Improving Cache Utilization of Nested Parallel Programs by Almost Deterministic Work Stealing

Shumpei Shiina, *Graduate Student Member, IEEE* and Kenjiro Taura

Abstract—Nested (fork-join) parallelism eases parallel programming by enabling high-level expression of parallelism and leaving the mapping between parallel tasks and hardware to the runtime scheduler. A challenge in dynamic scheduling of nested parallelism is how to exploit data locality, which has become more demanding in the deep cache hierarchies of modern processors with a large number of cores. This paper introduces *almost deterministic work stealing (ADWS)*, which efficiently exploits data locality by deterministically planning a cache-hierarchy-aware schedule, while allowing a little scheduling variety to facilitate dynamic load balancing. Furthermore, we propose an extension of our prior work on ADWS to achieve better shared cache utilization. The improved version of the scheduler is called *multi-level ADWS*. The idea is that only part of a computation whose working set size is small enough to fit into a shared cache is scheduled by ADWS within the cache recursively, thus avoiding excessive capacity misses. Our evaluation on a benchmark of parallel decision tree construction demonstrated that multi-level ADWS outperformed the conventional random work stealing of Cilk Plus by 61% and it showed a 40% performance improvement over the previous ADWS design.

Index Terms—Dynamic load balancing, locality, nested parallelism, task parallelism, task scheduling, work stealing

1 Introduction

Nested (fork-join) parallelism enables programmers to write high-level code to express many parallel algorithms, including parallel divide-and-conquer algorithms (e.g., Quicksort). By using nested parallel constructs, we can straightforwardly express the recursively nested parallelism that is inherent in many parallel algorithms; at the same time, we can achieve better composability of parallel programs by arbitrarily nesting parallelism. Because the runtime system is responsible for mapping logically parallel computations to physical processor cores, programmers do not have to consider the actual mapping of computations, which reduces the burden of parallel programming. Moreover, this programming style often leads to good data locality, because the working set is divided into smaller ones as a problem is recursively decomposed. A representative set of algorithms that use this data locality property is known as *cache-oblivious algorithms* [1], [2]. While these algorithms are oblivious to specific cache hierarchies and sizes, they are proved to have asymptotically optimal bounds on the number of cache misses. The key to cache-oblivious algorithms is to organize data locality in a hierarchical way so that the data locality matches any cache configuration, and many parallel variants of cache-oblivious algorithms have been developed by using nested parallelism [3], [4].

• *The authors are with the Department of Information and Communication Engineering, Graduate School of Information Science and Technology, University of Tokyo, Bunkyo-ku, Tokyo 113-8654, Japan. E-mail: {shiina, tau}@eidos.ic.i.u-tokyo.ac.jp.*

Manuscript received 27 April 2022; revised 20 July 2022; accepted 28 July 2022. Date of publication 3 August 2022; date of current version 23 August 2022.
This work was supported in part by JSPS KAKENHI under Grant 21J22305 and in part by NEDO Project under Grant JPNP16007.
(Corresponding author: Shumpei Shiina.)
Recommended for acceptance by S. Chandrasekaran.
Digital Object Identifier no. 10.1109/TPDS.2022.3196192

Nevertheless, it is challenging to exploit the data locality of nested parallelism on the hierarchical caches of modern CPU architectures. *Work stealing* [5]—presumably the most widely used scheduling strategy for nested parallelism—is known to efficiently utilize private caches [6] but not hierarchical caches. Even on a processor with a single shared cache and multiple private caches, it has been reported [7] that work stealing is not optimal for data sharing on a shared cache, and that *parallel depth-first (PDF) scheduling* [8] outperforms work stealing. For deeper and more complicated cache hierarchies, however, neither work stealing nor PDF scheduling is optimal; accordingly, for arbitrary cache hierarchies, *space-bounded schedulers* [4], [9], [10] have been proposed.

Another aspect of the data locality of nested parallelism is that, when the same or similar computations are repeatedly executed on the same working set, parallel sub-computations (i.e., *tasks*) at the same position in the task hierarchy are likely to access the same data [6]. This finding is based on the observation that many algorithms exhibit almost the same memory access pattern for the same array across consecutive iterations. Previous works have investigated this direction: *constrained work stealing* [11] and *LAWS* [12], [13] improve data locality by scheduling tasks somewhat deterministically, but they are not designed for arbitrary cache hierarchies. Space-bounded schedulers, on the other hand, are designed for arbitrary cache hierarchies but do not account for iterative data locality. To the best of our knowledge, no scheduler has been designed for both iterative computations and arbitrary cache hierarchies at the same time.

Hence, to fulfill both of these requirements, we introduce a simpler and more straightforward approach, called *almost deterministic work stealing (ADWS)*. ADWS first deterministically maps tasks to processor cores so that the task hierarchy matches the cache hierarchy, which enables exploitation of data locality on hierarchical caches. This deterministic task mapping also enables exploitation of data locality for iterative

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

computations, because, at every iteration, data in the same location is likely to be accessed by the same core. A limitation of ADWS is that it requires programmers to specify hints on the relative amount of work for each task. However, these hints can be rough estimates, because ADWS also performs dynamic load balancing to fix load imbalances derived from the initial deterministic task mapping, which is why ADWS stands for *almost* deterministic work stealing.

By leveraging the prior published version of ADWS [14], this paper further extends ADWS to achieve better utilization of shared caches. The problem with the previous ADWS design is that it does not facilitate shared cache reuse when the overall working set size is much larger than the shared cache sizes. Accordingly, when a set of cores sharing a cache receives a task whose working set size is much larger than the shared cache, parallel execution of its descendants may result in a large number of capacity misses. In other words, ADWS may cause many more shared cache misses that would occur in serial execution, because cores sharing a cache may execute almost independent parts of a problem, so that almost no cache reuse is expected.

Such inefficiency in the use of shared caches is not specific to ADWS but also applies to work stealing and its variants. To date, many approaches have addressed this issue, including *CATS* [15] and space-bounded schedulers [4], [9], [10], to name a few. By borrowing the underlying concept of those approaches, we introduce a scheduling framework called *multi-level scheduling*, which generalizes the concept of *two-level scheduling* in the previous literature [12], [13], [15], [16], [17]. Whereas the previous concept of two-level scheduling assumed specific cache configurations and specific scheduling strategies, multi-level scheduling generalizes the concept for any level of a cache hierarchy, with arbitrary schedulers for each cache level. The idea of multi-level scheduling is that, for each shared cache, the cores that share it simultaneously execute only one task (and its descendants) whose working set size is small enough to fit into the shared cache, while leaving other tasks unscheduled. We assume that the working set size for each task is given by the programmer as a hint. Once a task is assigned to a shared cache, its descendants are scheduled recursively to the children of the cache. For each cache level, we can arbitrarily choose the scheduling policy to map tasks from the parent cache to the children. By using the framework of multi-level scheduling, we have straightforwardly devised an extension of ADWS, called *multi-level ADWS*, in which the previous version of ADWS (called *single-level ADWS* here) is applied to each cache level. Other variants of multi-level schedulers can also be considered by applying different scheduling strategies to each cache level: for example, we can consider *multi-level work stealing*, in which traditional random work stealing is applied hierarchically.

To empirically study the performance of our approach, we conducted a comprehensive performance analysis using seven benchmarks on a two-socket Cascade Lake machine. In this analysis, we compared five schedulers: single-level WS (traditional random work stealing), single-level ADWS, multi-level WS, multi-level ADWS, and a space-bounded scheduler (a port of the implementation by Simhadri et al. [18], [19]). The results show that multi-level ADWS outperformed the other schedulers on highly memory-bound benchmarks over a wide range of working set sizes. Notably, for decision tree construction with a large real-world dataset having a size of about 2 GB, both single- and multi-level ADWS outperformed Cilk Plus [20], a widely used tasking runtime with single-level WS, by 15% and 60%, respectively. These results also confirm the advantage of multi-level ADWS over single-level ADWS. Moreover, multi-level ADWS outperformed multi-level WS by 18%, which confirms the benefit of exploiting data locality for iterative computations. Other experiments demonstrated that ADWS is tolerant to load imbalances even when work hints are incorrect, and that NUMA local allocation policy can be exploited by deterministic scheduling of ADWS.

```
1  Function BuildDecisionTree(rows)
2      foreach attr ∈ attributes do
3          split ← ComputeBestSplit(rows, attr)
4          if split is better than bestSplit then
5              bestSplit ← split
6      if Splitting at bestSplit is beneficial then
7          (rowsL, rowsR) ← Partition(rows, bestSplit)
8          spawn nodeL ← BuildDecisionTree(rowsL)
9          spawn nodeR ← BuildDecisionTree(rowsR)
10         sync
11         return InnerNode(nodeL, nodeR)
12     else
13         return LeafNode(rows)
```

Fig. 1. Algorithm for parallel decision tree construction.

2 Background

In this section, we first introduce the preliminaries of scheduling for nested parallelism through an example of parallel construction of a decision tree.

2.1 Motivating Example: Decision Tree Construction

A decision tree can be constructed by a simple divide-and-conquer algorithm, that can be straightforwardly parallelized by nested fork-join constructs, as studied in [21]. This paper assumes the CART [22] algorithm for binary classification with continuous-valued attributes.

Fig. 1 shows a simplified algorithm for decision tree construction. It receives training data (*rows*) as input, each row of which consists of a class (0 or 1) and multiple attributes with continuous values. The resulting decision tree is then used to predict the class from given attributes. At line 7, the rows are partitioned into two parts on the basis of *bestSplit*, which specifies which column (attribute) is used to split the rows, and its threshold. Then, the partitioned rows (*rowsL* and *rowsR*) are used to recursively construct two subtrees, which can be computed in parallel (line 8–9). These two parallel tasks are created by the *spawn* keyword, and the wait for their completion is indicated by the *sync* keyword.

Before partitioning, *bestSplit* is chosen for the split with the lowest value of the Gini impurity [22], where this choice involves consecutive parallel computations (line 2–5). The best split is first determined for each attribute (*split* at line 3) and *bestSplit* is then chosen from among all the attributes. This paper assumes that the best split for each attribute is determined by constructing histograms [23], [24], as efficient implementations such as LightGBM [25] do, rather

```
task_group tg;
tg.run([]{ A(); });
tg.run([]{ B(); });
tg.run([]{ C(); });
tg.run([]{ D(); });
tg.wait();
```

(a) Task group in TBB [26]

```
task_group tg(w_all, size);
tg.run([]{ A(); }, w1);
tg.run([]{ B(); }, w2);
tg.run([]{ C(); }, w3);
tg.run([]{ D(); }, w4);
tg.wait();
```

(b) Extension for ADWS

Fig. 2. Task group notation and its extension for ADWS.

than by sorting the rows by each attribute. As a result, the ComputeBestSplit function at line 3 can be written as a simple, flat parallel loop (or parallel reduction) to construct a histogram. The Partition function can also be parallelized through double buffering as in Quicksort.

2.2 Nested Parallelism and Data Locality

Next, we discuss the basics of nested parallel computations and their properties of data locality. Although we used the *spawn-sync* style of Cilk [27], [28], [29] to express the fork-join constructs in Fig. 1, in the following, we instead use the *task group* constructs adopted by C++ library implementations such as Intel TBB [26]. The notation and concept of tasks and task groups are illustrated in Figs. 2 and 3, respectively. A *task* is a series of computations that should occur sequentially, and only one task (the *root* task) exists at the beginning. A task can spawn child tasks by creating a *task group*. A task group consists of any number of child tasks (`A`, `B`, `C`, and `D` in Fig. 2a), and a program waits for the completion of child tasks at the same time (`tg.wait()`). A task can sequentially create multiple task groups, whose executions cannot overlap within a task (i.e., a new task group can be created only after the previous one is completed, if it exists).

Fig. 4 shows an example of a computation graph for decision tree construction. As explained above, the procedure is based on a two-way divide-and-conquer algorithm. In the figure, the first (topmost) two flat parallel computations at each recursion represent the graphs for finding the best split (the ComputeBestSplit function at line 3 in Fig. 1), assuming only two attributes for each row. The third computation corresponds to partitioning (the Partition function at line 7), after which two tasks are spawned and the trees are recursively constructed. Note that this computation graph is input-dependent and not known in advance: the computation graph is dynamically unfolded during execution. Although the figure shows a graph that is well-balanced for simplicity, in reality it can be highly unbalanced, depending on the best split choice.

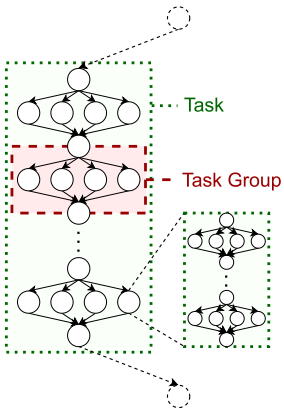


Fig. 3. Concept of nested parallel computations.

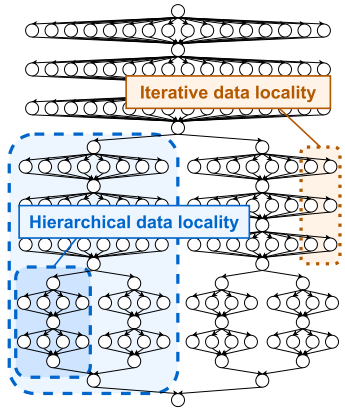


Fig. 4. Example of a computation graph for decision tree construction.

Fig. 4 also illustrates the data locality that resides in the computation graph. Here, we introduce two types of data locality: *hierarchical data locality* and *iterative data locality*. Hierarchical data locality specifies multiple levels of data locality in the task hierarchy: descendant tasks of the same parent are likely to access the same subset of the data, as the working set is usually decomposed recursively into smaller ones. Iterative data locality occurs when the same or similar parallel computations are repeatedly executed: tasks at the same position in the graph are likely to access the same data, as many algorithms exhibit almost the same memory access pattern across consecutive iterations. In Fig. 4, the tasks enclosed by the dashed line have hierarchical data locality, because the data rows are recursively partitioned into disjoint parts. In contrast, the tasks enclosed by the dotted line have iterative data locality, which typically exists in consecutive flat parallel loops, because tasks at the same position tend to access the same rows across multiple iterations. In the computations for finding the best split for each attribute, a series of data is repeatedly accessed in the same order.

2.3 Work Stealing

A task scheduler—which maps the tasks in a computation graph to actual hardware—should respect both the hierarchical and iterative data locality explained above, but this is challenging on the hierarchical caches of modern hardware. For an example of a scheduler for nested parallelism, we explain *work stealing*, which is arguably the most popular scheduling strategy for nested parallelism.

A *worker* is a virtualized processor core (or a hardware thread), which is typically substantiated as a kernel-level thread. In this paper, we assume that as many workers as the number of processor cores are created at the beginning of execution. In work stealing, each worker is equipped with a double-ended queue called a *task queue*, which stores local tasks that are ready to execute. When a worker spawns a new task, it may immediately execute the new task by

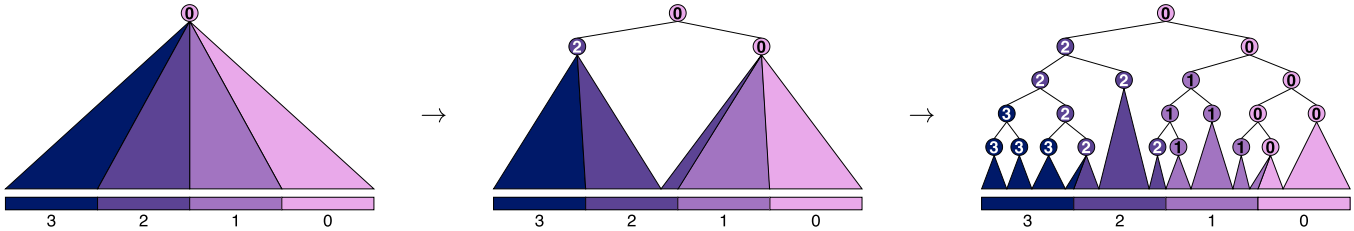


Fig. 5. Overview of deterministic task mapping in (single-level) ADWS with four workers. The nodes represent tasks, and the numbers on the tasks indicate which workers will execute them. The triangles below the nodes represent the *distribution ranges* over the workers to which the tasks' descendants will be distributed.

pushing the continuation of the current task into the local task queue (called the *work-first* policy [29]). Alternatively, it may continue to execute the current task by pushing the new task into the queue (called the *help-first* policy [30])[1]. When a worker has nothing to do (typically when an executing task is completed), it first tries to pop a ready task from the local task queue. If the local task queue is empty, it tries to steal a ready task from another worker's task queue. The victim for work stealing is chosen uniformly at random, an approach that has been shown to have good asymptotic bounds for space, execution time, and communication [5].

A thief steals a task from the other end of the queue for local push/pop operations, so that it steals the oldest task in the queue. This stealing strategy enables workers to steal work at a large granularity and thus exploits hierarchical data locality for private caches [6]. However, for deeper and more complicated hierarchical caches, this strategy is suboptimal because a set of tasks sharing data locality may be executed by workers that are distant in the cache hierarchy, which results in many cache misses. In addition, work stealing does not respect iterative data locality because of its randomness [6]: cached data for previous iterations are unlikely to be reused for successive iterations. Other schedulers that have been proposed for hierarchical caches are explained in detail in Section 7.

3 Almost Deterministic Work Stealing

This section introduces *almost deterministic work stealing (ADWS)*, a task scheduler that is designed to address both hierarchical and iterative data locality on arbitrary cache hierarchies. This section is based on the content of our previously published paper [14], but it provides a more refined explanation. The scheduling in ADWS is based on deterministic task mapping but also performs dynamic load balancing to fix load imbalances, which is why we describe it as *almost* deterministic.

3.1 Deterministic Task Mapping

ADWS first performs locality-aware task mapping by using user-provided hints on the relative amounts of work for each task. Fig. 2b shows the extension of the task group constructs for ADWS, which includes a few additional hints. The required hints are ratios of work with respect to the entire task group: `w_all` is for the total work for all tasks in the task group, and `w1`, `w2`, `w3`, and `w4` are for the work for each task (the `size` parameter is used only for *multi-level scheduling* introduced later in Section 4). The ratios do not have to be absolute values as long as the ratios of `w1`, ..., `w4` to `w_all` are all correct.

Fig. 5 illustrates the deterministic task mapping in ADWS. We assume that workers that are close in the hardware topology are numbered adjacently. The task tree is vertically divided so that workers that are close work on tasks that are close in the tree, thus respecting hierarchical data locality. To exploit iterative data locality, the task mapping is generated deterministically by recursively dividing *distribution ranges* according to the ratios of work in each task group. A distribution range is represented as a range of workers $[x, y)$, where x and y are real numbers ($x \leq y$), and it is illustrated as a triangle below a node (task) in the figure. First, the root task has a distribution range $[0.0, P)$, where P is the number of workers, which means that the descendants of the root task should be uniformly distributed among all workers. When a new task group is created, the current task's distribution range is divided according to the relative work for each child task, and each child task is assigned the corresponding subdivided distribution range. The endpoint values of the distribution ranges can be real numbers rather than just integers; that is, a boundary between distribution ranges can be in the middle of a worker. The basic rule of scheduling is that a task with range $[x, y)$ is assigned to worker $\lfloor x \rfloor$ (where $\lfloor \ldots \rfloor$ denotes the floor function), i.e., the rightmost worker in the range shown in the figure. The numbers on the nodes in the figure represent the assigned workers.

To explain the detailed algorithm of ADWS, we first classify tasks into two types. A task with range $[x, y)$ is classified as a *cross-worker task* if $\lfloor x \rfloor \neq \lfloor y \rfloor$ and as a *non-cross-worker task* if $\lfloor x \rfloor = \lfloor y \rfloor$. Cross-worker tasks have higher priority than non-cross-worker tasks, because their descendants should be distributed to multiple workers as soon as possible. Task groups inherit the same distribution range from the task that creates them, and we classify task groups into cross-worker or non-cross-worker task groups, as well. Next, suppose that worker i creates a cross-worker task group; then, we further consider three kinds of tasks within the task group, as illustrated in Fig. 6. The basic rules to schedule these three kinds of tasks are as follows:

1) Tasks with range $[x, y)$ such that $\lfloor x \rfloor > i$ are passed to worker $\lfloor x \rfloor$. They can be cross-worker or non-cross-worker tasks.

1. The work-first policy is also called the *child-first* policy or *continuation stealing*, and the help-first policy is also called the *parent-first* policy or *child stealing*.

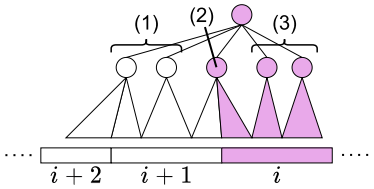


Fig. 6. Classification of tasks in a cross-worker task group. (1) Tasks passed to another worker (which can be either cross-worker or non-cross-worker tasks). (2) Cross-worker task assigned to local worker i. (3) Non-cross-worker tasks assigned to local worker i.

2) A task with range $[x, y)$ such that $\lfloor x \rfloor = i$ and $\lfloor y \rfloor > i$ is immediately executed by worker i. It is a cross-worker task assigned to worker i, and it is guaranteed to be the only one for each cross-worker task group.
3) Tasks with range $[x, y)$ such that $\lfloor x \rfloor = \lfloor y \rfloor = i$ are executed later by worker i. They are non-cross-worker tasks assigned to worker i.

Finally, we explain the algorithm for deterministic task mapping, which is shown in Fig. 7. The INIT, RUN, and WAIT functions correspond to those in the task group notation shown in Fig. 2. When a task group is initialized, it is assigned the total work for the group (according to a programmer's hint) and the distribution range of the executing task (line 16–18). The child tasks in the group are created from left to right as shown in Fig. 6, and their distribution ranges are divided and assigned at line 21–22. The worker first migrates tasks of type (1) to the appropriate workers at line 26. When the worker finds a task of type (2), it immediately executes it at line 26 and pushes the continuation of the current task (T_{cur}) into the task queue. The continuation left in the queue only contains tasks of type (3), which will be executed later by the worker at line 26 in the work-first manner.

After all the tasks of a task group are completed, the continuation of the parent task (T_{cur}) is returned to its owner (line 31). This is because of the rule that cross-worker tasks must be executed by their owner (i.e., worker $\lfloor x \rfloor$ for cross-worker tasks with range $[x, y)$). A cross-worker task returned to the owner is immediately executed by the owner when it reaches a scheduling point (e.g., spawn, task completion, or possibly preemption [31]), as it has the highest priority. Cross-worker tasks can be executed without delay because at most one cross-worker task is executable by the same worker simultaneously. The reason is that a cross-worker task group can have only one child cross-worker task that is owned by the same worker (i.e., the task of type (2) in Fig. 6), and the parent cannot proceed until the child has finished.

Following the algorithm for deterministic task mapping, each worker executes its assigned tasks from left to right in the figures. Fig. 8 illustrates the execution order for worker i. First, worker i preferably executes cross-worker tasks, which are the leftmost tasks in the figure; then, it executes the non-cross-worker tasks that it created and those migrated from other workers. In the implementation, each worker's local task queue is split into *primary queues* and *migration queues*, which are further divided by task depth. This separation is needed for the dynamic load balancing explained later in Section 3.2. Basically, a worker's primary queues are used for storing non-cross-worker tasks that it creates[2], while its migration queues are for passing non-cross-worker tasks across workers. The primary queues follow last-in first-out (LIFO) order for ordinary push/pop operations, while the migration queues follow first-in first-out (FIFO) order to pop tasks that were migrated by other workers. As the primary queues are checked before the migration queues in popping a local task, the tasks in Fig. 8 are executed from left to right, as the tasks $T_0, \ldots, T_7$ are numbered. This is almost the same order as serial execution, which is assumed to be highly tuned by programmers to achieve good cache performance [7]. Hence, ADWS is considered to be private-cache-friendly as well as work stealing [6].

However, deterministic task mapping alone cannot always provide a good schedule for tasks. That is, it can distribute tasks so that each worker's work is well balanced, but it does not guarantee that workers can proceed without any stalls. Fig. 9 shows an example of execution stalls caused by task dependencies. Following the left-to-right execution policy, worker $i+1$ executes task 4 after tasks 1, 2, and 3, but worker i cannot proceed to the next task group until task 4 is completed. Thus, worker i is stalled until worker $i+1$ executes task 4, and subsequently, worker $i+1$ is stalled by the delayed tasks assigned to worker i. This situation happens when task synchronization is frequent, i.e., when tasks create multiple task groups sequentially. Generally, it is hard to provide a good schedule that does not cause any stalls due to task dependencies, because the task graph is not known ahead of time: only the relative amount of work for the tasks in each task group is known. Our approach to solve this problem is to combine deterministic task mapping with the dynamic load balancing technique explained in the next section.

3.2 Dynamic Load Balancing

Dynamic load balancing in ADWS resolves both the execution stall problem explained above and load imbalances that dynamically appear. A load imbalance can result from the fluctuation of execution times (e.g., OS noise, frequency scaling) and from imprecise work hints provided by programmers, as it is often hard (or even impossible) for them to give a precise estimate for work. A naive approach to fix such execution stalls and load imbalances would be to perform random work stealing when local tasks are exhausted, but this could largely increase cache misses on hierarchical caches by moving data across distant caches; moreover, a steal can incur further steals, thus collapsing the task structure made by deterministic task mapping.

Instead, our approach is to localize the range of workers for choosing a victim by using the task structure generated by deterministic task mapping. The range of workers for work stealing (called the *steal range*) changes during execution, depending on the completion status of cross-worker task groups. Roughly speaking, the steal range is restricted to a set of workers that are working on the same unfinished

2. Note that T_3 and T_4 in Fig. 8 are in fact the continuations of T_1 and T_0 respectively, but they are abstracted as non-cross-worker tasks, because a continuation only spawns non-cross-worker tasks for the current task group.

```
14 TG ← Task group of interest
15 T_cur ← Current task (that creates TG)
16 Function TaskGroup::Init(work)
17 |  TG.work ← work
18 |  TG.range ← T_cur.range
19 Function TaskGroup::Run(T_new, work)
20 |  Add T_new to TG.tasks
21 |  TG.work ← TG.work − work
22 |  (T_new.range, TG.range) ← divide TG.range according
   |   to the ratio work : TG.work
23 |  if T_cur is a cross-worker task and T_new belongs to
   |   another worker then
24 |  |  Migrate T_new to worker ⌊T_new.range.x⌋
25 |  else
26 |  |  Execute T_new and push the continuation of current
   |  |   task T_cur to a local task queue
27 Function TaskGroup::Wait()
28 |  foreach T ∈ TG.tasks do
29 |  |  Wait for completion of T
30 |  if T_cur is a cross-worker task then
31 |  |  Migrate T_cur to worker ⌊T_cur.range.x⌋
```

Fig. 7. Algorithm for deterministic task mapping in ADWS.

cross-worker task group. For example, in Fig. 9, by restricting the steal range of worker i and $i+1$ to within the task group of interest, worker i and $i+1$ will steal task 4 and 6 respectively, without stealing tasks outside this task group.

Specifically, to determine the steal range for each worker, we use the tree of cross-worker task groups that is created during deterministic task mapping. When a new cross-worker task group is created, we assign to it a pointer to the parent task group. Fig. 10 illustrates the tree of cross-worker task groups, by omitting the non-cross-worker groups from Fig. 5. Here, the tree nodes represent task groups rather than tasks. The solid arrows point to the task groups having the current steal ranges for each worker. They are chosen from among the *dominant* task groups, which have triangles below the nodes. A task group is dominant if it is a cross-worker task group and at least one of its child cross-worker tasks has already been completed. A worker i is *dominated* by a dominant task group with the distribution range $[x, y)$ if $\lfloor x \rfloor \leq i < \lfloor y \rfloor$ (worker $\lfloor y \rfloor$ is not dominated). The current steal range of worker i is determined by the topmost dominant task group that dominates worker i.

At an early stage of execution, every worker has its own dominant task group that dominates only itself (Fig. 10a), assuming that enough parallelism is provided. As tasks are completed, the task groups that are close to the root become dominant. In Fig. 10b, worker 1 still has its own dominant task group, but its ancestor (with range $[0, 2.x)$) already dominates it; therefore, the current steal range of worker 1 is determined to be that of the ancestor. Every time a worker tries to steal a task, it first checks whether any ancestor has become dominant before performing a steal. This can be done by traversing the cross-worker task groups from the bottom up in the tree until reaching the root, which takes time proportional to the tree depth at most. This cost will be negligible in most cases, because it is incurred only when work stealing is performed (cf. the work-first principle [29]). Eventually, the root task group becomes dominant (Fig. 10c), which is equivalent to traditional random work stealing that targets all workers.

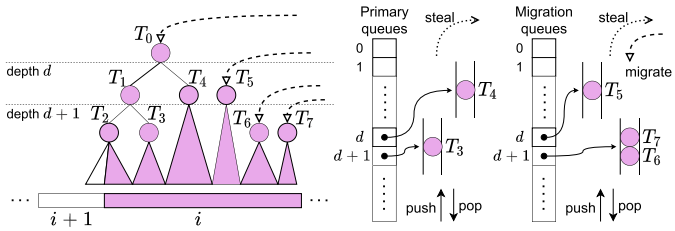


Fig. 8. Illustration of the task execution order for worker i in ADWS, with the tasks numbered in that order. T_0, T_1, and T_2 are cross-worker tasks assigned to worker i. Non-cross-worker tasks generated by worker i (T_3 and T_4) are pushed into the local primary queues, and those from other workers (T_5, T_6, and T_7) are migrated to the migration queues. The primary and migration queues are separated by task depth.

Once the current steal range of worker i is determined as $[x, y)$ ($\lfloor x \rfloor \leq i < \lfloor y \rfloor$), it randomly chooses worker j from $\lfloor x \rfloor \leq j \leq \lfloor y \rfloor$ ($j \neq i$) and tries stealing work from worker j. To steal the rightmost task in figures, worker i first tries to steal a task from the migration queues of worker j, and if it fails, it tries to steal from the primary queues. As an exception, worker i should not steal tasks from the migration queues of worker $\lfloor x \rfloor$ and from the primary queues of worker $\lfloor y \rfloor$. This rule is to avoid stealing tasks outside the topmost dominant task group. Tasks should not be stolen from the migration queues of worker $\lfloor x \rfloor$, because it stores tasks that were migrated from outside the steal range. Similarly, tasks in the primary queues of worker $\lfloor y \rfloor$ are out of the range $[x, y)$. This is why we split the local task queue into primary queues and migration queues. Note that, to maintain this separation, the descendants of tasks that are migrated to migration queues are pushed into the migration queues unless stolen, and the same applies to primary queues.

Furthermore, to avoid stealing tasks from outside the topmost dominant task group, we also manage *task depths* and separate primary and migration queues by task depth. A task depth is used to distinguish between tasks from different cross-worker task groups, and a task depth is defined as the depth of cross-worker task groups in the task hierarchy (the root depth is 0). Hence, when a non-cross-worker task group is created, the task depth is not incremented. Tasks inherit the depth of the parent task group, and tasks at depth d are pushed into the primary or migration queue at depth d, as illustrated in Fig. 8. Because the descendants

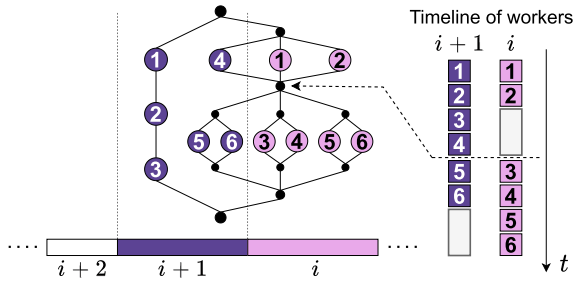


Fig. 9. Example of execution stalls in the deterministic task mapping of ADWS. The numbers in the nodes (tasks) represent the execution order for each worker, and each one has the same amount of work. The timelines show the timing of when the tasks are executed by each worker. This example shows that even if the same amount of work is allocated to each worker, execution stalls can happen because of task dependencies.

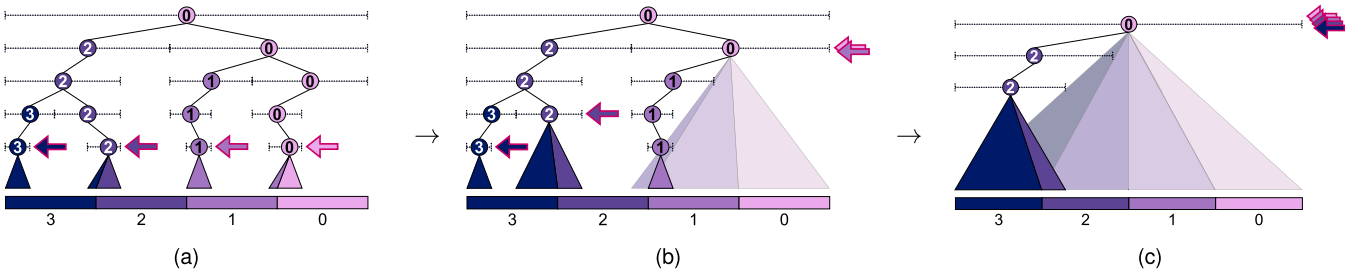


Fig. 10. Tree of cross-worker task groups that is used for dynamic load balancing in ADWS. The nodes represent cross-worker task groups, with non-cross-worker task groups omitted from Fig. 5. Dominant task groups are represented as nodes with triangles below. The solid arrows point to the topmost dominant task group, which corresponds to the steal range for each worker.

of a non-cross-worker task have the same depth, they are pushed into the same queue unless stolen. For example, the descendants of T_4 in Fig. 8 are pushed into the same primary queue at depth d. So as not to steal tasks from outside a task group at depth d, we have to steal from queues whose depth is greater than or equal to d within the steal range. If a task is successfully stolen, the descendants of the task are executed by the work-first policy and managed in the primary queue of the thief at the same depth.

As a summary of these scheduling rules in ADWS, Fig. 11 shows the algorithm to try to get a runnable task when a worker has no work to do. It first checks the local primary queues from the bottom up (line 33–35) and then checks the local migration queues from the top down (line 36–38), so that tasks are executed from left to right in Fig. 8. If no local task is found, it then tries to steal work from the current steal range. When a worker is not dominated by any task group, it does not perform work stealing (line 40), which prevents tasks migrated by deterministic task allocation from being stolen too soon. When stealing a task, a worker first checks the migration queues from the bottom up (line 44–46) and then checks the primary queues from the top down (line 48–50), which is the reverse order of local pop operations[3], so that the rightmost tasks are preferably stolen. Note that, in an actual implementation, we can reduce the number of queue checks by remembering which queues are empty. The GetRunnableTask function is repeatedly called in a worker's scheduler loop (spin loop) until it succeeds.

3.3 Issue of Shared Cache Utilization

The deterministic task mapping straightforwardly matches the task hierarchy to the cache hierarchy so that adjacent workers that are close in the hardware topology can exploit hierarchical data locality; however, there is still room for ADWS to further exploit hierarchical data locality on shared caches. Let us consider a case in which a task group is mapped to workers sharing a cache, and the descendants of the task group access the same working set of data many times. Even though the descendants are executed on the same cache, if the working set size is larger than the capacity of the shared cache, parallel execution of them can cause many capacity misses. For a shared cache, it is often preferable to execute on a small working set at the same time (cf. cache blocking in general), so that the working set size does not exceed the cache capacity. On the other hand, the deterministic task mapping in ADWS divides the whole computation graph vertically and assigns tasks to workers in one go, so that the data sharing among workers becomes as minimal as possible.

For the example of decision tree construction in Section 2.1, suppose that the overall data (*rows* in Fig. 1) do not fit into a shared cache, but the partitioned data for each subtree (*rowsL* and *rowsR*) do fit into the shared cache. In this case, parallel execution of the two tasks for constructing subtrees would cause many more capacity misses than one-by-one execution of them. Thus, if we execute one of the two subtasks after the other one is completed, we can promote data reuse within each task on the shared cache, which would naturally occur in serial execution.

4 Multi-Level Scheduling

To address the issue of shared cache utilization in ADWS, we introduce *multi-level scheduling*, a generic scheduling framework for promoting data reuse on shared caches. In this section, before we introduce *multi-level ADWS* in Section 5 by combining multi-level scheduling with ADWS, we explain the design of multi-level scheduling, which is orthogonal to that of ADWS. We borrow the idea of *two-level scheduling* [12], [13], [15], [16], [17], which is designed for specific hardware configurations and specific scheduling

```
32 Function GetRunnableTask()
33     for d ← d_max to 0 do                    // check local primary queues
34         T ← Pop from PrimaryQueues[d] of myself
35         if task T is popped then return T
36     for d ← 0 to d_max do                    // check local migration queues
37         T ← Pop from MigrationQueues[d] of myself
38         if task T is popped then return T
       // Begin work stealing
39     TG ← find the topmost dominant task group
40     if TG does not exist then return NULL
41     [x, y) ← TG.range (i.e., current steal range)
42     victim ← random choice from ⌊x⌋, ..., ⌊y⌋ except myself
43     if victim ≠ ⌊x⌋ then                     // steal from migration queues
44         for d ← d_max to TG.depth do
45             T ← Steal from MigrationQueues[d] of victim
46             if task T is stolen then return T
47     if victim ≠ ⌊y⌋ then                     // steal from primary queues
48         for d ← TG.depth to d_max do
49             T ← Steal from PrimaryQueues[d] of victim
50             if task T is stolen then return T
51     return NULL
```

Fig. 11. Algorithm for getting a runnable task in ADWS. This function is repeatedly called in a scheduler loop until a task is successfully popped.

3. Steal and migration operations take place on a different side of each queue than local push/pop operations, as illustrated in Fig. 8.

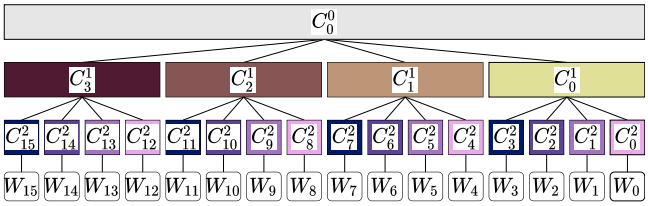


Fig. 12. Example of a tree of caches with 16 cores. A node denoted by C_i^l is the level-l cache with index i, and one denoted by W_k is a worker that is pinned to the core with private cache C_k^2.

```
52 Function ExecuteTaskGroup(TG)
53     W ← Worker of interest
54     C ← W.cache  // Cache for which W is the current leader
55     if C is not a leaf and Size(TG) ≤ Capacity(C) then
56         W.cache ← A child of C to which W belongs
57         Schedule TG among the children of C and wait
58         W'.cache ← C
59     else
60         Schedule TG among the siblings of C and wait
```

Fig. 13. Algorithm for execution of a task group in multi-level scheduling.

strategies at each cache level, and we generalize it to a general scheduling framework for arbitrary cache hierarchies and arbitrary scheduling policies for each cache level.

The core idea of multi-level scheduling is to limit the working set size of tasks that are simultaneously scheduled on a shared cache so that it does not exceed the shared cache capacity. Here, we require another hint provided by programmers; the working set sizes of task groups must be known for performing multi-level scheduling. In multi-level scheduling, when a task group that fits into a shared cache is generated, it is assigned to the shared cache and executed only by the workers sharing the cache. This assignment is hierarchically applied to hierarchical caches with any number of cache levels. Although it may decrease the amount of parallelism available to workers, it has the benefit of increasing the shared cache utilization for memory-bound programs in which each task group intensively reuses data among its descendants.

4.1 Preliminaries

To explain the design of multi-level scheduling more specifically, we first model a cache hierarchy as a tree of caches [32]. We assume that leaf nodes are private caches (e.g., L1 caches) and that the root node is the main memory, which is of infinite size. Fig. 12 shows an example of such a tree of caches. A cache C_i^l is identified by its cache level l and an index i among the level-l caches. The root node (main memory) is denoted by C_0^0, given that the root cache level is 0. Note that the cache levels are numbered here in the reverse order of the prevalent naming convention for L1, L2, and L3 caches. A worker W_k is pinned to the core with private cache C_k^2.

For multi-level scheduling, the working set sizes for task groups must be known in advance. Existing studies have also used hints on working set sizes to improve the scheduling of nested parallel programs. For example, Chen et al. [12], [13], [15] proposed an online profiling method to estimate the working set sizes for iterative computations, and space-bounded schedulers [4], [9], [10] use programmer-provided hints on the working set sizes for each task. In this paper, we adopt the latter approach for simplicity. Fig. 2b shows the programming style of multi-level ADWS (which will be introduced in Section 5), which requires a working set size parameter (`size`) for each task group[4], in addition to the work hints for each task required by ADWS. Note that multi-level scheduling itself does not require the work hints. The working set size of a task group TG that is specified by a programmer is denoted by $Size(TG)$, and the capacity of a cache C is denoted by $Capacity(C)$.

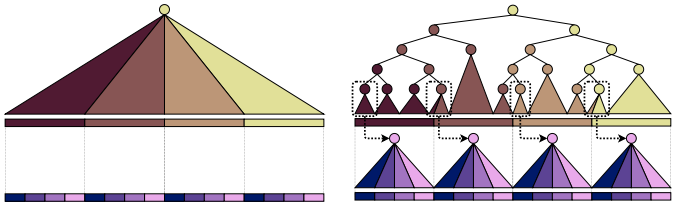
4. The previous approaches [4], [9], [10], [12], [13], [15] assigned the working set sizes to tasks, but this paper assigns them to task groups instead.

4.2 Design and Algorithm

In multi-level scheduling, tasks are cascaded from the top down in the tree of caches. To explain the design and algorithm of multi-level scheduling, we first focus on the scheduling between a level-l cache and its child caches at level $l+1$. If the working set size of a task group is less than or equal to the cache capacity, then the task group can be *tied* to the cache. Once a task group is tied to a cache C, the task group's descendants are all executed by the workers sharing C. At most one task group is tied to each cache simultaneously. While a task group is tied to cache C_i^l at cache level l, the task group's descendants are scheduled by the children of C_i^l at level $l+1$. If a child cache C_j^{l+1} finds a task group TG for which $Size(TG) \leq Capacity(C_j^{l+1})$, then TG is tied to C_j^{l+1} and scheduled recursively by the children of C_j^{l+1}.

We can choose any strategy to schedule tasks among the children of C_i^l. This is simplified by introducing cache *leaders* and considering the scheduling among the cache leaders at each cache level. Specifically, at most one of the workers sharing a cache acts as its leader. At the beginning of execution, leaders are elected from the bottom up in the cache tree; that is, the leader of a level-l cache is chosen from the leaders of its children at level $l+1$. If a worker becomes the leader at cache level l, it stops being the leader at level $l+1$. Note that cache leaders can change during execution and can be absent, and a worker can only be the leader of at most one cache at a time. After the leader of the root cache C_0^0 is chosen, it starts to run the root task (or the main function). If a worker W is the current leader of cache C_i^l when it encounters a task group TG such that $Size(TG) \leq Capacity(C_i^l)$, then TG is tied to C_i^l. At the same time, W stops being the leader of C_i^l and becomes the leader of the next-level $(l+1)$ cache to which W belongs. Then, the descendants of TG are scheduled by the leaders of the children of C_i^l. When TG is completed, the leader of C_i^l is elected again from the leaders of the children of C_i^l.

Fig. 13 shows the algorithm for executing a task group TG. The function ExecuteTaskGroup is a wrapper for each task group's execution (i.e., child tasks of TG are spawned and awaited at line 57 or line 60), which is independent of the scheduling strategies for each cache level. All that must be done is to determine whether to tie TG to the child cache. The cache for which worker W is currently the leader is retrieved by $W.cache$ (line 54), and we denote it as C. If TG is tied to C, then W stops being the leader of C and moves on to the next cache level by becoming the leader of the



(a) ADWS at cache level 1 (b) ADWS at cache level 2

Fig. 14. Illustration of multi-level ADWS. In each case, the upper part depicts level-1 caches, and the lower part depicts level-2 caches (corresponding to Fig. 12). Level-1 leaves (enclosed by dotted lines) are scheduled by ADWS among the level-1 caches, and each leaf is passed to a level-2 cache. Its descendants are then scheduled recursively by ADWS among the level-2 caches.

```
61 Function ExecuteTaskGroup(TG)
62     l ← Current cache level
63     (i, j) ← Integer parts of the current distribution range (i ≤ j)
64     l_next ← l,  cap ← Σ_{k=i}^{max(i,j-1)} Capacity(C_k^l)
65     while l_next < l_max and Size(TG) ≤ cap do
66         l_next ← l_next + 1
67         cap ← Σ_{k=i}^{max(i,j-1)} Σ_{c∈D(C_k^l, l_next)} Capacity(c)
68     if l ≠ l_next then
69         Apply single-level ADWS to ∪_{k=i}^{max(i,j-1)} D(C_k^l, l_next)
70     else
71         Continue to schedule TG at current cache level l
```

Fig. 15. Algorithm for cache hierarchy flattening in multi-level ADWS.

child cache to which W belongs (line 56). TG is then scheduled among the leaders of the children of C (line 57). After TG is completed, the worker executing the continuation of TG (which may be different from the previous leader W) becomes the leader of the parent cache C (line 58). On the other hand, if TG is not tied to C, then W continues to schedule TG among the leaders of the siblings of C at the current cache level, and it remains the leader of C (line 60).

A task group tied to C_i^l is called a *level-l leaf*. A level-l leaf fits into C_i^l, but its parent does not, assuming that C_i^l has the same capacity for all i. In overviewing the scheduling at cache level l, level-l leaves appear to be sequential tasks that are scheduled by the sibling caches at level l. In fact, each level-l leaf is scheduled by the workers sharing cache C_i^l in parallel, but it appears to be executed sequentially by C_i^l, as at most one level-l leaf is tied to a cache until it is completed. Hence, the scheduling at cache level l can be abstracted as the scheduling for level-l leaves among the leaders of the level-l caches. In addition, as at most one level-l leaf is tied to a cache at a time, we have only one root task for each level of scheduling, which is the same situation as in ordinary (single-level) schedulers for nested parallel computations. Because of these characteristics, we can easily apply existing schedulers for nested parallelism, such as work stealing and ADWS, to each cache level in multi-level scheduling.

5 Multi-Level ADWS

By combining multi-level scheduling with ADWS, we introduce *multi-level ADWS*. For clarity, we refer to the original design of ADWS (introduced in Section 3) as *single-level ADWS*. Multi-level ADWS is a multi-level scheduler whose scheduling policy at each cache level is single-level ADWS. The idea of multi-level ADWS is illustrated in Fig. 14, with a cache hierarchy that corresponds to Fig. 12. When a level-l cache encounters a level-l leaf while level-l leaves are being scheduled by single-level ADWS among level-l caches, the level-l leaf is passed to the next cache level $l+1$, and single-level ADWS is applied recursively at level $l+1$.

A downside of multi-level scheduling is that even for computations that are small enough to fit into entire caches, they are scheduled in a multi-level way, thus causing load imbalances. For example, if the overall working set of the root task group fits into cache C_i^l, then only the workers sharing C_i^l would work on it, which leaves workers outside C_i^l idle. Hence, we introduce a technique called *cache hierarchy flattening*, in which part of the cache hierarchy is logically flattened to single-level caches to enable the application of single-level ADWS over them. The rationale for this technique is that, when an overall working set fits into an entire hierarchical cache, single-level ADWS can efficiently exploit data locality, because the issue of shared cache utilization (Section 3.3) does not apply.

We first explain the operation of cache hierarchy flattening by using the example of a tree of caches in Fig. 12. For example, if the overall working set size of the root task group is less than or equal to the sum of the capacities of level-1 caches $C_0^1, \ldots, C_3^1$, then the whole computation should be scheduled by single-level ADWS over all workers; otherwise, the load will not be well balanced. Accordingly, we flatten all of the level-2 caches into single-level caches and apply single-level ADWS over them ($C_0^2, \ldots, C_{15}^2$). Moreover, cache hierarchy flattening can be applied to a sub-computation for part of the cache hierarchy. Suppose that C_2^1 is assigned a cross-worker task with range $[2.x, 4.0)$, which means that C_2^1 is responsible for distributing tasks to C_3^1 and to itself in ADWS at level 1. When C_2^1 encounters a task group TG such that $Size(TG) \leq Capacity(C_2^1) + Capacity(C_3^1)$, the task group should be scheduled by single-level ADWS over the children $C_8^2, \ldots, C_{15}^2$, rather than scheduling TG only within C_2^1.

Fig. 15 shows the algorithm for cache hierarchy flattening in a general case. If the distribution range of TG is $[x, y)$, then the integer parts of this range are $i = \lfloor x \rfloor$ and $j = \lfloor y \rfloor$ (line 63). First, we consider the total capacity of the caches within the current distribution range, $C_i^l, \ldots, C_{\max(i,j-1)}^l$ (line 64)[5]. If $Size(TG)$ is less than or equal to that total capacity, then the total capacity of the children of $C_i^l, \ldots, C_{\max(i,j-1)}^l$ is further checked (line 65–67). Here, $D(C, l)$ denotes the set of level-l caches that are descendants of cache C. This is repeated until either the total capacity of the descendants becomes smaller than $Size(TG)$ or the cache level reaches the maximum level l_{max}. Then, if the next cache level l_{next} is updated, we flatten the level-l_{next} caches under $C_i^l, \ldots, C_{\max(i,j-1)}^l$ and schedule TG over them by single-level ADWS (line 68–71). Otherwise, we continue to schedule TG at the current cache level without flattening the caches.

Note that, even if we apply cache hierarchy flattening to other scheduling strategies that are combined with multi-level scheduling (e.g., multi-level work stealing), the benefit is limited if the corresponding single-level scheduling is not

5. We do not include C_j^l $(j > i)$ in this range (indicated by the largest index, $\max(i, j-1)$) because C_j^l can also assign a level-l leaf for itself, which should be prioritized to cache hierarchy flattening by C_i^l.

TABLE 1
Experimental Environment

CPU model	Intel Xeon Platinum 8280 (Cascade Lake)
# of cores	56 (28 cores/socket × 2 sockets)
Frequency	2.7 GHz (Turbo 4.00 GHz)
L1 data cache	32 KB/core
L2 cache	1 MB/core
L3 cache	38.5 MB/socket
Compiler	GCC v7.5.0 (`-O3 -march=native`)
OS	CentOS Linux 7 (`3.10.0-957.el7.x86_64`)

designed for hierarchical caches. For example, as conventional random work stealing would cause many cache misses on hierarchical caches because of its randomness, cache hierarchy flattening for multi-level work stealing would also cause many cache misses. On the other hand, multi-level ADWS can benefit from cache hierarchy flattening because single-level ADWS is already cache-hierarchy-aware, to some extent, and it is therefore expected to perform well over a wide range of working set sizes.

6 Performance Evaluation

To evaluate the performance of ADWS, we conducted experiments with seven benchmarks. In short, our primary findings in this section are as follows:

- ADWS outperforms other existing schedulers over a wide range of working set sizes by improving cache utilization. Multi-level ADWS reduces shared cache misses and performs better than single-level ADWS in many cases, but it has drawbacks of increased core idle times and private cache misses (Section 6.3).
- ADWS is tolerant to load imbalances. Even when we do not provide any work hints to ADWS and the workload is highly unbalanced, it performs similarly or better than random work stealing (Section 6.4).
- ADWS can take advantage of the NUMA local allocation policy because of its deterministic scheduling. By allocating physical memory to proper NUMA nodes, ADWS can perform better than when the NUMA interleave policy is adopted (Section 6.5).

6.1 Experimental Settings

For evaluation, we used a two-socket Cascade Lake machine of the Oakbridge-CX supercomputer at The University of Tokyo. The machine's configuration is summarized in Table 1. Hyper-threading was disabled, while Turbo Boost and Transparent Huge Pages (THP) were enabled. The `-march=native` option was not set for compiling scheduler code because the resulting assembly included SIMD instructions, which slowed down the CPU frequency.

For evaluation, we implemented five schedulers: single-level WS (*SL-WS*)[6], single-level ADWS (*SL-ADWS*), multi-level WS (*ML-WS*), multi-level ADWS (*ML-ADWS*), and a space-bounded scheduler (*SB*). All of these schedulers were implemented on MassiveThreads [33], a lightweight user-level threading library. We ported the SB implementation published by Simhadri et al. [18], [19] to MassiveThreads. We adopted the "SB-D" variant with a minor modification to avoid lock contention, because in most cases it performed better than the "SB" variant in their paper, which used centralized task queues. For SB parameters, we set $\sigma = 0.5$ and $\mu = 0.2$, which was the same setting as in their evaluation. For SB only, we specified the working set sizes for tasks rather than task groups to follow their implementation. To validate the baseline performance of MassiveThreads, we also compared it with the widely used tasking runtime Cilk Plus [20] (the version shipped with GCC v7.5.0), because both MassiveThreads and Cilk Plus adopt work-first random work stealing by default; we thus expect that SL-WS and Cilk Plus would perform similarly.

In all evaluations, as many kernel-level threads (workers) as cores were created and pinned to cores by setting their affinity. Unless explicitly noted, we set the `-interleave=all` option to the `numactl` command when running the experiments, which mean that physical pages were almost evenly distributed to all NUMA nodes. Serial execution times were measured with the `-localalloc` option on a fixed core to avoid remote memory accesses. For each measurement, we repeated the computation 11 times within each program execution and omitted the first run (warm-up). We also repeated the execution five times from the outside of the program; thus, we plotted the mean (geometric or arithmetic mean depending on the context) of 50 runs in total for each point.

To break down the schedulers' execution times, we profiled three metrics: the *busy time*, *idle time*, and *overhead*. The *busy time* was the time during which a worker was executing a task, i.e., performing meaningful work in a benchmark program. The *idle time* was the time during which a worker was searching for ready tasks because no local task existed. The *overhead* was the remaining time besides the busy time and idle time, which was the overhead of the scheduler itself. Because profiling incurs additional overheads, we conducted profiling separately from the main evaluation. As we did not modify Cilk Plus, we could not obtain a profiling result for Cilk Plus.

We also profiled cache miss counts by using the performance monitoring units (PMUs) in the Intel processors. Instead of command-line `perf`, we directly called the `perf_event_open()` Linux system call from the benchmark programs to avoid counting unnecessary events (e.g., operations at initialization) and to profile each repetition separately. For private caches, we counted L2 misses by monitoring the `L2_RQSTS.MISS` event [34]. For shared caches, we counted L3 misses by monitoring the `LLC_LOOKUP.ANY` event[7] in uncore Caching/Home Agent (CHA) units [35]. All of these events count the total number of cache replacements, including data reads, writes, and prefetches.

6.2 Benchmarks

We used seven benchmarks to evaluate the performance of the schedulers. Each benchmark is explained in the following.

6. *WS* denotes conventional random work stealing with the work-first policy, which is the default scheduler of MassiveThreads.

7. With the register state of `Cn_MSR_PMON_BOX_FILTER0.state=0x1`.

Recursive Repeated Map (RRM): This benchmark is a simplified version of the artificial benchmark that was used in the evaluation of SB [18], [19]. It takes an array of double-precision floating-point numbers as input and logically divides the array recursively into two subarrays. At each recursion before dividing an array, a *map* function, which just multiplies each element by a constant and adds the product to itself, is applied to the array three times. Recursion stops when the array size becomes smaller than 32 KB. Each map function is also recursively parallelized until the array size becomes smaller than 128 KB. An array is divided in the ratio $1 : \alpha$ at each recursion, where α is a work ratio parameter, which determines how much the computation graph is unbalanced.

Quicksort: This benchmark implements the well-known divide-and-conquer Quicksort algorithm, by parallelizing two recursive calls for the partitioned arrays. The partition operation is also parallelized to increase the parallelism through double buffering; thus, the total working set size is double the input array size. As it involves parallel computations for an array before recursive parallel calls, the computation pattern is similar to that of RRM. The shape of the computation graph depends on the input and the pivot, which is chosen as the median of the first three elements in our evaluation. The cutoff sizes for both recursion and partitioning were 64 KB. Each element was a random double-precision floating-point number.

KDTree: This benchmark constructs a kd-tree for randomly generated three-dimensional points (double-precision). The algorithm is similar to Quicksort and the computation graph is also irregular. The pivot value is chosen as the median of the first three values along an axis (x, y, or z) that is chosen in a round-robin way at each recursion level. The algorithm keeps creating tree nodes until the array size becomes smaller than 4 KB. The cutoff sizes for both recursion and partitioning were set to 64 KB. Compared to Quicksort, KDTree is more memory-bound because the recursion in KDTree stops earlier than in Quicksort, which reduces the amount of computation per memory access.

Decision Tree: This benchmark is the same as that explained in Section 2.1. We used the HIGGS dataset downloaded from the UCI machine learning repository [36], which was published by Baldi et al. [37]. The task is binary classification using 28 attributes of double-precision floating-point numbers for each row. 500,000 out of 11,000,000 rows were used for testing data, and the other rows were used for training data, which had a size of about 2 GB. The performance evaluation accounted only for the training time. For validation, we determined that our implementation achieved an accuracy of 72% for the testing data, while the accuracy with random prediction was 52%. The maximum depth for a decision tree was set to 17, which achieved the best accuracy when using all the training data. The cutoff size for recursion was 64 KB, and that for parallel loops and partitioning was 256 KB. The computation graph can be irregular, but note again that Fig. 4 shows a well-balanced graph for simplicity.

Matrix Multiplication (MatMul): This benchmark performs multiplication of single-precision floating-point dense matrices (SGEMM). The algorithm is based on cache-oblivious matrix multiplication for square matrices [38], which recursively divides a square matrix into four submatrices. The cutoff size for recursion was 64×64, and the computation kernel was optimized by hand for 64×64 matrices with AVX-512 instructions. We added a padding of 128 bytes to each row to avoid cache conflict with matrices whose size was a power of two, because this padding size showed the best performance.

Heat2D: This benchmark is a simple five-point stencil computation with double buffering, which is parallelized by recursively dividing a square grid into four equally sized subgrids. Heat2D has a high degree of iterative data locality because the same computation is repeated across iterations, whereas it does not have much hierarchical data locality because tasks do not share much data with others within each iteration. Thus, we cannot expect much cache reuse when the overall working set does not fit into the whole caches. The cutoff size for recursion was 64×64. The execution time for 50 iterations was used for performance evaluation. We added a padding of 256 bytes to avoid cache conflict with matrices whose size was a power of two, because this padding size showed the best performance.

SPH: This benchmark is a 3D dam-breaking simulation using the smoothed particle hydrodynamics (SPH) method, which calculates short-range particle interactions within an effective radius. The SPH computation kernel refers to the method by Becker and Teschner [39]. The implementation was ported from FDPS [40], which is based on an octree [41] to partition a 3D space and efficiently find the neighbors of particles. The number of particles in an octree node was given as roughly estimated work hints for ADWS, and the working set size hints were calculated based on the number of particles. The performance measurement included only the force calculation part, which is straightforwardly parallelized by traversing the octree, whereas other parts such as tree building were excluded. Each leaf of an octree had at most 32 particles, and the results were obtained as the execution times for five iterations.

6.3 Overall Performance Results

Fig. 16 shows the speedup on 56 cores compared with the serial execution times (except for MatMul) for different working set sizes. For MatMul, we plotted the FLOPS instead of the speedup; the machine's peak performance was 8.6 TFLOPS (the CPU frequency while executing AVX-512 instructions was 2.4 GHz). The work ratio parameter α of RRM was set to 1.0, which generates a well-balanced computation graph. Fig. 17 shows a breakdown of the execution times, and Fig. 18 shows the cache miss counts. For each benchmark, the largest problem size, corresponding to the rightmost point in Fig. 16, was used for the performance analysis results shown in Figs. 17 and 18.

First, the performance of SL-WS and Cilk Plus was similar in most cases, which indicates that the baseline performance of MassiveThreads was comparable to that of Cilk Plus. This observation is also supported by Fig. 17, in which the overhead of MassiveThreads is so small that only the difference in scheduling decisions (rather than the overheads) affected the overall performance.

For small working set sizes less than the total L3 cache size, ADWS outperformed the other schedulers on all of the

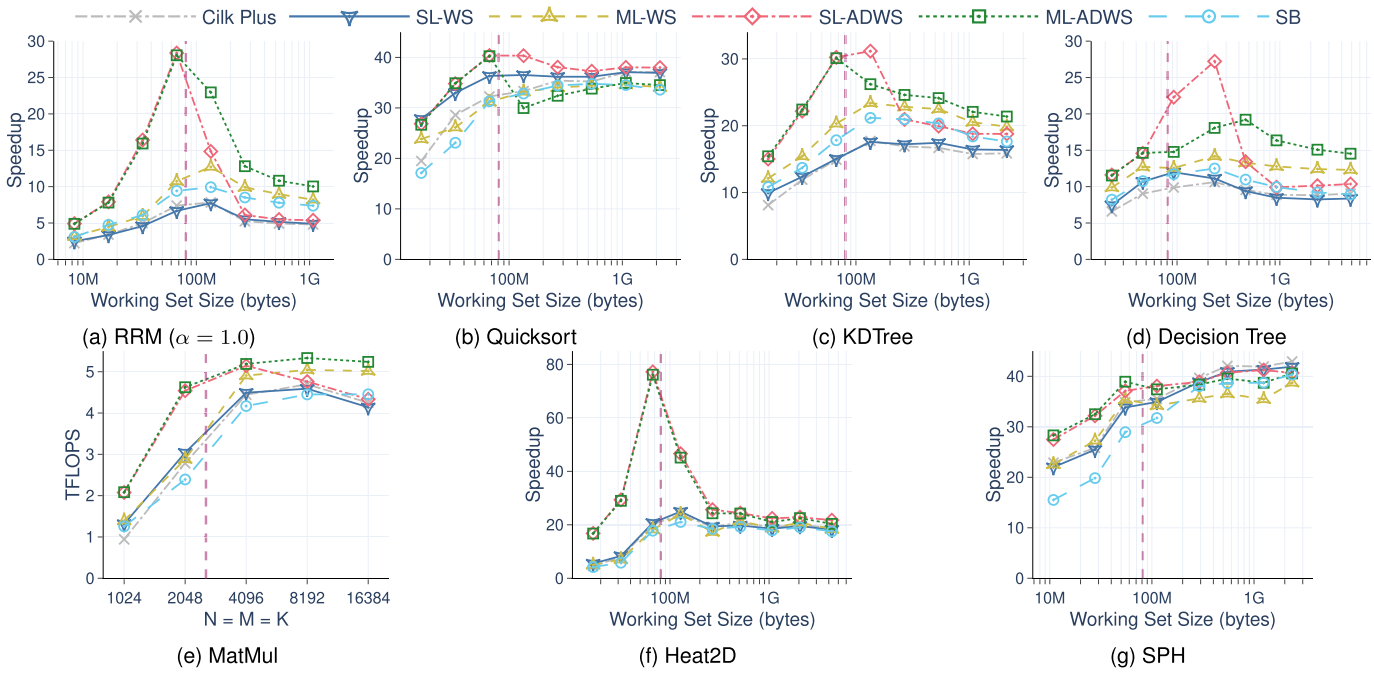


Fig. 16. Speedup (or FLOPS) on 56 cores with various working set sizes. Each vertical dashed line represents the total size of the system's L3 caches ($38.5\,\mathrm{MB/socket} \times 2\,\mathrm{sockets} = 77\,\mathrm{MB}$).

benchmarks. In this range of working set sizes, ML-ADWS performed almost the same as SL-ADWS because of the cache hierarchy flattening. The performance improvement of ADWS was outstanding for Heat2D, which showed speedup by almost 80x at most by fully exploiting iterative data locality through the deterministic scheduling. Such super linear speedup was possible because parallel execution can use multiple L3 caches, whereas serial execution can use only one L3 cache. The other schedulers based on randomness for scheduling underperformed ADWS because they could not exploit iterative data locality.

For large working set sizes in the RRM, KDTree, Decision Tree, and MatMul benchmarks, the multi-level schedulers outperformed their single-level counterparts. Notably, ML-ADWS showed improvements over SL-ADWS of 21% on MatMul with $N = 16384$ and 40% on Decision Tree with full training data. Compared with Cilk Plus, ML-ADWS achieved 61% performance improvement on Decision Tree with full training data. By examining Fig. 17, we can see that ML-ADWS greatly reduced the busy times for these benchmarks, which indicates that the cache utilization was improved. This claim is also supported by the L3 cache miss counts shown in Fig. 18: the multi-level schedulers incurred many fewer L3 misses than the single-level schedulers. Moreover, the numbers of L3 misses for the multi-level schedulers were almost the same as in serial execution, which indicates that these schedulers were nearly optimal in terms of L3 cache misses. Performance improvements of these benchmarks were due to better exploitation of hierarchical data locality by ML-ADWS; in contrast, cache misses in Heat2D were not reduced because Heat2D has little hierarchical data locality.

For Quicksort, however, the multi-level schedulers performed worse than the single-level schedulers. The reason is clearly apparent in Fig. 17: the idle times of the multi-level schedulers were much longer than those of the single-level schedulers. Fig. 18 shows that L3 misses were reduced with the multi-level schedulers, but the effect of the idle times was larger, which resulted in worse overall performance. Accordingly, the characteristics of the benchmarks determined whether single- or multi-level scheduling was better. We can also see increases in the idle times of the multi-level schedulers on KDTree and Decision Tree, but the benefit of improved cache utilization overwhelmed the side effect of increased idle times, thus resulting in better overall performance. The long idle times of the multi-level schedulers were due to their design, in which only one level-l leaf at a time could be tied to a cache C_i^l. In particular, in irregular

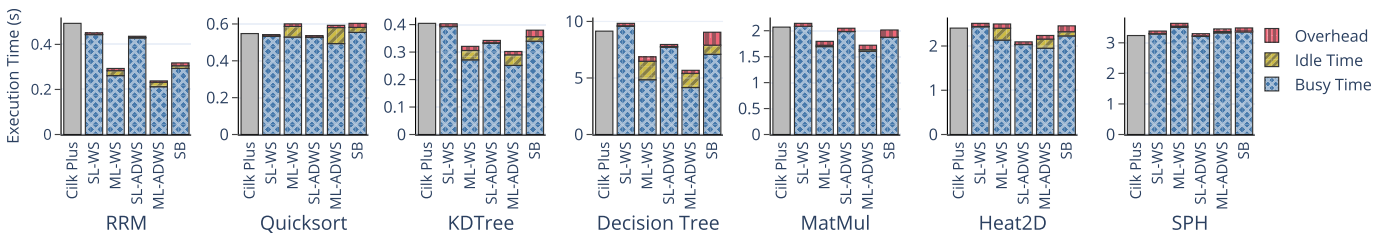


Fig. 17. Breakdown of the execution times profiled by the schedulers (corresponding to the rightmost points in Fig. 16).

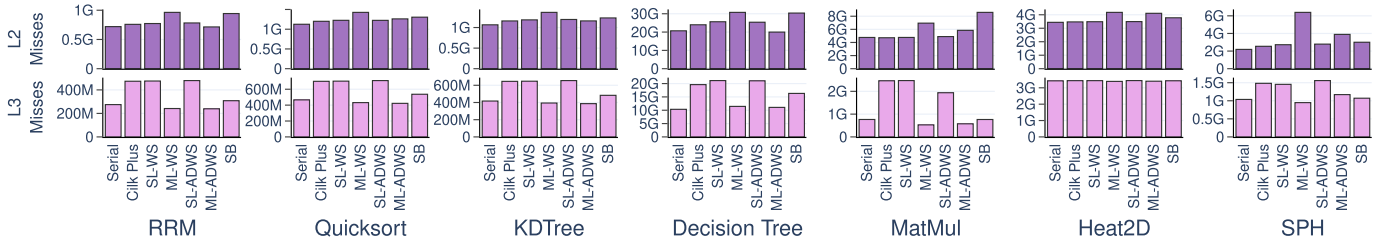


Fig. 18. Cache miss counts profiled by `perf` system calls (corresponding to the rightmost points in Fig. 16).

computation, level-l leaves can sometimes be so small that they expose little parallelism, which leads to underutilization of the cores sharing C_i^l.

The SB scheduler was expected to be tolerant of increases in the idle times in irregular computations, as discussed by Blelloch et al. [4]. However, our evaluation results showed a tradeoff between idleness and data locality in SB. The execution time breakdown (Fig. 17) shows that SB had shorter idle times than the multi-level schedulers, but it also had longer busy times. The cache miss counts (Fig. 18) show that SB greatly reduced the number of L3 misses as compared with the single-level schedulers, but it incurred more L3 misses than the multi-level schedulers in most cases. This was because SB tries to tie multiple tasks to a cache as long as the total working size of the tied tasks is less than or equal to the cache capacity. At the same time, however, the granularity of tasks tied to caches becomes smaller to fill up the remaining capacity of the caches, which reduces the chance of cache reuse within each task, thus incurring more L3 misses than with multi-level scheduling.

There was another tradeoff between private cache (L2) misses and shared cache (L3) misses. The trend was that the multi-level schedulers and SB incurred more L2 misses than the single-level schedulers, but fewer L3 misses. As SL-WS and SL-ADWS are designed so that each worker follows the serial execution order as much as possible, both can exploit data reuse that would occur naturally in serial execution, which makes them private-cache-friendly. In contrast, the shared-cache-aware schedulers (ML-WS, ML-ADWS, and SB) are likely to disturb the serial execution order for each worker so as to improve the shared cache utilization. Nevertheless, ML-ADWS had fewer L2 misses than ML-WS, because ADWS could reduce the number of steals through deterministic task mapping.

Comparison of the multi-level schedulers shows that ML-ADWS consistently performed better than ML-WS on RRM, KDTree, Decision Tree, and MatMul. All of these benchmarks have, at each recursion level, consecutive parallel computations whose data locality can be exploited by deterministic scheduling. The most obvious example is the Decision Tree benchmark (see Section 2.1), for which multi-level ADWS showed a 16% performance improvement with respect to multi-level WS with full training data.

Overall, ML-ADWS performed better than the other schedulers on many benchmarks over a wide range of data sizes, but for certain benchmarks, its performance was degraded by increased idle times (Quicksort) or increased L2 misses (SPH). These tradeoffs are imposed by the design of multi-level scheduling to improve shared cache utilization. Nevertheless, we can observe that either SL- or ML-ADWS performed the best on all of the benchmarks: when ML-ADWS did not perform well, SL-ADWS performed the best, and vice versa. A solution to the tradeoff issue would be to switch between SL- and ML-ADWS (perhaps partially by cache hierarchy flattening) according to the workload characteristics, such as the arithmetic intensity and the amount of parallelism. Consideration of the workload characteristics in addition to the programmer-specified working set sizes for application of cache hierarchy flattening would be an interesting future work.

6.4 Sensitivity to Imprecise Work Hints

In contrast to the above experiments in which appropriate work hints are given for ADWS, in this section, we study the performance of ADWS when work hints are imprecise or unavailable. Here, we suppose that work hints are not provided by programmers, and ADWS always guesses that the child tasks in a task group have the same amount of work. We first use the RRM benchmark to artificially generate unbalanced computation graphs by changing the work ratio parameter α. Without work hints from programmers, the work ratio of the two recursion calls in RRM is always presumed to be $1:1$, which is a random guess without a priori knowledge about work. Fig. 19 shows the performance results of RRM when changing α, which divides an array in the ratio $1:\alpha$ at each recursion. The y-axis represents performance improvements over SL-WS, which was calculated by $1 - T/T_{\mathrm{SL-WS}}$, where T is the execution time of each scheduler, and $T_{\mathrm{SL-WS}}$ is that of SL-WS. For comparison, it also shows the performance results of SL- and ML-ADWS that receive an appropriate work ratio of $1:\alpha$, which were

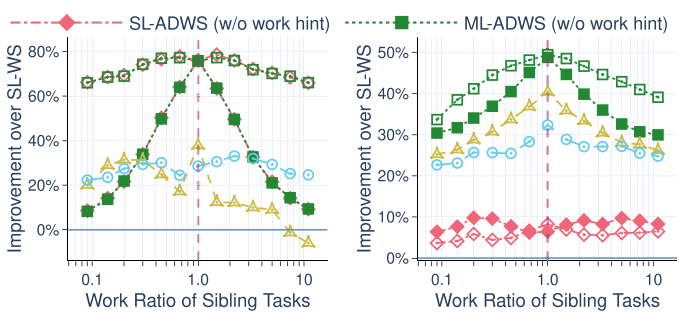
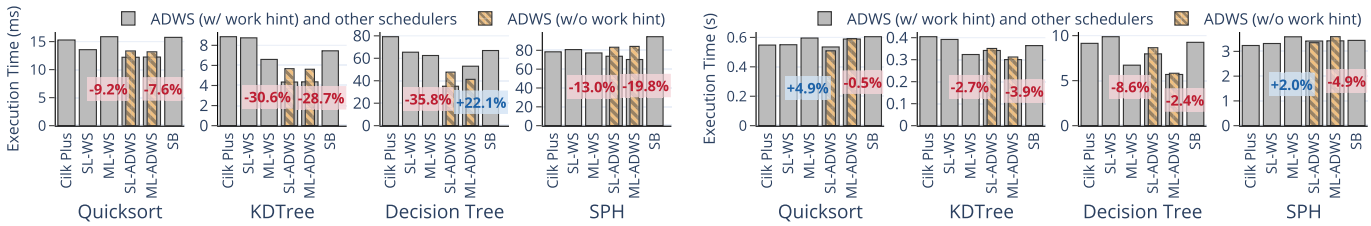


Fig. 19. Sensitivity of imprecise work hints for various unbalanced computation graphs. It shows performance improvements of the schedulers over SL-WS on the RRM benchmark with various work ratios (α parameter). In addition to the legends in Fig. 16, the performance of SL- and ML-ADWS without work hints, which guess that the work ratio is $1:1$, is shown with the filled markers.



(a) Working set sizes close to the total L3 cache size (Quicksort: 64 MB, KDTree: 64 MB, Decision Tree: 89 MB, SPH: 52 MB).

(b) Working set sizes that are the largest sizes in Fig. 16 (Quicksort: 2.0 GB, KDTree: 2.0 GB, Decision Tree: 4.5 GB, SPH: 2.2 GB).

Fig. 20. Evaluation of ADWS when no work hints are available. The hatched bars represent the execution times of ADWS without any work hints, along which the performance improvements over ADWS that was guided by programmers' work hints are shown (we anticipate negative values for performance improvements because no work hints were given).

plotted as the same, open markers as in Fig. 16, whereas those without programmer-provided work hints were plotted as filled markers.

The evaluation with the working set size of 64 MB (Fig. 19a), which fits into the system's L3 caches, shows that ADWS outperformed the other schedulers when α was close to 1. When the computation graph was highly unbalanced (i.e., when α was far from 1), the performance of ADWS was degraded when work hints were not provided, but it still had better performance than SL-WS. For example, even with $\alpha \sim 10$ (i.e., when the work ratio $1 : 10$ was assumed to be $1 : 1$), ADWS (without work hints) performed slightly better than SL-WS. This result indicates that ADWS is tolerant to load imbalance, even when work hints are imprecise or unavailable. This characteristic stems from the algorithm of dynamic load balancing explained in Section 3.2, in which ADWS localizes work stealing as much as possible and automatically regresses to conventional random work stealing when load imbalance is large. The evaluation with the large working set size of 1024 MB (Fig. 19b) shows similar results, in which ML-ADWS (w/o work hint) performed similarly to ML-WS when the computation graph is highly unbalanced, which means that ADWS at each cache level shortly regressed to conventional random work stealing when α is far from 1.

Next, we study the performance of other benchmarks: Quicksort, KDTree, Decision Tree, and SPH. We exclude Matmul and Heat2D because the random guess for work ratios is always correct for these benchmarks as their computation pattern is regular. Fig. 20 highlights the changes of performance of ADWS when work hints are not provided by programmers. In most cases, performance improvements of ADWS (without work hints) over ADWS (with work hints) were negative, which means that ADWS cannot fully exploit data locality if precise work hints are not provided. Nevertheless, the performance of ADWS (even without work hints) was comparable to that of other schedulers such as SL-WS. These results are consistent with those of RRM (Fig. 19) in that the degree of performance degradation was larger for working set sizes close to the total L3 cache size (Fig. 20a; up to 35.8% in Decision Tree) than for larger working set sizes (Fig. 20b; less than 10% performance drop). Note that the performance of ML-ADWS in Decision Tree was unstable in Fig. 20a because the working set size was slightly larger than the total L3 cache size (89 MB $>$ 77 MB) and thus the number of L3-leaf tasks was too small.

These results suggest that we can use ADWS as the default scheduler for nested parallel computations even without work hints provided by programmers. When a well-balanced computation graph is given, there is a chance to improve cache utilization by ADWS. On the other hand, when the computation graph is highly unbalanced, it can perform dynamic load balancing as well as conventional random work stealing. While the previous section showed that multi-level schedulers have several drawbacks, ADWS itself is robust to load imbalance. In addition, if programmers can provide appropriate work hints, the performance can be further improved.

6.5 NUMA Memory Policy for Deterministic Scheduling

To this point, we used `numactl -interleave=all` command, which distributes physical pages to all NUMA nodes as evenly as possible. However, by taking advantage of the deterministic scheduling in ADWS, we can optimize the memory mapping so that local memory accesses are increased. In this section, we study the performance of ADWS when the NUMA memory placement is properly arranged for ADWS. To allocate memory to NUMA nodes where the main computation will take place, we parallelized the initialization of the working set (with a memory access pattern that resembles the main computation) in each benchmark, so that physical pages are mapped to proper NUMA nodes by the *first-touch* policy[8].

Fig. 21 shows performance comparison between the NUMA interleave policy and the local allocation policy for ADWS. The working set sizes are the largest sizes in Fig. 16. For schedulers other than ADWS, we do not show the performance with the local allocation policy because their scheduling is not deterministic. Overall, SL-ADWS benefits from the local allocation policy more than ML-ADWS, because SL-ADWS causes more accesses to the main memory because of many L3 misses. Nevertheless, the performance of ML-ADWS is improved by about 20% in RRM and Heat2D, which are highly memory-bound and regular computations. For MatMul, which computes $C = AB$, we could not see performance improvements because the memory access pattern for matrix A and B is not regular.

8. In our experiments, to prevent physical pages from being dynamically migrated by *automatic NUMA balancing* in Linux, we used `mbind()` syscall to fix physical pages to specific NUMA nodes.

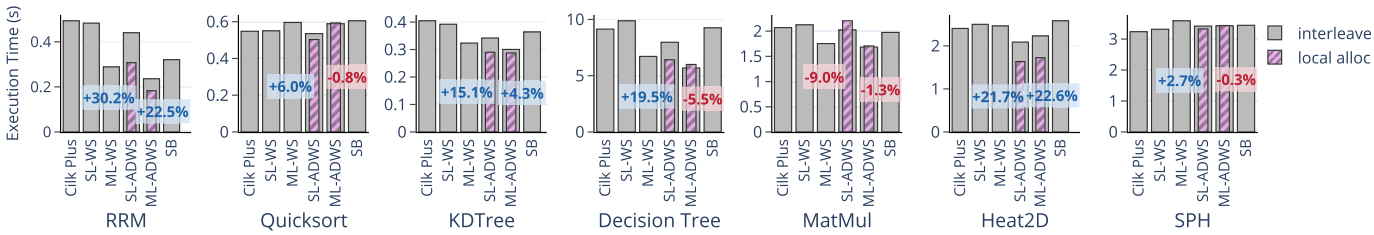


Fig. 21. Performance comparison between the NUMA interleave policy and the local allocation policy with ADWS. Working set sizes correspond to the rightmost points in Fig. 16. The performance improvements over the interleave policy are shown along with the hatched bars of the local allocation policy for ADWS.

7 Related Work

Conventional random work stealing [5] (explained above in Section 2.3) has been shown to exploit hierarchical data locality well on private caches [6], but not on shared caches. The parallel depth-first (PDF) scheduler [8], [42] was one of the earliest schedulers tailored for shared caches. The idea is that all workers simultaneously execute tasks that are as close as possible in depth-first order, i.e., an order similar to that of serial execution. Variants of PDF schedulers, such as *AsyncDF* [43], have also been proposed to reduce the overhead with fine-grained parallelism. An experimental study by Chen et al. [7] demonstrated that the PDF scheduler performs better than work stealing on a shared cache. However, the PDF scheduler cannot well exploit hierarchical data locality well on private caches, which has led to research on schedulers for both private caches and a shared cache. Narlikar [44] proposed a scheduler that combined work stealing with the PDF scheduler, and Blelloch et al. [17] proposed the *Controlled-PDF* scheduler. The Controlled-PDF scheduler applied a specific case of multi-level scheduling for two-level cache hierarchies with private caches and a single shared cache, where the PDF scheduling was applied at each level.

For machines with multiple shared caches, such as multi-socket, multi-core architectures, Chen et al. [15] proposed CATS, which introduced triple-level work stealing. Triple-level work stealing is also a specific case of multi-level scheduling for multi-socket, multi-core architectures, where conventional work stealing is applied at each level. In CATS, the working set sizes for each task are estimated by online profiling for iterative computations. LAWS [12], [13] is an improved version of CATS for NUMA architectures. It incorporates an initial task mapping at the level of NUMA nodes to promote local memory accesses. The initial task mapping of LAWS is different from that of ADWS in that the deterministic task mapping in ADWS can be applied to every level of a cache in a more general, unified way, whereas LAWS handles only NUMA nodes.

All of the above approaches assume specific hardware configurations with a fixed number of cache levels. The space-bounded (SB) scheduler [4], [9], [10] was proposed for arbitrary cache hierarchies, and theoretical analysis was provided. Simhadri et al. [18], [19] conducted the first experimental analysis of the SB scheduler and showed that it could reduce shared cache misses in practice. The idea of the SB scheduler is similar to that of multi-level scheduling in that it ties a task to a cache so that the task's descendants are executed only within that cache. On the other hand, the SB scheduler differs from multi-level scheduling in that it can tie multiple tasks to a cache as long as the tasks' total working set size does not exceed the cache capacity. Accordingly, as indicated by Blelloch et al. [4], the SB scheduler is expected to be more robust than multi-level scheduling against irregular computations, and our experimental results confirmed that finding. However, our evaluation also revealed that the SB scheduler caused more cache misses than multi-level scheduling in practice. A variant of the SB scheduler for programs with dynamic memory allocation has also been studied [45]. Nevertheless, none of the SB schedulers have addressed iterative data locality.

On the other hand, the issue of iterative data locality was pointed out by Acar et al. [6], and they proposed *locality-guided work stealing* to resolve the issue by setting an *affinity* for each task. The affinity specifies the preferred *places* at which a task should be executed, and many approaches have adopted concepts that are similar to affinity [46], [47], [48], [49], [50], [51], [52]. To determine the execution places, some approaches [47], [48], [51] require programmer-specified hints about places, and Drebes et al. [49], [50] used data dependency information obtained from a data-flow-based parallel runtime system (OpenStream [53]). In contrast, ADWS exploits iterative data locality through deterministic scheduling, and the hints for ADWS are oblivious to hardware-specific properties such as places.

Constrained work stealing [11] is also based on deterministic scheduling, in which the execution of nested parallel programs is traced and replayed for iterative programs. It utilizes a *steal tree* [54], which is a lightweight tracing mechanism for a work stealing schedule, and it allows for dynamic work stealing in addition to replay to fix load imbalances. However, because it uses conventional random work stealing to schedule the first iteration, it does not consider hierarchical data locality in deep cache hierarchies. In contrast, ADWS addresses both hierarchical and iterative data locality on arbitrary cache hierarchies in a unified way.

In addition, many studies have improved steal strategies for work stealing without relying on any scheduling hints by the programmer. Min et al. [55] proposed *hierarchical victim selection*, which attempts to steal from the nearest workers in hierarchical order, and Drebes et al. [49] adopted a similar approach in their system. *Probability Work Stealing (PWS)*, proposed by Quintin and Wagner [16], changes the probability of choosing victims for stealing so that close workers are preferably chosen as victims. These heuristic methods can somewhat improve cache utilization, but the

performance improvement is marginal as previously studied [14], [18], [19]. Quintin and Wagner [16] also proposed *Hierarchical Work Stealing (HWS)*, which is similar to the multi-level scheduling applied in this paper but operates on distributed memory. It splits tasks into *global* tasks, which can be stolen across nodes, and *local* tasks, which can only be stolen within a node. This idea of splitting tasks into two levels on distributed memory was also studied in [56], [57]. HWS was designed to reduce stealing across nodes rather than to limit the working set sizes for shared caches; accordingly, not much improvement in cache utilization is expected, because it determines global tasks on the basis of their depths rather than the working set sizes.

8 Conclusion

In this paper, we introduced two variants of ADWS to improve the cache utilization of nested parallel programs: single-level and multi-level ADWS. Single-level ADWS is based on a deterministic, cache-hierarchy-aware schedule with a little scheduling variety to allow for dynamic load balancing. Multi-level ADWS is built on single-level ADWS and combined with multi-level scheduling, which is a general scheduling framework for improving shared cache utilization. Unlike existing schedulers, ADWS is designed to exploit both hierarchical and iterative data locality on arbitrary cache hierarchies, while requiring some additional hints by programmers.

Our empirical performance analysis demonstrated that ADWS improved the cache utilization on many benchmarks with nested parallelism over a wide range of working set sizes. An interesting lesson that we learned from the experiments is that there are performance tradeoffs between single- and multi-level ADWS: multi-level ADWS incurs more private cache misses and longer core idle times, but it can greatly reduce shared cache misses. Considering that either single- or multi-level ADWS performed the best in almost all cases in our experiment, an interesting research direction would be to investigate automatic switching between single- and multi-level ADWS through online workload characterization. Moreover, we expect that the benefits of multi-level ADWS will be further highlighted with deeper and more complicated memory hierarchies for which the cost of shared cache misses is much more expensive, such as on distributed memory or future processors.

Acknowledgments

This article is based on results obtained from a project, JPNP16007, commissioned by the New Energy and Industrial Technology Development Organization (NEDO). This research was conducted using the Fujitsu PRIMERGY CX400M1/CX2550M5 (Oakbridge-CX) in the Information Technology Center, The University of Tokyo.

References

[1] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran, "Cache-oblivious algorithms," in *Proc. IEEE 40th Annu. Symp. Found. Comput. Sci.*, 1999, pp. 285–297.

[2] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran, "Cache-oblivious algorithms," *ACM Trans. Algorithms*, vol. 8, no. 1, pp. 4:1–4:22, Jan. 2012.

[3] G. E. Blelloch, P. B. Gibbons, and H. V. Simhadri, "Low depth cache-oblivious algorithms," in *Proc. 22nd Annu. ACM Symp. Parallelism Algorithms Archit.*, 2010, pp. 189–199.

[4] G. E. Blelloch, J. T. Fineman, P. B. Gibbons, and H. V. Simhadri, "Scheduling irregular parallel computations on hierarchical caches," in *Proc. 23rd Annu. ACM Symp. Parallelism Algorithms Archit.*, 2011, pp. 355–366.

[5] R. D. Blumofe and C. E. Leiserson, "Scheduling multithreaded computations by work stealing," *J. ACM*, vol. 46, no. 5, pp. 720–748, Sep. 1999.

[6] U. A. Acar, G. E. Blelloch, and R. D. Blumofe, "The data locality of work stealing," in *Proc. 12th Annu. ACM Symp. Parallel Algorithms Archit.*, 2000, pp. 1–12.

[7] S. Chen *et al.*, "Scheduling threads for constructive cache sharing on CMPs," in *Proc. 19th Annu. ACM Symp. Parallel Algorithms Archit.*, 2007, pp. 105–115.

[8] G. E. Blelloch, P. B. Gibbons, and Y. Matias, "Provably efficient scheduling for languages with fine-grained parallelism," *J. ACM*, vol. 46, no. 2, pp. 281–321, Mar. 1999.

[9] R. A. Chowdhury, F. Silvestri, B. Blakeley, and V. Ramachandran, "Oblivious algorithms for multicores and network of processors," in *Proc. IEEE 24th Int. Parallel Distrib. Process. Symp.*, 2010, pp. 1–12.

[10] R. A. Chowdhury, V. Ramachandran, F. Silvestri, and B. Blakeley, "Oblivious algorithms for multicores and networks of processors," *J. Parallel Distrib. Comput.*, vol. 73, no. 7, pp. 911–925, Jul. 2013.

[11] J. Lifflander, S. Krishnamoorthy, and L. V. Kale, "Optimizing data locality for fork/join programs using constrained work stealing," in *Proc. IEEE Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2014, pp. 857–868.

[12] Q. Chen, M. Guo, and H. Guan, "LAWS: Locality-aware work-stealing for multi-socket multi-core architectures," in *Proc. 28th ACM Int. Conf. Supercomput.*, 2014, pp. 3–12.

[13] Q. Chen and M. Guo, "Locality-aware work stealing based on online profiling and auto-tuning for multisocket multicore architectures," *ACM Trans. Archit. Code Optim.*, vol. 12, no. 2, pp. 22:1–22:24, Jul. 2015.

[14] S. Shiina and K. Taura, "Almost deterministic work stealing," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2019, pp. 1–6.

[15] Q. Chen, M. Guo, and Z. Huang, "CATS: Cache aware task-stealing based on online profiling in multi-socket multi-core architectures," in *Proc. 26th ACM Int. Conf. Supercomputing*, 2012, pp. 163–172.

[16] J.-N. Quintin and F. Wagner, "Hierarchical work-stealing," in *Proc. 16th Eur. Conf. Parallel Process.*, 2010, pp. 217–229.

[17] G. E. Blelloch, R. A. Chowdhury, P. B. Gibbons, V. Ramachandran, S. Chen, and M. Kozuch, "Provably good multicore cache performance for divide-and-conquer algorithms," in *Proc. 19th Annu. ACM-SIAM Symp. Discrete Algorithms*, 2008, pp. 501–510.

[18] H. V. Simhadri, G. E. Blelloch, J. T. Fineman, P. B. Gibbons, and A. Kyrola, "Experimental analysis of space-bounded schedulers," in *Proc. 26th ACM Symp. Parallelism Algorithms Archit.*, 2014, pp. 30–41.

[19] H. V. Simhadri, G. E. Blelloch, J. T. Fineman, P. B. Gibbons, and A. Kyrola, "Experimental analysis of space-bounded schedulers," *ACM Trans. Parallel Comput.*, vol. 3, no. 1, pp. 8:1–8:27, Jun. 2016.

[20] C. E. Leiserson, "The cilk concurrency platform," *J. Supercomputing*, vol. 51, no. 3, pp. 244–257, Mar. 2010.

[21] G. J. Narlikar, "A parallel, multithreaded decision tree builder," Carnegie Mellon University, Tech. Rep. CMU-CS-98–184, Dec. 1998.

[22] L. Breiman, J. Friedman, C. J. Stone, and R. A. Olshen, *Classification and Regression Trees*. Cambridge, MA, USA: CRC, 1984.

[23] K. Alsabti, S. Ranka, and V. Singh, "CLOUDS: A decision tree classifier for large datasets," in *Proc. 4th Int. Conf. Knowl. Discov. Data Mining*, 1998, pp. 2–8.

[24] R. Jin and G. Agrawal, "Communication and memory efficient parallel decision tree construction," in *Proc. SIAM Int. Conf. Data Mining*, 2003, pp. 119–129.

[25] G. Ke *et al.*, "LightGBM: A highly efficient gradient boosting decision tree," in *Proc. 31st Int. Conf. Neural Inf. Process. Syst.*, 2017, pp. 3149–3157.

[26] J. Reinders, *Intel Threading Building Blocks: Outfitting C for Multi-Core Processor Parallelism*. Chicago, IL, USA: O'Reilly Media, Jul. 2007.

[27] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An efficient multithreaded runtime system," in *Proc. 5th ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, 1995, pp. 207–216.

[28] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, " Cilk: An efficient multithreaded runtime system," *J. Parallel Distrib. Comput.*, vol. 37, no. 1, pp. 55–69, Aug. 1996.
[29] M. Frigo, C. E. Leiserson, and K. H. Randall, "The implementation of the Cilk-5 multithreaded language," in *Proc. ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 1998, pp. 212–223.
[30] Y. Guo, R. Barik, R. Raman, and V. Sarkar, "Work-first and help-first scheduling policies for async-finish task parallelism," in *Proc. IEEE 23rd Int. Symp. Parallel Distrib. Process. Symp.*, 2009, pp. 1–5.
[31] S. Shiina, S. Iwasaki, K. Taura, and P. Balaji, "Lightweight preemptive user-level threads," in *Proc. 26th ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, 2021, pp. 374–388.
[32] B. Alpern, L. Carter, and J. Ferrante, "Modeling parallel computers as memory hierarchies," in *Proc. IEEE Workshop Program. Models Massively Parallel Comput.*, 1993, pp. 116–123.
[33] J. Nakashima and K. Taura, "MassiveThreads: A thread library for high productivity languages," *Concurrent Objects Beyond*, vol. 8665, pp. 222–238, Jan. 2014.
[34] Intel 64 and IA-32 Architectures Software Developer's Manual – Volume 3B: System Programming Guide, Part 2, Intel Corporation, Nov. 2020.
[35] Intel Xeon Processor Scalable Memory Family Uncore Performance Monitoring Reference Manual, Intel Corporation, Jul. 2017.
[36] D. Dua and C. Graff, "UCI machine learning repository," 2017. [Online]. Available: http://archive.ics.uci.edu/ml
[37] P. Baldi, P. Sadowski, and D. Whiteson, "Searching for exotic particles in high-energy physics with deep learning," *Nature Commun.*, vol. 5, no. 4308, pp. 1–9, 2014.
[38] R. D. Blumofe, M. Frigo, C. F. Joerg, C. E. Leiserson, and K. H. Randall, "An analysis of dag-consistent distributed shared-memory algorithms," in *Proc. 8th Annu. ACM Symp. Parallel Algorithms Archit.*, 1996, pp. 297–308.
[39] M. Becker and M. Teschner, "Weakly compressible SPH for free surface flows," in *Proc. ACM SIGGRAPH/Eurograph. Symp. Comput. Animation*, 2007, pp. 209–217.
[40] M. Iwasawa, A. Tanikawa, N. Hosono, K. Nitadori, T. Muranushi, and J. Makino, "Implementation and performance of FDPS: A framework for developing parallel particle simulation codes," *Pub. Astronomical Soc. Jpn.*, vol. 68, no. 4, pp. 54:1–54:22, Jun. 2016.
[41] J. Barnes and P. Hut, "A hierarchical $o(n\log n)$ force-calculation algorithm," *Nature*, vol. 324, pp. 446–449, Dec. 1986.
[42] G. E. Blelloch and P. B. Gibbons, "Effectively sharing a cache among threads," in *Proc. 16th Annu. ACM Symp. Parallelism Algorithms Architectures*, 2004, pp. 235–244.
[43] G. J. Narlikar and G. E. Blelloch, "Space-efficient scheduling of nested parallelism," *ACM Trans. Program. Lang. Syst.*, vol. 21, no. 1, pp. 138–173, Jan. 1999.
[44] G. J. Narlikar, "Scheduling threads for low space requirement and good locality," *Theory Comput. Syst.*, vol. 35, no. 2, pp. 151–187, Jan. 2002.
[45] G. E. Blelloch, P. B. Gibbons, and H. V. Simhadri, "Provably efficient scheduling of dynamically allocating programs on parallel cache hierarchies," in *Proc. IEEE 24th Int. Conf. High Perform. Comput.*, 2017, pp. 124–133.
[46] A. Robison, M. Voss, and A. Kukanov, "Optimization via reflection on work stealing in TBB," in *Proc. IEEE 22nd Int. Parallel Distrib. Process. Symp.*, 2008, pp. 1–8.
[47] J. Deters, J. Wu, Y. Xu, and I.-T. A. Lee, "A NUMA-aware provably-efficient task-parallel platform based on the work-first principle," in *Proc. IEEE Int. Symp. Workload Characterization*, 2018, pp. 59–70.
[48] Y. Guo, J. Zhao, V. Cave, and V. Sarkar, "SLAW: A scalable locality-aware adaptive work-stealing scheduler," in *Proc. IEEE 24th Int. Parallel Distrib. Process. Symp.*, 2010, pp. 1–12.
[49] A. Drebes, K. Heydemann, N. Drach, A. Pop, and A. Cohen, "Topology-aware and dependence-aware scheduling and memory allocation for task-parallel languages," *ACM Trans. Archit. Code Optim.*, vol. 11, no. 3, pp. 30:1–30:25, Aug. 2014.
[50] A. Drebes, A. Pop, K. Heydemann, A. Cohen, and N. Drach, "Scalable task parallelism for NUMA: A uniform abstraction for coordinated scheduling and memory management," in *Proc. Int. Conf. Parallel Architectures Compilation*, 2016, pp. 125–137.
[51] J. Maglalang, S. Krishnamoorthy, and K. Agrawal, "Locality-aware dynamic task graph scheduling," in *Proc. IEEE 46th Int. Conf. Parallel Process.*, 2017, pp. 70–80.
[52] V. Kumar, "PufferFish: NUMA-aware work-stealing library using elastic tasks," in *Proc. IEEE 27th Int. Conf. High Perform. Comput. Data, Analytics*, 2020, pp. 251–260.
[53] A. Pop and A. Cohen, "OpenStream: Expressiveness and data-flow compilation of openmp streaming programs," *ACM Trans. Archit. Code Optim.*, vol. 9, no. 4, pp. 53:1–53:25, Jan. 2013.
[54] J. Lifflander, S. Krishnamoorthy, and L. V. Kale, "Steal tree: Low-overhead tracing of work stealing schedulers," in *Proc. 34th ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2013, pp. 507–518.
[55] S.-J. Min, C. Iancu, and K. Yelick, "Hierarchical work stealing on manycore clusters," in *Proc. 5th Conf. Partitioned Glob. Address Space Program. Models*, 2011, pp. 1–10.
[56] Y. Wang *et al.*, "An adaptive and hierarchical task scheduling scheme for multi-core clusters," *Parallel Comput.*, vol. 40, no. 10, pp. 611–627, Dec. 2014.
[57] J. Paudel, O. Tardieu, and J. N. Amaral, "On the merits of distributed work-stealing on selective locality-aware tasks," in *Proc. IEEE 42nd Int. Conf. Parallel Process.*, 2013, pp. 100–109.

Shumpei Shiina (Graduate Student Member, IEEE) received the BS and MS degrees from the University of Tokyo, in 2019 and 2021, respectively. He is currently working toward the PhD degree with the University of Tokyo. His major research interests include programming models and runtime systems for task parallelism, memory management on distributed memory, and efficient thread scheduling.

Kenjiro Taura received the BS, MS, and DSc degrees from the University of Tokyo, in 1992, 1994, and 1997, respectively. He is currently a professor in the Department of Information and Communication Engineering, The University of Tokyo. His major research interests include parallel/distributed computing and programming languages. His expertise includes efficient dynamic load balancing, parallel and distributed garbage collection, and parallel/distributed workflow systems.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/csdl.