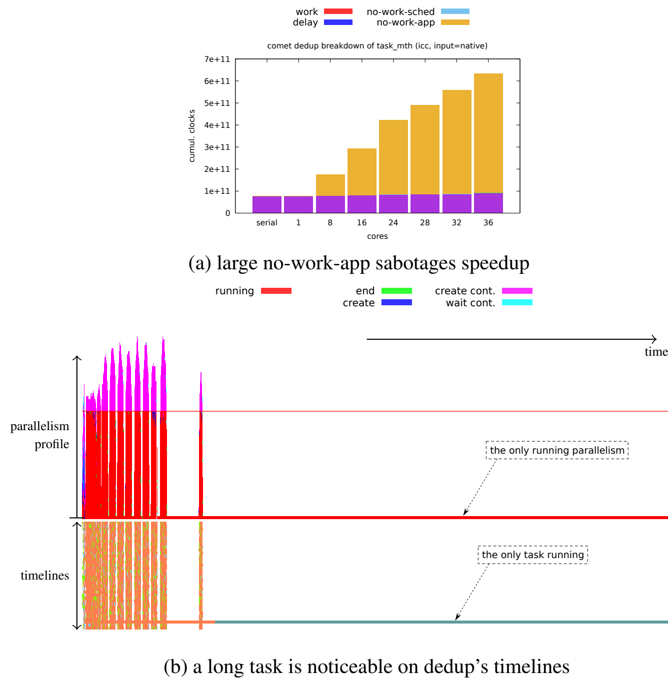
(b) bodytrack ver. 2

(c) speedup of ver. 1 (~8x) (while ver. 2 is ~20x)

(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

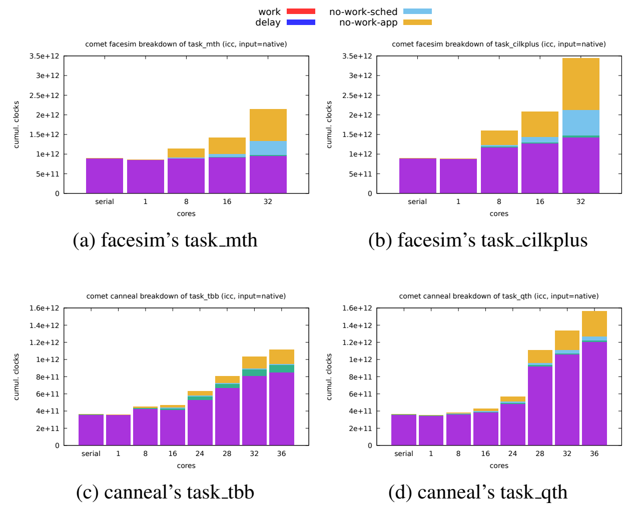
Fig. 4 Bodytrack: ver. 2 improves substantially over ver. 1 by overlapping I/O tasks with computation tasks. Gray areas represent unexpanded computation nodes whose start time and end time are known at this level of detail, but workers on which their sub-graphs of tasks were executed are not.



(a) large no-work-app sabotages speedup

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

Fig. 5 *Dedup* has a long serial interval at the end of the execution due to file I/O (flushing memory buffer to file).



(a) facesim's task_mth

(b) facesim's task_cilkplus

(c) canneal's task_tbb

(d) canneal's task_qth

Fig. 6 Performance variation between task versions in *facesim* and *canneal*

SEC benchmarks. Southern et al. [16] did a scalability analysis of PARSEC benchmarks based on input sizes. Barrow-Williams et al. [17] examined data sharing patterns. Bhadauria et al. [18] evaluated PARSEC benchmarks using hardware performance counters. A vectorized version of PARSEC is introduced and characterized using hardware performance counters by Cebrian et al. [19]. The PARSEC paper [2] 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.

6. Conclusion

We have presented TP-PARSEC–a benchmark suite extended from PARSEC with support 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.

© 2014 Information Processing Society of Japan

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.

Acknowledgments This work was in part supported by Grant-in-Aid for Scientific Research (A) 16H01715.

References

[1] Duran González, A., Teruel, X., Ferrer, R., Martorell Bofill, X. and Ayguadé Parra, E.: Barcelona openmp tasks suite: A set of benchmarks targeting the exploitation of task parallelism in openmp, *38th International Conference on Parallel Processing*, pp. 124–131 (2009).

[2] Bienia, C., Kumar, S., Singh, J. P. and Li, K.: The PARSEC benchmark suite: characterization and architectural implications, *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, ACM, pp. 72–81 (2008).

[3] Huynh, A. and Taura, K.: Delay Spotter: A Tool for Spotting Scheduler-Caused Delays in Task Parallel Runtime Systems, *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, pp. 114–125 (online), DOI: 10.1109/CLUSTER.2017.82 (2017).

[4] Huynh, A., Thain, D., Pericàs, M. and Taura, K.: DAGViz: A DAG Visualization Tool for Analyzing Task-parallel Program Traces, *Proceedings of the 2nd Workshop on Visual Performance Analysis*, VPA '15, New York, NY, USA, ACM, pp. 3:1–3:8 (online), DOI: 10.1145/2835238.2835241 (2015).

[5] Nichols, B., Buttlar, D. and Farrell, J.: *Pthreads programming: A POSIX standard for better multiprocessing*, " O'Reilly Media, Inc." (1996).

[6] Dagum, L. and Menon, R.: OpenMP: an industry standard API for shared-memory programming, *IEEE computational science and engineering*, Vol. 5, No. 1, pp. 46–55 (1998).

[7] Intel: Intel Threading Building Blocks (TBB), Intel Corp. (online), available from ⟨https://www.threadingbuildingblocks.org⟩ (accessed 2018-2-9).

[8] Intel: Intel Cilk Plus, Intel Corp. (online), available from ⟨https://www.cilkplus.org/⟩ (accessed 2018-2-9).

[9] Nakashima, J. and Taura, K.: MassiveThreads: A thread library for high productivity languages, *Concurrent Objects and Beyond*, Springer, pp. 222–238 (2014).

[10] MassiveThreads: Light weight thread library, University of Tokyo (online), available from ⟨https://github.com/massivethreads/massivethreads⟩ (accessed 2018-2-9).

[11] Wheeler, K. B., Murphy, R. C. and Thain, D.: Qthreads: An API for programming with millions of lightweight threads, *2008 IEEE International Symposium on Parallel and Distributed Processing*, pp. 1–8 (online), DOI: 10.1109/IPDPS.2008.4536359 (2008).

[12] Qthreads: Lightweight locality-aware user-level threading runtime, Sandia National Laboratories (online), available from ⟨https://github.com/Qthreads/qthreads⟩ (accessed 2018-2-9).

[13] Chasapis, D., Casas, M., Moretó, M., Vidal, R., Ayguadé, E., Labarta, J. and Valero, M.: PARSECSs: Evaluating the impact of task parallelism in the PARSEC benchmark suite, *ACM Transactions on Architecture and Code Optimization (TACO)*, Vol. 12, No. 4, p. 41 (2016).

[14] Lee, I.-T. A., Leiserson, C. E., Schardl, T. B., Zhang, Z. and Sukha, J.: On-the-Fly Pipeline Parallelism, *ACM Transactions on Parallel Computing*, Vol. 2, No. 3, pp. 17:1–17:42 (2015).

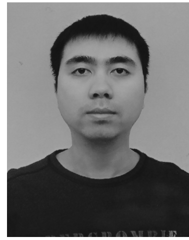
[15] Majo, Z. and Gross, T. R.: (Mis) understanding the NUMA memory system performance of multithreaded workloads, *Workload Characterization (IISWC), 2013 IEEE International Symposium on*, IEEE, pp. 11–22 (2013).

[16] Southern, G. and Renau, J.: Analysis of PARSEC workload scalability, *Performance Analysis of Systems and Software (ISPASS), 2016 IEEE International Symposium on*, IEEE, pp. 133–142 (2016).

[17] Barrow-Williams, N., Fensch, C. and Moore, S.: A communication characterisation of Splash-2 and Parsec, *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, IEEE, pp. 86–97 (2009).

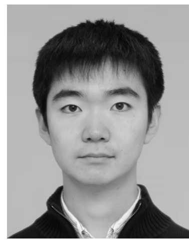
[18] Bhadauria, M., Weaver, V. M. and McKee, S. A.: Understanding PARSEC performance on contemporary CMPs, *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, IEEE, pp. 98–107 (2009).

[19] Cebrian, J. M., Jahre, M. and Natvig, L.: ParVec: vectorizing the PARSEC benchmark suite, *Computing*, Vol. 97, No. 11, pp. 1077–1100 (2015).

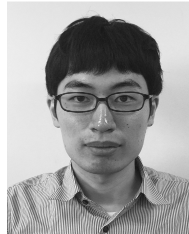


An Huynh is a third-year Ph.D. student at the Department of Information and Communication Engineering, University of Tokyo. He received his B.S. and M.S. degrees from the University of Tokyo in 2013 and 2015, respectively. His research interests include parallel programming and performance analysis.

Christian Helm is a Ph.D. student at the Department of Information and Communication Engineering, University of Tokyo. He received his B.S. and M.S. degrees from the University of Applied Sciences Regensburg. His research interests include performance analysis, memory systems and parallel processing.



Shintaro Iwasaki is a Ph.D. candidate at University of Tokyo in Japan. He received his B.S. and M.S. degrees from University of Tokyo in 2015 and 2017, respectively. His current research interests include parallel languages, compilers, runtime systems, and scheduling techniques.



Wataru Endo is a Ph.D. student at University of Tokyo. He received his B.S. and M.S. degrees from the University of Tokyo in 2015 and 2017, respectively. His current research interests include distributed memory systems, memory coherence, and communication performance.

Byambajav Namsraijav completed his Masters and undergraduate studies at the University of Tokyo. During his Masters, his research focus was on task parallel computing and its performance prediction. Before that, he researched about information-centric networking security.

Kenjiro Taura is a professor at the Department of Information and Communication Engineering, University of Tokyo. He received his B.S., M.S., and D.Sc. degrees from the University of Tokyo in 1992, 1994, and 1997, respectively. His major research interests are centered on parallel/distributed computing and programming languages. He is a member of ACM and IEEE.

© 2014 Information Processing Society of Japan