# Almost Deterministic Work Stealing

Shumpei Shiina
shiina@eidos.ic.i.u-tokyo.ac.jp
University of Tokyo
Tokyo, Japan

Kenjiro Taura
tau@eidos.ic.i.u-tokyo.ac.jp
University of Tokyo
Tokyo, Japan

## ABSTRACT

With task parallel models, programmers can easily parallelize divide-and-conquer algorithms by using nested fork-join structures. *Work stealing*, which is a popular scheduling strategy for task parallel programs, can efficiently perform dynamic load balancing; however, it tends to damage data locality and does not scale well with memory-bound applications. This paper introduces *Almost Deterministic Work Stealing (ADWS)*, which addresses the issue of data locality of traditional work stealing by making the scheduling *almost* deterministic. Specifically, ADWS consists of two parts: (i) *deterministic task allocation*, which deterministically distributes tasks to workers based on the amount of work for each task, and (ii) *hierarchical localized work stealing*, which dynamically compensates load imbalance in a locality-aware manner. Experimental results show that ADWS is up to nearly 6 times faster than a traditional work stealing scheduler with memory-bound applications, and that dynamic load balancing works well while maintaining good data locality.

## CCS CONCEPTS

• **Computing methodologies → Shared memory algorithms**.

## KEYWORDS

Work Stealing, Task Parallelism, Locality, Load Balancing

## 1 INTRODUCTION

Task parallel models are promising approaches to achieving both high performance and productivity on modern multicore computers. Task parallel models have an important characteristic of being *processor-oblivious*, which means that programmers can create many tasks regardless of the number of processors. Moreover, in task parallel models, divide-and-conquer algorithms can be expressed by using nested fork-join structures, which well matches memory hierarchies in modern computers.

Numerous strategies have been proposed to effectively schedule a computation graph of task parallel programs. There are offline scheduling approaches, which determine the mapping of tasks to processors prior to execution, perhaps at compile time [18]. In order for offline scheduling to be effective, tasks' execution times and dependencies between them need to be known ahead of time, which is often not the case. In such cases, online dynamic scheduling is necessary. Up to now, many dynamic schedulers for task parallel programs have been developed. There are programming languages supporting lightweight dynamic task scheduling, including Cilk [6, 15], Chapel [8] and X10 [9]. There are also task parallel runtime libraries which don't require compiler support, including Intel TBB [27], Argobots [29] and MassiveThreads [23]. Task parallelism was also introduced in OpenMP 3.0 [3] and has evolved to support dependencies and affinities [24].

*Work stealing* is a popular strategy to schedule task parallel programs dynamically. In work stealing, each worker executes tasks autonomously from its own local task queue until it is exhausted; it then tries to steal tasks from another worker (called victim). A victim is usually chosen randomly from all workers (*random work stealing*), and this randomness causes the problem of data locality. There are mainly two issues. One is that it does not respect machine's hierarchy of locality; workers on the same socket are likely to work on tasks remote on the computation graph, which are therefore likely to access non-overlapping data. The other issue is it breaks locality of iterative computations [1], which repeats similar access patterns across iterations. Such applications often benefit from repeating the same task mapping across iterations, thereby reusing data on the cache across iterations. In practice, random work stealing works well within one socket [7], but it is less likely to scale to multiple sockets or multiple levels of the memory hierarchy (e.g., NUMA).

To mitigate the problem of data locality in random work stealing, a number of strategies have been proposed. There are approaches that require hardware-specific locality hints of programmers [12, 16], which take non-trivial programmer efforts. HotSLAW [22] proposed to mitigate the first issue by a heuristic that first attempts to steal from a victim within a close proximity of the stealing worker. Many others try to address the second issue by repeating the same task mapping across multiple iterations [10, 11, 19, 21, 35]. Obviously, they are applicable only to iterative applications.

This paper explores a simpler, arguably more straightforward, approach to the problem of data locality of random work stealing, called Almost Deterministic Work Stealing (ADWS), which schedules tasks *almost* deterministically. Specifically, it consists of two parts: *deterministic task allocation* and *hierarchical localized work stealing*. ADWS addresses the locality issue of iterative applications by making the scheduling almost deterministic (predictable from the computation graph). ADWS targets nested fork-join programs, and the issue of mismatch between task hierarchy and machine hierarchy is resolved by making both the initial deterministic task mapping and the dynamic work stealing respect the machine hierarchy. For the initial deterministic task mapping, ADWS asks the programmer to specify a hint on the amount of work done by the generated task. This is certainly a burden on the programmer and

*SC '19, November 17–22, 2019, Denver, CO, USA*
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6229-0/19/11…$15.00
https://doi.org/10.1145/3295500.3356161

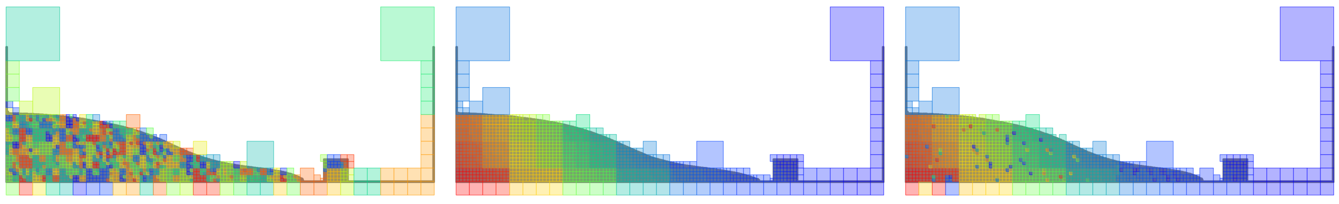


**Figure 1: A visualization of the task mapping among 64 workers in particle simulation of 2D dam breaking. The rectangles are cells of the quadtree (corresponding to computations) overlaid on Figure 15. The color of the cells represents the rank of workers, and the workers are colored gradually from blue to red in numerical order.**

may not be readily available, but they do not have to be very precise as the dynamic load balancing is always there to fix up the load imbalances of the deterministic mapping. Also, as we show in Section 4.1, the amount of work can be a relative number (the amount of the child's work relative to that of the parent). ADWS is different from other strategies in several ways. (i) unlike some work only dealing with bag-of-tasks without any dependencies among them, it can work for more general and powerful nested fork-join programs, (ii) it addresses both of the issues mentioned above — locality of iterative applications and memory hierarchy-aware scheduling — in a simple unified algorithm. ADWS was implemented on MassiveThreads [23] and experimental results show it outperforms other existing scheduling methods.

Figure 1 helps us understand how ADWS works. It visualizes the task mapping overlaid on the figure of 2D particle simulation (Figure 15). Figure 1a shows how tasks are mapped on cores by random work stealing, whereas Figure 1b and Figure 1c show the mapping of deterministic and "almost" deterministic schedulers, respectively. In ADWS (Figure 1c), workers steal tasks to compensate the load imbalance based on the task distribution of Figure 1b while maintaining most of the data locality.

To summarize the main contributions, this paper

(1) proposes Almost Deterministic Work Stealing (ADWS) scheduler (Section 4), which consists of
    (a) deterministic task allocation, which initially allocates tasks to workers deterministically at runtime (Section 4.2) and
    (b) hierarchical localized work stealing, which dynamically steals from a limited range of workers (Section 4.3), and
(2) shows ADWS outperforms other existing scheduling strategies with memory-bound applications (Section 5).

## 2 RELATED WORK

Affinity-based approaches are often used to mitigate data locality problems for iterative applications. First, *locality-guided work stealing* was proposed in [1]. When a task with an affinity for a worker is created, the task is pushed to both the local queue and the mail box of that worker. Workers try to pop tasks from the mailbox before attempting to steal. Work in [28] solved the problem of the initial task distribution in locality-guided work stealing; that is, workers stealing inappropriate tasks before tasks with affinity for them arrive.

*Work pushing*, which is similar to the affinity-based approach, was investigated in [12–14]. In [13, 14], data locality was optimized by using data dependency information in OpenStream [26], which is a data-flow extension of OpenMP. The programming model is different from the basic fork-join model. NUMAWS [12] adopts a basic fork-join programming model, and they proposed an efficient work pushing algorithm based on the *work-first* principle [15]. It requires hardware-specific hints from programmers to obtain good performance.

Where locality hints from programmers are concerned, SLAW [16] and NUMAWS [12] require annotations for *places* where the task should be executed (e.g., NUMA nodes), which are hardware-specific. While ADWS requires hints from programmers, it only requires application-specific hints. DistWS [25] requires application-specific hints rather than hardware-specific ones, but the hints are not intuitive for programmers because they are not transparent to how DistWS works. In ADWS, the hints (the amount of work for tasks) are transparent to how ADWS works, and therefore the programmers don't have to know how it works or even what work stealing is.

HotSLAW [22] extends the principle of SLAW, but it does not require locality hints from programmers. The authors proposed *hierarchical victim selection* (which attempts to steal from the nearest workers in hierarchical order) and *hierarchical chunk selection* (which determines the number of tasks to steal dynamically based on the distance in the memory hierarchy). However, HotSLAW does not address the locality problem in iterative applications.

Some approaches use the structure of iterative applications to improve data locality [10, 11, 19, 21, 35]. ADWS is different from these approaches because the application of ADWS is not limited to iterative applications. *Constrained work stealing* [21] schedules tasks deterministically for iterative applications by tracing and replaying. Its previous work [20] showed the execution of task parallel programs can be traced and replayed by using *StealTree*, and constrained work stealing revises the traced scheduling by relaxing the scheduling constraints (e.g., allow steals while replaying) for the first several iterations. However, because random work stealing is used to schedule the first iteration, there is no guarantee that workers in the same NUMA node will execute close tasks in the computation graph.

Like ADWS, there are also approaches that combine the initial partitioning of tasks with work stealing. Work in [34] proposed an approach that used offline analysis for tasks with no dependencies (*bag-of-tasks*). The initial distribution of tasks is determined by using offline analysis, and they used a hierarchical work stealing strategy based on the hardware topology. ADWS is different from their approach in that ADWS targets tasks with dependencies (nested fork-join models) and does not require offline analysis at compile time. AHS [32] initially distributes tasks to each node rather than to each core, which means data in the private cache cannot be reused in iterative applications. CATS [11] and LAWS [10] are similar to AHS in the sense that they initially distribute tasks to the NUMA node level, and they are only optimized for iterative applications (as remarked above).

*Localized work stealing* was proposed in [30], which attempts to steal its own tasks from workers who have previously stolen them by maintaining a list of workers who are working on the tasks. Localized work stealing is similar to hierarchical localized work stealing that is described in this paper in the sense that it dynamically makes some groups preferable to steal from.

## 3 BACKGROUND

### 3.1 Task Parallelism

The computation of task parallel programs is expressed as a directed acyclic graph (DAG). Figure 2 is an example of a DAG. A DAG consists of nodes (corresponding to computations) and edges (representing dependencies between nodes). There are three types of edges: *spawn* edges, *continue* edges and *join* edges. When a new task is spawned, a *spawn* edge and a *continue* edge connect the parent node to the spawned task and to the continuation of the parent task, respectively. In Figure 2, the outlined edges represent *spawn* edges, and the solid edges represent *continue* edges. A chain of nodes connected with *continue* edges represents a task. In the rest of this paper, we will place a spawned task to the left of the parent, and the continuation to the right of the parent, as shown in Figure 2. *Join* edges are required to wait for the completion of other tasks, as represented by the dotted edges in Figure 2.

A *fully-strict* computation is one in which all *join* edges from a task go to the parent of the task [7]. Many problems can be expressed in the form of a *fully-strict* computation, e.g., problems solvable with divide-and-conquer algorithms. Figure 3 is a simplified expression of a DAG of *fully-strict* computations. This example spawns four child tasks at a time and waits for them recursively. Some nodes and edges are omitted from general representation of a DAG (like Figure 2) to make the fork-join structure clear. We shall refer to a group of child tasks spawned at the same time "task group". Figure 6 is an example of task groups created in programs.

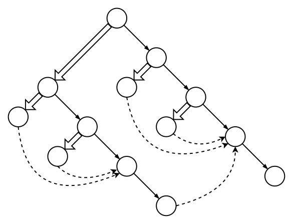


**Figure 2: Example of a DAG in task parallelism.**

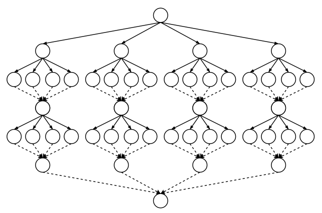


**Figure 3: Expression of a DAG of *fully-strict* computations.**

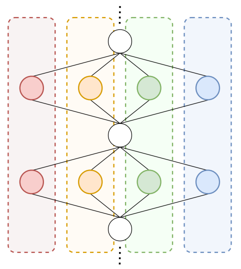


**Figure 4: Locality of iterative applications. If the program sweeps the same data repeatedly in the same order, the nodes within each dotted box share data locality.**

### 3.2 Work Stealing

*Work stealing* is a popular strategy for scheduling task parallel computations. A work stealing scheduler has a set of *workers*, which usually correspond to processors or cores. Each worker has its own task queue. When a new task is spawned, a worker pushes the task or the continuation to the local task queue and executes the other. When a worker completes the current task, it pops a task from its own task queue and starts to execute it. Workers push/pop tasks to/from the task queue in last-in, first-out (LIFO) order. If the local task queue is exhausted, the worker (thief) tries to steal tasks from the task queue of another worker (victim). The thief tries to steal until a steal succeeds. The thief steals tasks from the opposite end of the task queue, i.e., in first-in, first-out (FIFO) order, which means the stolen task is the oldest task spawned by the victim in the queue. Victims are usually chosen randomly from all workers (*random work stealing*). Because of this, the mapping of tasks is determined randomly.

Next, we describe two scheduling policies, *work-first* and *help-first* [33]. With the *work-first* policy, a worker first executes the spawned task and puts the continuation into the local task queue at the time of the spawn. With the *help-first* policy, a worker first executes the continuation and puts the spawned task into the local task queue. With the *work-first* policy, the serial execution order is preserved because we usually execute function calls first in serial execution. This is an important characteristics because programs are usually optimized for serial execution, and following this order preserves the data access pattern in serial execution.

### 3.3 Locality of Random Work Stealing

Random work stealing is known to have good data locality in computation graphs on flat shared-memory machines [1], i.e., each worker tends to execute close nodes in the DAG. However, workers in groups that have the same level of locality do not always execute close nodes in the DAG. Since close nodes in a DAG are likely to have computations that will touch overlapping data, it will cause more cache misses. For example, in NUMA architectures, traditional random work stealing degrades performance because workers on the same socket do not work on close tasks in a DAG.

As pointed out in [1], there is also data locality among tasks that are not close in DAGs for iterative applications. Figure 4 illustrates data locality of iterative applications. Usually, iterative applications sweep the data in the same order at each iteration, which means

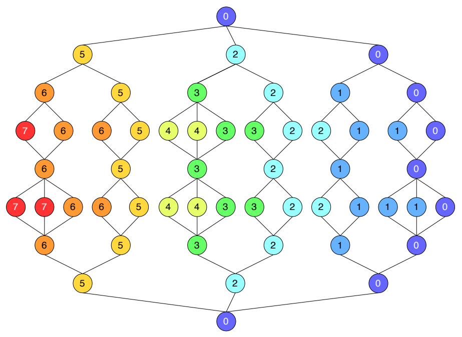


Figure 5: Example of the desired distribution of tasks we propose. The number in the nodes denotes the worker who executes the task. The tasks are split vertically in the DAG and allocated to workers from right to left in numerical order.

data locality exists vertically in the DAG. In random work stealing, workers tend to execute tasks at different locations at each iteration due to its randomness, so the cache cannot be reused for consecutive iterations.

## 4 ALMOST DETERMINISTIC WORK STEALING

We had to guarantee both locality for iterative applications and locality in the computation graph for multilevel memory hierarchies. Deterministic scheduling is considered to be a good solution for ensuring data locality for iterative applications. In addition to this, we had to allocate close tasks in a DAG to workers in the same group (e.g., NUMA node). Figure 5 shows the desired distribution of tasks. Each worker is allocated almost the same amount of work (load balancing), and workers are placed from right to left [1] in numerical order so that workers in the same group have tasks close to each other in the DAG (We assumed that workers in the same group were numbered adjacently). We also wanted the scheduler to dynamically balance the load, because load imbalances can occur even if the initial partitioning of tasks is not bad (e.g., because of OS noise, CPU frequency scaling).

To achieve the task distribution as seen in Figure 5, we propose *Almost Deterministic Work Stealing (ADWS)*. ADWS consists of two parts. The first part is *deterministic task allocation*, which initially allocates tasks to each worker deterministically based on the amount of work for each task specified by programmers. The second part is *hierarchical localized work stealing*, in which workers steal tasks from close workers based on the task distribution done by deterministic task allocation when load imbalances appear.

### 4.1 Programming Model

In ADWS, programmers have to specify the amount of work for each task. Even if the amount of work is roughly known and not precise, hierarchical localized work stealing is expected to compensate the load imbalance in a locality-aware manner. Although specifying the amount of work requires additional effort from programmers,

[1]By placing workers from right to left in the DAG, we can preserve the serial execution order (from left to right) for each worker as well as the *work-first* policy (see Section 4.2.2). Recall that first spawned tasks are placed in the left as noted in Section 3.1.

```
task_group tg;
tg.run([]{ ... });
tg.run([]{ ... });
tg.run([]{ ... });
tg.run([]{ ... });
tg.wait();
```

```
task_group tg(w_all);
tg.run([]{ ... }, w1);
tg.run([]{ ... }, w2);
tg.run([]{ ... }, w3);
tg.run([]{ ... }, w4);
tg.wait();
```

Figure 6: TBB's task group (left) and ADWS's extension (right).

it is application-specific and programmers can write programs regardless of the machine architecture, i.e., it is processor-oblivious and has good code portability.

We adopted TBB-like task group notation [27] as the programming model, and the class of problems is restricted to *fully-strict* computations in ADWS. We can spawn tasks and wait for them by using task groups as shown in Figure 6. In ADWS, we only have to specify the total amount of work in each task group (`w_all`) and the amount of work for each spawned task (`w1, w2, w3, w4`). We assume `w_all == (w1 + w2 + w3 + w4)`, and they don't need to be absolute values. It is sufficient to specify the ratio of work for each spawned task relative to the total work in the task group (`w_all`). We could force `w_all` to be always 1 and remove the parameter `w_all`, but to explicitly specify a denominator simplifies programs in most cases (see below).

The execution order is also an important factor in ADWS. The execution order of tasks by each worker is almost the same as the serial execution order (see Section 4.2.2). Because of this, programmers should write programs that work efficiently in serial execution, i.e., the data access pattern should be optimized for serial execution. If it is an iterative program, the data access pattern should be almost the same for consecutive iterations, so that workers access the same data for consecutive iterations. ADWS distributes tasks in the same order even for consecutive task groups in each task, which means accessing data in the same order for consecutive task groups enables better data locality (see matrix-multiplication example in Section 4.1.2).

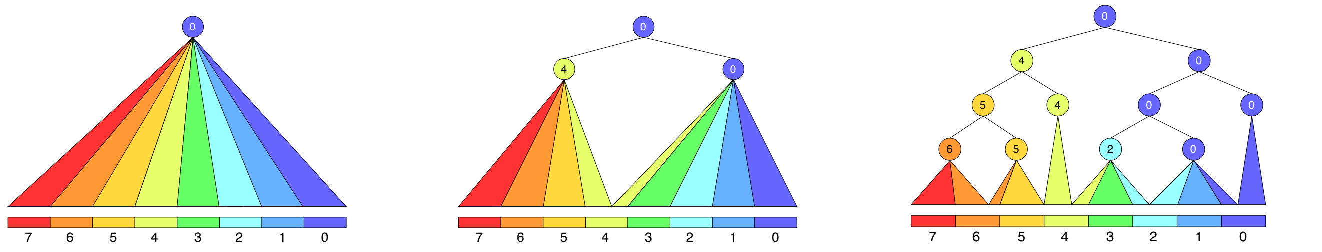
#### 4.1.1 Example 1. Calculation of Particle Interactions.

We introduce particle simulation as an example of a way to specify the amount of work for each task. For example, FDPS [2] manages a group of particles by using the Barnes-Hut octree structure [4]. With task parallelism, we can easily parallelize the calculation of particle interactions by traversing the octree.

```
particle_interaction(node) {
  if (node is leaf) {
    Calculate particle interactions in node;
  } else {
    task_group tg(node.n_particles);
    for (child in node.children) {
      tg.run([]{particle_interaction(child);}, child.n_particles);
    }
    tg.wait();
  }
}
```

This code specifies the number of particles in a node as the amount of work for the task. Though the number of particles is an approximation for the amount of work, we can use a rough estimate of work in ADWS.

[2]A framework for developing parallel particle simulation codes [17]



**(a) First, suppose that only one task exists and worker 0 is executing the task. The range of workers covers all workers, i.e., the descendant tasks of the task are distributed among all workers.**

**(b) When a child task is spawned, the range of workers is divided into two subranges according to the ratio of work for the spawned task (the left node) and the continuation (the right node).**

**(c) The distribution of tasks is determined recursively and in parallel. A node which has multiple workers in its range is executed by the rightmost worker in the range (i.e., the worker with the smallest rank).**

**Figure 7: Overview of deterministic task allocation for eight workers. The bottom rectangles represent the number line of workers, and the triangles below the nodes represent the range of workers for each node. The number in the node denotes the worker that executes the node.**

#### 4.1.2 *Example 2. Matrix Multiplication* .

We also introduce the matrix-multiplication (*matmul* in short) as an example, which calculates $C = AB$, where $A$, $B$, and $C$ are matrices. By dividing the matrices into four submatrices, we get

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} = \begin{pmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{pmatrix}$$

We can get a result by using the products of submatrices, so we can apply the *divide-and-conquer* algorithm.

```
1  matmul($A$, $B$, $C$) {
2    if (size_of($C$) < CUTOFF) {
3      $C$ += $AB$;
4    } else {
5      task_group tg1(4);
6      tg1.run([]{ matmul($A_{11}$, $B_{11}$, $C_{11}$); }, 1);
7      tg1.run([]{ matmul($A_{21}$, $B_{11}$, $C_{21}$); }, 1);
8      tg1.run([]{ matmul($A_{11}$, $B_{12}$, $C_{12}$); }, 1);
9      tg1.run([]{ matmul($A_{21}$, $B_{12}$, $C_{22}$); }, 1);
10     tg1.wait();
11     task_group tg2(4);
12     tg2.run([]{ matmul($A_{12}$, $B_{21}$, $C_{11}$); }, 1);
13     tg2.run([]{ matmul($A_{22}$, $B_{21}$, $C_{21}$); }, 1);
14     tg2.run([]{ matmul($A_{12}$, $B_{22}$, $C_{12}$); }, 1);
15     tg2.run([]{ matmul($A_{22}$, $B_{22}$, $C_{22}$); }, 1);
16     tg2.wait();
17   }
18 }
```

There are two task groups in a task (line 5-10 and line 11-16). `tg1` calculates the former terms ($A_{11}B_{11}$, $A_{21}B_{11}$, $A_{11}B_{12}$, and $A_{21}B_{12}$), and `tg2` calculates the latter terms ($A_{12}B_{21}$, $A_{22}B_{21}$, $A_{12}B_{22}$, and $A_{22}B_{22}$). To avoid concurrent writes to the $C$ buffer, `tg2` is executed after `tg1` is completed. The matrices are divided recursively until the size of matrices becomes small enough (line 2). Because each task in a task group is expected to have the same amount of work, we simply specify 4 as the total amount of work in the task group and 1 as the amount of work for each task.

Note that the order of calculating submatrices of $C$ in `tg1` (line 6-9) is the same as that of `tg2` (line 12-15). In ADWS, workers calculate close to the same location of submatrices of $C$ in consecutive task groups, and this can lead to good locality across consecutive task groups. We can consider the locality of iterative applications to be a specific case of locality of consecutive task groups; an application is called iterative when the root task has multiple task groups.

## 4.2 Deterministic Task Allocation

Here we describe *deterministic task allocation*, which is the central part of ADWS. In deterministic task allocation, first, workers determine the distribution of tasks so that the load is balanced, and then workers execute tasks allocated to them. We can also view it as a static load balancing algorithm for nested fork-join programs.

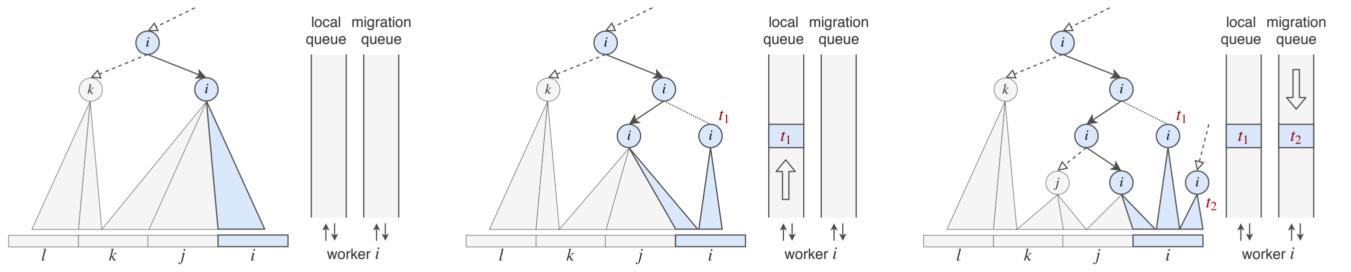
#### 4.2.1 *Basic Idea.*

Let us consider what we should do when a new task is spawned. Because we want to complete the spawned task and the continuation in the same execution time, we allocate "an amount of workers" to them proportionally to the amount of work for each task [3]. We said "an amount of workers" instead of "the number of workers" because it can be a floating-point number. In the implementation of deterministic task allocation, we manage "a range of workers" rather than an amount of workers. A range of workers is represented by two floating-point numbers, which are represented as points on the number line of workers. This idea is illustrated in Figure 7. The rectangles at the bottom represent the number line of workers, and we call them the "work region". We assumed that there was only one task in the first stage and worker 0 was executing the task while the others were idle. The initial range of workers covered all the workers, because we wanted to distribute all descendant tasks evenly to all the workers. When a task is spawned, we divide the current range of workers into two subranges based on the amount of workers, as calculated by the amount of work for the spawned task and the continuation. The amount of work for the spawned task is given by the programmers. Subsequently, the range of workers is recursively divided into small subranges.

#### 4.2.2 *Algorithm* .

In deterministic task allocation, the workers distribute tasks while executing the DAG. First, workers dive into the bottom of the DAG to find the left boundary of the worker's own

[3]If we take a critical path into account, the spawned task and the continuation are not completed in the same execution time. The expected execution time of fully-strict computations by using a random work stealing scheduler with $P$ workers is given as $T_P = T_1/P + O(T_\infty)$, where $T_1$ is the total amount of work of the computation and $T_\infty$ is the critical path [2, 7]. If the parallelism ($T_1/T_\infty$) is sufficiently larger than $P$, i.e., $T_1/T_\infty \gg P$, our assumption $T_P \approx T_1/P$ is true.



**(a) A search-root task is migrated to worker $i$ and a search starts. At a spawn, the range of workers is divided into two subranges in the middle of worker $k$'s work region, and worker $i$ migrates the spawned task to worker $k$ and executes the continuation.**

**(b) When the range of workers is divided at worker $i$'s work region, worker $i$ puts the continuation into its local queue and executes the spawned task. The tasks in its local queue are executed after the search finishes.**

**(c) The search by worker $i$ continues until worker $i$ encounters the end of task execution or a *wait*. Other workers can migrate tasks to worker $i$, and worker $i$ executes them after all tasks in its local queue are completed.**

**Figure 8: Search algorithm of deterministic task allocation carried out by worker $i$. The arrows with solid lines represent the path of the search by worker $i$, and the ones with dotted lines represent task migration.**

work region. We call this step "search", in an analogy to searching a value in a binary tree.

Workers migrate tasks to other workers during a search. We define the term "search-root task" as a task in which its range of workers is across multiple workers. A task that has one worker in the range of workers is called a "non-search-root task". Figure 8 illustrates the search algorithm. First, worker $i$ receives a search-root task from other workers. Then, worker $i$ starts searching for the left boundary of its own work region. If the range of workers is divided at the work region of another worker, worker $i$ migrates the spawned task to the worker at the boundary and executes the continuation. If it is divided at worker $i$'s own work region, worker $i$ puts the continuation to its local queue and executes the spawned task. By following this algorithm, tasks are always assigned to the rightmost worker in their range of workers. The search finishes when the task being executed is completed, or when the worker encounters a *wait*. We also describe the search algorithm in the pseudocode below.

```
1  thread local variables { rank, cur_work, cur_range };
2  create_task_group(total_work) { cur_work = total_work; }
3  create_task(new_task, work) {
4    if (searching) {
5      worker_amount = cur_range.left - cur_range.right;
6      middle = cur_range.left - worker_amount * work / cur_work;
7      left_range = {left: cur_range.left, right: middle};
8      right_range = {left: middle, right: cur_range.right};
9      new_task.range = left_range;
10     if (rank == (int)middle) { /* Go to the left */
11       cur_range = left_range;
12       Put continuation to local queue;
13       run new_task;
14     } else { /* Go to the right */
15       Migrate new_task to worker (int)middle;
16       cur_work -= work;
17       cur_range = right_range;
18     }
19   } else { /* Go to the left (work-first) */
20     Put continuation to local queue;
21     run new_task;
22   }
23 }
```

`worker_amount` at line 5 represents the amount of workers in the current node, and `middle` at line 6 denotes the division point of the range of workers in the number line of workers.

Search-root tasks are put in a special buffer (as discussed in Section 4.2.3) and can be executed immediately because there can be one search-root task at a time (∵ Theorem 2). The continuations put into the local queue during a search are executed after the search in LIFO order, and the non-search-root tasks migrated from other workers are executed in FIFO order after all tasks in the local queue are completed. Tasks are executed by the *work-first* policy while not searching. By following this execution order, we can preserve the serial execution order, i.e., the execution order is from left to right in figures.

We had assumed that worker 0 executes the root task initially, but this assumption is not true in practice. For example, in iterative programs, we cannot predict which worker will execute the root task when an iteration finishes, so we have to explicitly migrate the root task to worker 0 after the completion of all *waits*. In some cases, a task has multiple task groups, like the matmul example (Section 4.1.2). We also have to migrate a search-root task back to its owner (*return migration*) after a task group in the search-root task is completed. The owner of the task is defined as the rightmost worker in the range of workers. Then, the owner starts a search from the next task group using the same range of workers as the previous task group. Figure 5 helps us understand how tasks in consecutive task groups are distributed.

### 4.2.3 *Implementation of Task Migration*

We introduce some lemmas and theorems, and then we describe the implementation of task migration using the characteristics induced by the theorems. Here, we simply write a range of workers as $[i, j]$ $(i \leq j)$ by using the rank of each worker (an integer value); worker $i$ is the rightmost worker and worker $j$ is the leftmost worker in the figures [4]. Thus the owner of this range is defined as worker $i$. If worker $i$ has the range $[i, j]$ while searching, worker $i$ has the privilege of migrating tasks to worker $i + 1, \ldots, j$ (worker $i$ cannot migrate tasks to itself).

[4]Workers are placed from right to left in numerical order in the figures, which is the reverse order of the literal notation $[i, j]$ on the paper.

First, we consider the implementation of the queue for non-search-root tasks.

Lemma 1. *Let $i$ be an arbitrary number of workers except for 0 ($i = 1, \ldots, P-1$). Suppose that worker $j$ ($0 \le j < i$) has the privilege of migrating non-search-root tasks to worker $i$ during a search. Then worker $j$ either delegates the privilege to another worker or keeps the privilege without delegation through any spawn.*

Proof. Worker $j$ has the range of workers $[j, k]$ ($j < i \le k$) because it has the privilege of migrating non-search-root tasks to worker $i$. When worker $j$ spawns a task, the range of workers is divided into two subranges $[j, l]$ and $[l, k]$ ($j \le l \le k$) (the range is split at the middle of worker $l$'s work region). If $l = j$, worker $j$ puts the continuation to its local queue and executes the spawned task; therefore worker $j$ keeps the privilege. Otherwise, worker $j$ executes the continuation with the new range $[j, l]$, and the spawned task is migrated to worker $l$. If $i \le l$, worker $j$ keeps the privilege without delegation; otherwise, worker $j$ loses the privilege and worker $l$ gets the privilege instead. □

Lemma 2. *Let $i$ be an arbitrary number of workers except for 0 ($i = 1, \ldots, P-1$). Suppose that only worker $j$ ($0 \le j < i$) has the privilege of migrating non-search-root tasks to worker $i$, and worker $j$ encounters a task group ($TG_1$) during a search. Then only one worker has the privilege at the same time until all tasks in $TG_1$ are completed, and the privilege is returned to worker $j$ when $TG_1$ is completed.*

Proof. Because of Lemma 1, only one of the owners of the child tasks in $TG_1$ gets (or keeps) the privilege. Let it be worker $k$ ($j \le k < i$), and let $T_k$ denote the child task of $TG_1$ executed by worker $k$. Now let us assume that this lemma is true for worker $k$. If worker $k$ encounters the first task group in $T_k$, only one worker has the privilege at the same time until the task group is completed, and the privilege is returned back to worker $k$ with $T_k$ under this assumption. The same is true of the consecutive task groups in $T_k$. When $T_k$ is completed, worker $k$ loses the privilege and the privilege is returned to the owner of $TG_1$ (i.e., worker $j$). As a base case, if $T_k$ has no task groups, the privilege is immediately returned to worker $j$. Thus the lemma is proved recursively. □

Theorem 1. *Let $i$ be an arbitrary number of workers except for 0 ($i = 1, \ldots, P-1$). Then workers do not migrate non-search-root tasks to worker $i$ simultaneously at any point in time.*

Proof. At the beginning of the program, only one task exists, and only worker 0 has the privilege of migrating non-search-root tasks to worker $i$. Because of this, we can apply Lemma 2; therefore only one worker can migrate non-search-root tasks to worker $i$ at any point throughout the program. □

By Theorem 1, migration of non-search-root tasks does not require lock operations (**lock-free**). Because of this, we can implement queues without any locks or CAS operations for deterministic task allocation. We need locks if we enable hierarchical localized work stealing, but even if we use queues with locks, we anticipate that the lock contention is unlikely to happen during a search because multiple workers never push tasks to the queue at the same time, and steals rarely happen during a search because of the management of *steal ranges* (see Section 4.3).

Next, we discuss how search-root tasks should be treated.

Lemma 3. *Let $i$ be an arbitrary number of workers ($i = 0, \ldots, P-1$). Then once search-root task $T_i$ is migrated to worker $i$, other workers can migrate only non-search-root tasks to worker $i$ during searches until $T_i$ is completed.*

Proof. $T_i$ must have range $[i, j]$ ($i < j$), and if worker $k$ ($0 \le k < i$) can migrate $T_i$ to worker $i$, worker $k$ must have range $[k, j]$. Once worker $k$ migrates $T_i$ to worker $i$, range $[k, j]$ is divided into $[k, i]$ and $[i, j]$ and worker $k$ continues the search from $[k, i]$. In the following search from $[k, i]$, workers can only migrate non-search-root tasks (with range $[i, i]$) to worker $i$. Worker $k$ only regains range $[k, j]$ after $T_i$ is completed. □

Theorem 2. *Let $i$ be an arbitrary number of workers ($i = 0, \ldots, P-1$). Then the number of search-root tasks migrated to worker $i$ and not popped by worker $i$ is at most one at any point in time.*

Proof. Because of Lemma 3, we only have to consider the return migration of search-root tasks while not searching. Worker $i$ starts a search when search-root task $T_i$ is migrated, and some search-root tasks with range $[i, j]$ ($i < j$) are spawned during the search. $T_i$ is migrated back to worker $i$ when a task group in $T_i$ is completed. Because of this, $T_i$ is not migrated back until all descendant search-root tasks are completed. The same is true of descendant search-root tasks, and therefore search-root tasks are only migrated back to worker $i$ after worker $i$ pops the previous search-root task. □

By Theorem 2, it is sufficient for each worker to have one buffer for the migration of search-root tasks in the implementation. Summarizing, what we need per worker is two queues and a buffer for the migration. Although we can achieve LIFO for tasks pushed during searches and FIFO for migrated non-search-root tasks with one queue per worker, we have two queues in ADWS (the local queue and the migration queue). This is because the steal strategy in hierarchical localized work stealing can be simplified by splitting the queue. The local queue is used for tasks generated during a search and their descendants, and the migration queue is for non-search-root tasks migrated from other workers and their descendants.

### 4.2.4 Problems with a Complex DAG .

One problem with deterministic task allocation is that, with a *complex DAG*, workers are sometimes forced to be idle for a while (i.e., the scheduler is not greedy). If nested tasks have multiple task groups, we call it a complex DAG. For example, task parallel matrix-multiplication (Section 4.1.2) has a complex DAG. On the other hand, A *simple* DAG has a single task group per task, such as the example of particle interactions in Section 4.1.1.

When dividing the range of workers during a search, we allow the range to be divided in the middle of a work region, i.e., the amount of workers allocated to tasks can be a floating-point number. Because of this, workers often execute tasks from distinct task groups. This is not a problem with a simple DAG, but it can be a problem with a complex DAG (see Figure 9). Tasks shaded in the middle of the DAG are allocated to the same worker $i$. First, worker $i$ starts a search from the left-side task group, and then executes tasks created during the search. The tasks in the right-side task group,

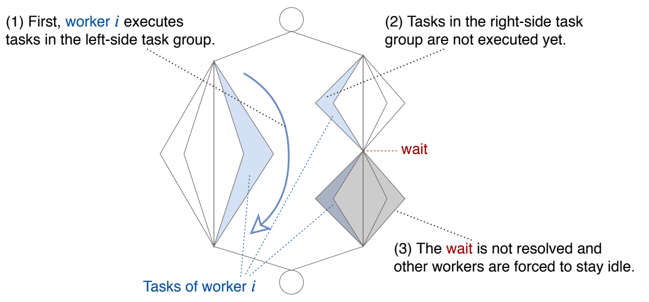

(1) First, worker $i$ executes tasks in the left-side task group.
(2) Tasks in the right-side task group are not executed yet.
wait
(3) The wait is not resolved and other workers are forced to stay idle.
Tasks of worker $i$

**Figure 9: Example of the problem with a complex DAG in deterministic task allocation.**

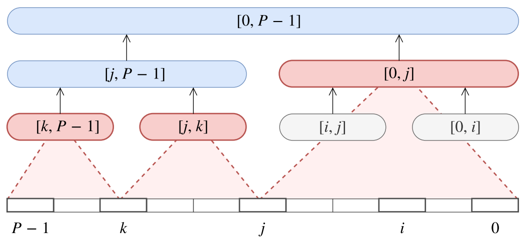


**Figure 10: The tree of steal ranges. The rectangles with the rounded corners with a range of workers represent steal ranges. Steal ranges are activated from the bottom up, and finally only the root steal range $[0, P-1]$ is activated. In this case, the active steal ranges are $[0, j]$, $[j, k]$ and $[k, P-1]$.**

which are migrated from other workers, are executed after that. If there is a *wait* in the middle of the right-side task group, workers in the right-side task group have to wait for worker $i$ to execute tasks in the right-side task group, and the following task groups cannot be executed because of it. Deterministic task allocation ensures the amount of work assigned to a worker is almost the same, but it doesn't provide the proper execution order. Deciding the proper execution order is not obvious because, if we execute tasks on one side, workers on the other side can be forced to wait. We can solve this problem with hierarchical localized work stealing, which is described in the next section.

## 4.3 Hierarchical Localized Work Stealing

In hierarchical localized work stealing, the range of victims is limited based on the task distribution done by deterministic task allocation and is dynamically updated from bottom up according to the completion status of the task groups.

*4.3.1 Algorithm .* As an additional procedure during a search, workers need to construct a tree of *steal ranges*. A steal range contains three elements: a range of workers, a pointer to the parent steal tree, and an active flag. When a worker encounters a task group during a search, the worker appends a new steal range to the tree. Figure 10 shows an example of the tree of steal ranges.

At the beginning of the execution, only worker 0 has a steal range $[0, P-1]$, where $P$ is the number of workers. Each worker holds its current steal range. Here is the basic rule:

- A current steal range of each worker (except for worker 0) is first set to the current steal range of the worker who migrates a search-root task to it.
- Workers append a newly created steal range to their current steal range when it encounters a task group during a search.
- During a search, workers pass their current steal range to other workers with the migration of search-root tasks.
- When a search finishes, the current steal range is activated.
- When a task group in the current steal range is completed, the steal range is deactivated, and the current steal range is set to the parent.
- When a search-root task is completed, the current steal range is activated.

When a steal range is activated, its descendants should be deactivated. For example, in Figure 10, if steal range $[0, i]$ is deactivated and the parent $[0, j]$ is activated by worker 0, the range $[i, j]$ should be also deactivated even if the range $[i, j]$ is still active. This is because if the parent $[0, j]$ is activated, the tasks in the range $[i, j]$ can be stolen by workers in the range $[0, i]$, and workers in the range $[i, j]$ may not be able to find tasks in the range $[i, j]$ if we do not deactivate the range $[i, j]$. This deactivation is done autonomously; that is, a worker checks if there are active ancestors of the current steal range from the bottom up every time the worker tries to steal. If the worker finds active ancestors, the worker deactivates the current steal range and updates its current steal range to the topmost active ancestor.

When there is no task, workers try to steal tasks if the current steal range is active. Victims are randomly chosen from the current steal range. Because workers should not steal tasks outside of the task group in their current steal range, we must consider which queue they should steal a task from. For example, thieves whose current steal range is $[j, k]$ in Figure 10 should not steal tasks in $[k, P-1]$ or $[0, j]$. To avoid inappropriate steals, we split the task queue into two parts: the *local queue* and the *migration queue*, as also remarked in Section 4.2.3. In that case, thieves whose current steal range is $[j, k]$ should steal tasks (i) in the local queue of worker $j$, (ii) in the migration queue of worker $k$, or (iii) in both queues of other workers in $[j, k]$.

We should also consider whether or not to migrate search-root tasks back to its owner, especially for complex DAGs. If any of the ancestors of the current steal range are already active, the tasks migrated during a search can be stolen from outside of the current steal range even if we start a search. Because of this, when any of the ancestors of the current steal range are already active, workers do not migrate a search-root task back to its owner and do not start a search from the following task groups in the search-root task.

Note that steals rarely happen during a search, because workers deactivate their current steal range when a task group in search-root tasks is completed [5]. That is, steals within the current steal range are suppressed during the search from the next task group. Then, contention of the migration queue can be avoided (see also Theorem 1), and the search can be done quickly. Though tasks can be stolen from outside of the current steal range if an ancestor of the current steal range is activated during a search, it does not occur so frequently.

[5]The parent steal range is activated when the search-root task is completed without encountering the next task group

| # of sockets | microarchitecture | model | # of cores | frequency | L1 data cache | L2 cache | L3 cache |
|---|---|---|---|---|---|---|---|
| 4 | Skylake | Xeon Gold 6130 | 16 | 2.1 GHz | 32 KB/core | 1 MB/core | 22 MB/socket |

**Table 1: CPU information used in the experiments.**

*4.3.2 Greediness* . Assuming a search can be done quickly, we can regard ADWS as an "almost" greedy scheduler, with some modification of the scheduler. The scheduler is *greedy* if workers always execute tasks whenever ready tasks are present. We call ADWS an "almost" greedy scheduler because this does not give any theoretical proof and only describes the characteristics of ADWS.

By following the hierarchical localized work stealing algorithm, it can be ensured that uncompleted tasks remain in the task group of the current steal range, and workers are expected to find uncompleted tasks in it. The scheduler is not greedy by that alone, because there is a blocker: a search-root task. A search-root task cannot be stolen by anyone, because it must be executed by the owner. We can solve this problem by executing the search-root task immediately after migration. It is sufficient to make it "almost" greedy because there is at most one search-root task per worker because of Theorem 2, and workers can always start to execute the search-root task immediately. On a practical level, workers cannot immediately start to execute a search-root task, but by executing it when a worker completes a task or reaches a *wait*, some of the greediness can be gained.

## 5 PERFORMANCE EVALUATION

We conducted experiments in a NUMA architecture to compare the performance of ADWS to that of other existing scheduling methods. The machine used in these experiments had 4 sockets, and each socket had 16 cores (64 cores in total). Information about the CPUs is shown in Table 1.

We implemented ADWS on *MassiveThreads* [23], which is a library for lightweight task parallelism with the *work-first* scheduling policy. We also implemented *hierarchical work stealing* (*hierarchical WS*) as an existing locality-aware scheduling strategy. In hierarchical WS, workers first try to steal tasks from the nearest workers in the memory hierarchy, and after several attempts, they try to steal from a broader range of workers. This scheduling strategy was proposed in [22], and work in [13] also adopted the same approach as a part of their system.

We compared the performance of (i) random work stealing (random WS), (ii) hierarchical WS, (iii) ADWS with no steal (i.e., deterministic task allocation only), and (iv) ADWS (with deterministic task allocation and hierarchical localized work stealing) for all benchmarks. These scheduling algorithms were all implemented on MassiveThreads. For the following measurements, we distributed workers in a *scatter* manner. This meant that the workers were distributed to all sockets as evenly as possible. In a scatter manner, we can utilize all of the available LLC and high memory bandwidth in NUMA. We compiled programs with gcc 5.4.0, and the version of the linux kernel was `4.4.0-141-generic`.

### 5.1 Heat2D Benchmark

We conducted the experiments on heat2D benchmark. Heat2D benchmark calculates a heat transfer (5-point stencil) in a square region ($N \times N$). It repeatedly updates the temperature of every point in the region. Since heat2D is iterative and highly memory-bound, data locality can significantly affect its performance.

We parallelized heat2D by using the divide-and-conquer algorithm, which divides the region into four square regions recursively, and the cutoff size was set to $64 \times 64$. To make it more memory-bound, we optimized the calculation kernel by using SIMD instructions. We used two buffers to calculate the stencil and added the padding of the cache line size (64 bytes in this case) to each matrix to avoid cache conflicts. The experiments were conducted using $N = 2048$, $N = 4096$ and $N = 8192$ matrices with single precision floating-point numbers, and they used 32 MB, 128 MB, and 512 MB of memory, respectively. We measured the execution time for 1000 iterations, and the experimental results were the median of ten executions.

For comparison, we implemented *constrained work stealing* (constrained WS) [21], which is an existing locality-aware scheduling policy specific to iterative applications. In constrained WS, the first iteration executes an ordinary random work stealing while recording the trace of work stealing using StealTree [20]. The next few iterations then revise the scheduling, by combining the replay and work stealing; workers basically follow the recorded schedule but still perform work stealing when they become idle (*RelWS*). Finally, subsequent iterations purely replay the recorded schedule without performing any work stealing (*StOWS*). We iterated *RelWS* for the first five iterations, because it showed the best performance.

Figure 11 shows the speedup with `numactl -iall`. With `numactl -iall` command, allocated memory is distributed across NUMA nodes almost evenly. The base case is the execution time of a single core execution without `numactl -iall`. As shown in Figure 11, OpenMP static and ADWS performed better than others. In many cases, the performance of OpenMP static is better than that of ADWS, which is because of tasking overheads of ADWS. Hierarchical WS did not improve the performance because it was oblivious to the locality of iterative applications. Constrained WS worked better than random WS and hierarchical WS, but the performance was worse than that of OpenMP static and ADWS.

There are two reasons for the performance difference between Constrained WS and ADWS. One is that the execution of the first iteration was scheduled by random WS, and data locality among tasks within a worker or socket was not optimized. In this experiment setting, the size of leaf tasks was relatively small ($64 \times 64$); therefore the effect of data locality among close tasks was significant, which leaded to the lower performance of constrained WS. In addition to this, the number of branches of *StealTree* monotonically increases as work stealing occurs in *RelWS*, which means that the DAG gets decomposed into smaller parts and data locality among tasks gets worse. The other reason is that the idle time of workers, caused by the change of execution time for each task, was not negligible. Tasks moved from the previous iteration by work stealing in *RelWS* have relatively bad data locality at this time, and the execution time of them is usually longer than that of tasks not stolen. Then data locality of these stolen tasks improves in the next iteration and their execution time gets shorter, which causes additional idle time for workers. This problem can be amortized by increasing the number of iterations for *RelWS*, but there is a trade-off between the idle time and the over-decomposed issue remarked above. On the other hand, in ADWS, the task mapping is nearly optimal and the idle time of workers is negligible because of its greediness (Section 4.3.2).

In Figure 11a, OpenMP static and ADWS show superlinear speedup. The performance of the single core execution was relatively bad because the data (32 MB) did not fit into one L3 cache (22 MB) in single core execution, but it fitted into four L3 caches with multiple sockets. Since OpenMP static and ADWS caused few DRAM accesses, they showed superlinear speedup.

Figure 11b shows the speedup with $N = 4096$, the data of which does not fit into L2 cache. The ratio of change in the speedup of ADWS (no steal) or ADWS increased with the number of workers. This can be explained by examining the cache architecture. The cache architecture of Skylake is *non-inclusive cache*, which means the L2 cache can have data that the L3 cache doesn't have. Therefore, the total available cache size increases with the number of workers, and the performance improves accordingly.

When $N = 8192$, the performance difference became small (Figure 11c). This was because the data (512 MB) did not fit into the caches and accesses to DRAM occurred frequently. In this case, the physical location of data was determined by `numactl -iall` policy, and many remote memory accesses occurred regardless of scheduling policies. OpenMP static and ADWS can avoid this problem by utilizing *first-touch* policy, which is the default memory allocation policy in Linux. In heat2D benchmark, we can allocate most of the memory to each worker's local DRAM by simply parallelizing the initialization of matrices in the same manner as the calculation.

Figure 12 shows the speedup when using *first-touch* policy with parallel initialization. The performance of OpenMP static and ADWS improved, whereas that of others did not change from Figure 11. The performance improvement clearly appears in Figure 12c compared to Figure 11c. In this case, ADWS performed better than deterministic schedulers (ADWS (no steal) and OpenMP static), which indicates there were load imbalances even if the load can be statically divided. Constrained WS could not take advantage of NUMA-aware memory allocation, because tasks are over-decomposed and the size of tasks are small compared to the page size. Figure 12a shows ADWS was up to nearly six times faster than random WS.

## 5.2 Matrix-Multiplication Benchmark

For the matrix-multiplication (matmul) benchmark, we calculated the multiplication of dense square matrices with size $N \times N$. The matmul benchmark is non-iterative and has a complex DAG. The implementation of the matmul benchmark is described in Section 4.1.2. The values of the matrices were single precision floating-point numbers, and the size of matrices used were $N = 2048$, $N = 4096$ and $N = 8192$. We also added padding to each matrix as well as heat2D, and we set the cutoff size to 128x128. Since the naive implementation of matmul was rather computation-bound, we optimized the calculation kernel by using SIMD instructions. The experimental results were the median of ten executions.

Figure 13 shows a comparison for GFLOPS achieved in the matmul benchmark. The theoretical peak performance in this environment was 134.4 GFLOPS per core at 2.1 GHz. We conducted this experiment with `numactl -iall`. In all cases in Figure 13, ADWS outperformed other methods with a large core count. Although the performance of hierarchical WS was better than that of random WS, ADWS performed the best. Notably, ADWS achieved 4085 GFLOPS with $N = 4096$, which is nearly half of the theoretical peak GFLOPS with 64 cores (8601.6 GFLOPS).

The performance of ADWS (no steal) was poor because of the problem with a complex DAG (described in Section 4.2.4). Closer scrutiny of the result reveals that ADWS (no steal) performed well when the number of workers was a power of two. As previously mentioned, we divided the matrices into four submatrices and spawned four tasks with the same amount of work. Then, when the number of workers was a power of two, the range of workers was not divided in the middle of the work region, and the problem with a complex DAG did not occur. The result shows that even if deterministic task allocation had a problem, ADWS performed well because of hierarchical localized work stealing.

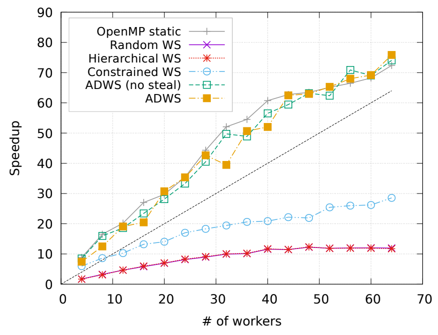
## 5.3 Calculation of Particle Interactions (SPH)

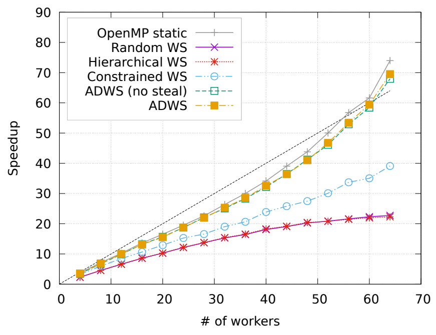
We also compared the performance of ADWS with that of the existing implementation of FDPS [17]. First, the original implementation of FDPS makes an array of leaf nodes by traversing the octree, and then uses OpenMP *for* loop for the array of leafs. Because the amount of work for each leaf can vary and a static scheduling can cause significant load imbalances, it uses a dynamic scheduling policy (*schedule(dynamic, 4)*). Section 4.1.1 shows the task parallel implementation for the calculation of particle interactions. In the task parallel implementation, the computation was parallelized while traversing the octree. This is more straight-forward than the original implementation using loop-based parallelism.

To evaluate the performance, we used FDPS to write particle simulation code of *Smoothed Particle Hydrodynamics (SPH)*. It simulated the dynamics of water (dam breaking) as shown in Figure 15, and we referenced [5] to implement it. With SPH, short-range interactions of particles within an effective radius are calculated at every iteration. We modified FDPS (v5.0b) to implement the task parallel version. In this study, we only compared the performance of the calculation of particle interactions, although we can parallelize other parts such as building trees by using task parallelism as ExaFMM does [31].

We showed a visualization of the task distribution in Figure 1. With random WS, tasks were distributed randomly and data locality was poor. The task distribution of OpenMP dynamic was similar to that of random WS (not shown due to space limitations). With ADWS (no steal), close tasks were allocated to close workers and data locality was good, but it could not dynamically balance the load. ADWS dynamically balanced the load while maintaining most of data locality.

Figure 14 shows a comparison of the speedup of 2D and 3D simulation of the dam breaking with $N$ particles. We measured only the particle interaction parts for 1000 iterations, and the result was the median of ten executions. We conducted this experiment


(a) $N = 2048$


(b) $N = 4096$

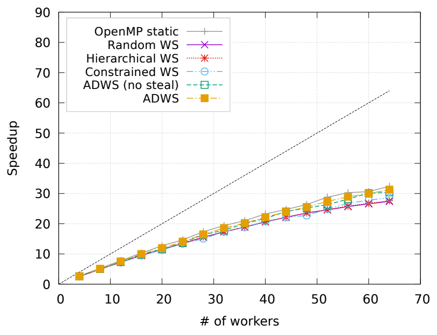
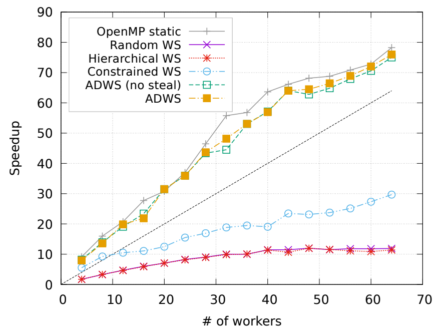
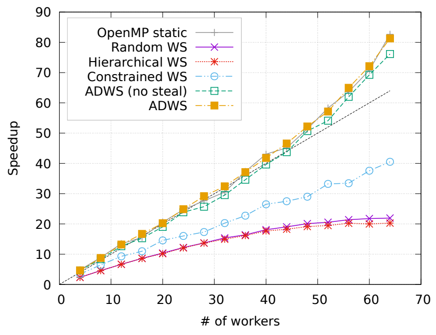

(c) $N = 8192$

**Figure 11: Comparison of the speedup of heat2D benchmark with `numactl -iall`. The straight dotted line represents the ideal linear speedup.**


(a) $N = 2048$


(b) $N = 4096$

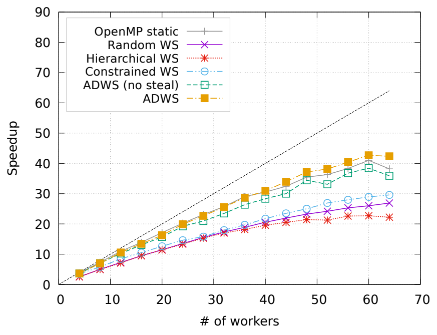
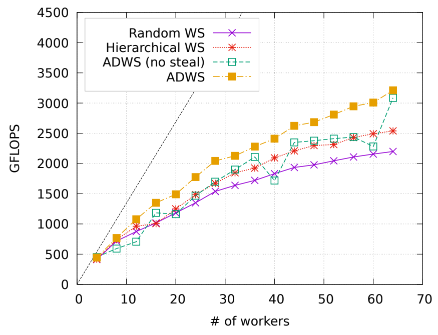
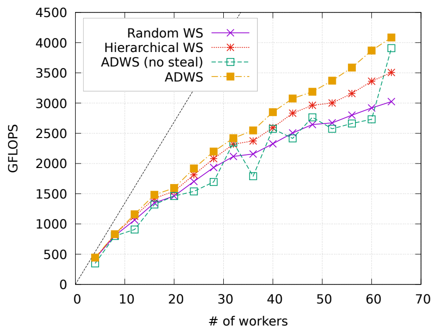

(c) $N = 8192$

**Figure 12: Comparison of the speedup of heat2D benchmark with NUMA-aware initialization with the *first-touch* policy. The straight dotted line represents the ideal linear speedup.**


(a) $N = 2048$


(b) $N = 4096$

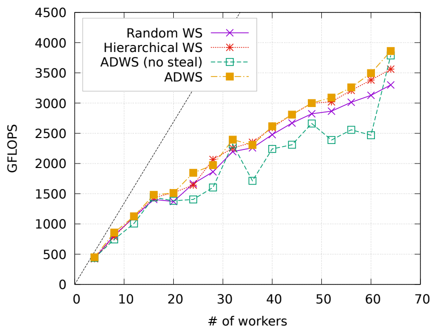
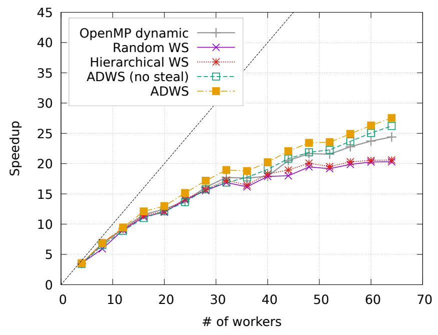
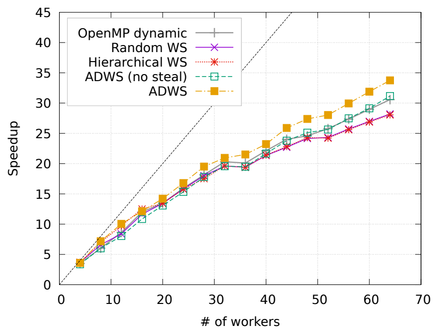

(c) $N = 8192$

**Figure 13: Comparison of GFLOPS of matmul benchmark. The straight dotted line represents the theoretical peak performance (134.4 GFLOPS/core).**


(a) 2D, $N = 35888$


(b) 2D, $N = 138968$

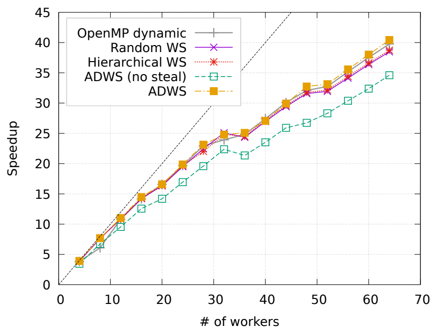

(c) 3D, $N = 182364$

**Figure 14: Comparison of speedup of 2D and 3D dam breaking simulation using FDPS. The straight dotted line represents the ideal linear speedup.**

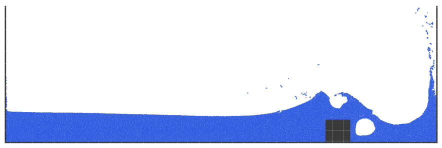


Figure 15: 2D dam breaking simulation with SPH

with `numactl -iall`, and the base case was the execution time of serial execution without OpenMP nor MassiveThreads.

Figure 14a and 14b show the speedup of the 2D simulation. In both cases, the performance of ADWS was better than that of others. Figure 14c shows the speedup of the 3D simulation, and only ADWS (no steal) performed worse than others. This indicates that data locality did not affect the performance (computation-bound) and only load imbalance affected the performance. From these results, ADWS is considered to be robust to load imbalance as well as OpenMP dynamic or random WS, while maintaining good data locality.

## 5.4 Sensitivity to Wrong Estimation of Work

In the heat2D benchmark, the region is divided into four parts with the same size; the work ratio of child tasks in a task group is $1 : 1 : 1 : 1$. In this experiment, we introduced parameter $\alpha$, which meant the maximum error ratio to the correct amount of work. Now we define terms *deterministic wrong estimation* and *random wrong estimation*. In *deterministic wrong estimation*, the work ratio was fixed to $1 - \alpha : 1 - 0.5\alpha : 1 + 0.5\alpha : 1 + \alpha$ at every iteration. In *random wrong estimation*, the work ratio was set to $1+r_1\alpha : 1+r_2\alpha : 1+r_3\alpha : 1+r_4\alpha$, where $r_i$ was an uniform random number in the range $[-1, 1)$ which is newly generated at every iteration. We used the heat2D benchmark to clarify sensitivity to incorrect estimation with $N = 4096$ and with 64 workers, changing parameter $\alpha$.

Figure 16 shows the result. The execution time increased as $\alpha$ increased. Within $\alpha < 10\%$, the performance didn't degrade more than about 30%. Even when $\alpha = 100\%$, some of data locality remained and the performance was better than that of random WS. When the estimation of work was largely wrong, the ADWS

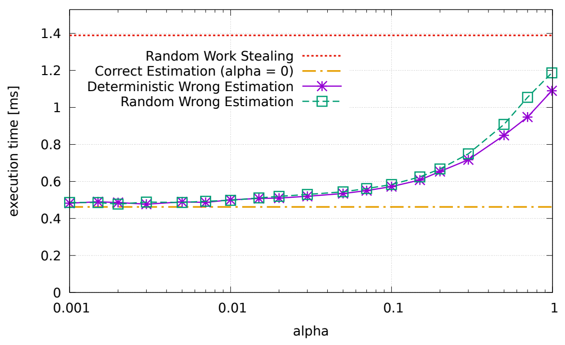


Figure 16: Mean execution time of one iteration in heat2D benchmark for 10k iterations. *alpha* means the maximum error ratio of an estimated amount of work for each task.

Table 2: Mean execution time of fib benchmark for 10 times

| | $n = 20$ | $n = 40$ | $n = 50$ |
|---|---|---|---|
| Random WS | 335 $\mu$s | 265 ms | 32.5 s |
| ADWS | 211 $\mu$s (-37%) | 289 ms (+9.1%) | 35.5 s (+9.2%) |

scheduler turned into random work stealing scheduler quickly because of the management of steal ranges, and there were few additional overheads over random work stealing (recall that the return migration of search-root tasks does not occur under random work stealing state in ADWS).

## 5.5 Overhead Measurement on Fib Benchmark

The fib benchmark calculates a Fibonacci number ($fib(n) = fib(n-1) + fib(n-2)$), which is parallelized by spawning $fib(n-1)$ and $fib(n-2)$ as tasks. Tasks in the fib benchmark have very little computation, so it is often used to measure task scheduler overhead. We used 64 cores with `numactl -iall` to measure the overhead of ADWS relative to random WS. We specified 2 and 1 as a rough estimated amount of work for $fib(n-1)$ and $fib(n-2)$. The last task ($fib(n-2)$) was not spawned as a task to reduce the overhead.

The result with 64 cores with `numactl -iall` is shown in Table 2. ADWS has only about 9% overhead compared with random WS when sufficient parallelism exists ($n = 40, 50$). Surprisingly, when the size of the computation was small ($n = 20$), ADWS was quite faster than random WS. This was because the search was done quickly in ADWS (as discussed in Section 4.3.1). In random WS, the initial distribution of tasks was relatively slow because of lock contention.

## 6 CONCLUSION AND FUTURE WORK

We introduced Almost Deterministic Work Stealing (ADWS) and showed ADWS outperformed other existing scheduling methods with memory-bound applications. Although the applications of ADWS are limited to cases in which programmers can specify the amount of work for each task in advance, performance improved significantly if the amount of work was correctly specified. Specifying the amount of work is certainly a burden on programmers, but it is not hardware-specific and the programming model is entirely processor-oblivious. Even a rough estimate enables hierarchical localized work stealing to dynamically compensate load imbalance in a locality-aware manner.

As future work, we consider it is possible to automatically estimate the amount of work for each task in iterative programs. By using this runtime approach, it is also expected that we can get more precise estimates of the amount of work than ones specified by programmers. Furthermore, because ADWS is compatible with arbitrary memory hierarchies because of its design, we expect to prove that ADWS efficiently runs even on distributed environments.

## 7 ACKNOWLEDGMENT

This work is partially supported by a project commissioned by the New Energy and Industrial Technology Development Organization (NEDO) and JSPS KAKENHI Grant Number JP16H01715.

## REFERENCES

[1] Umut A. Acar, Guy E. Blelloch, and Robert D. Blumofe. 2000. The Data Locality of Work Stealing. In *Proceedings of the Twelfth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA '00)*. ACM, New York, NY, USA, 1–12. https://doi.org/10.1145/341800.341801

[2] N. S. Arora, R. D. Blumofe, and C. G. Plaxton. 2001. Thread Scheduling for Multiprogrammed Multiprocessors. *Theory of Computing Systems* 34, 2 (01 Apr 2001), 115–144. https://doi.org/10.1007/s00224-001-0004-z

[3] E. Ayguade, N. Copty, A. Duran, J. Hoeflinger, Y. Lin, F. Massaioli, X. Teruel, P. Unnikrishnan, and G. Zhang. 2009. The Design of OpenMP Tasks. *IEEE Transactions on Parallel and Distributed Systems* 20, 3 (March 2009), 404–418. https://doi.org/10.1109/TPDS.2008.105

[4] Josh Barnes and Piet Hut. 1986. A hierarchical O (N log N) force-calculation algorithm. *nature* 324, 6096 (1986), 446.

[5] Markus Becker and Matthias Teschner. 2007. Weakly Compressible SPH for Free Surface Flows. In *Proceedings of the 2007 ACM SIGGRAPH/Eurographics Symposium on Computer Animation (SCA '07)*. Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, 209–217. http://dl.acm.org/citation.cfm?id=1272690.1272719

[6] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. 1996. Cilk: An Efficient Multi-threaded Runtime System. *J. Parallel and Distrib. Comput.* 37, 1 (1996), 55 – 69. https://doi.org/10.1006/jpdc.1996.0107

[7] Robert D. Blumofe and Charles E. Leiserson. 1999. Scheduling Multithreaded Computations by Work Stealing. *J. ACM* 46, 5 (Sept. 1999), 720–748. https://doi.org/10.1145/324133.324234

[8] B.L. Chamberlain, D. Callahan, and H.P. Zima. 2007. Parallel Programmability and the Chapel Language. *The International Journal of High Performance Computing Applications* 21, 3 (2007), 291–312. https://doi.org/10.1177/1094342007078442

[9] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. 2005. X10: An Object-oriented Approach to Non-uniform Cluster Computing. *SIGPLAN Not.* 40, 10 (Oct. 2005), 519–538. https://doi.org/10.1145/1103845.1094852

[10] Quan Chen, Minyi Guo, and Haibing Guan. 2014. LAWS: Locality-aware Work-stealing for Multi-socket Multi-core Architectures. In *Proceedings of the 28th ACM International Conference on Supercomputing (ICS '14)*. ACM, New York, NY, USA, 3–12. https://doi.org/10.1145/2597652.2597665

[11] Quan Chen, Minyi Guo, and Zhiyi Huang. 2012. CATS: Cache Aware Task-stealing Based on Online Profiling in Multi-socket Multi-core Architectures. In *Proceedings of the 26th ACM International Conference on Supercomputing (ICS '12)*. ACM, New York, NY, USA, 163–172. https://doi.org/10.1145/2304576.2304599

[12] J. Deters, J. Wu, Y. Xu, and I. A. Lee. 2018. A NUMA-Aware Provably-Efficient Task-Parallel Platform Based on the Work-First Principle. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*. 59–70. https://doi.org/10.1109/IISWC.2018.8573486

[13] Andi Drebes, Karine Heydemann, Nathalie Drach, Antoniu Pop, and Albert Cohen. 2014. Topology-Aware and Dependence-Aware Scheduling and Memory Allocation for Task-Parallel Languages. *ACM Trans. Archit. Code Optim.* 11, 3, Article 30 (Aug. 2014), 25 pages. https://doi.org/10.1145/2641764

[14] Andi Drebes, Antoniu Pop, Karine Heydemann, Albert Cohen, and Nathalie Drach. 2016. Scalable Task Parallelism for NUMA: A Uniform Abstraction for Coordinated Scheduling and Memory Management. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation (PACT '16)*. ACM, New York, NY, USA, 125–137. https://doi.org/10.1145/2967938.2967946

[15] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. 1998. The Implementation of the Cilk-5 Multithreaded Language. *SIGPLAN Not.* 33, 5 (May 1998), 212–223. https://doi.org/10.1145/277652.277725

[16] Y. Guo, J. Zhao, V. Cave, and V. Sarkar. 2010. SLAW: A scalable locality-aware adaptive work-stealing scheduler. In *2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS)*. 1–12. https://doi.org/10.1109/IPDPS.2010.5470425

[17] Masaki Iwasawa, Ataru Tanikawa, Natsuki Hosono, Keigo Nitadori, Takayuki Muranushi, and Junichiro Makino. 2016. Implementation and performance of FDPS: a framework for developing parallel particle simulation codes. *Publications of the Astronomical Society of Japan* 68, 4 (06 2016). https://doi.org/10.1093/pasj/psw053 54.

[18] Yu-Kwong Kwok and Ishfaq Ahmad. 1999. Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors. *ACM Comput. Surv.* 31, 4 (Dec. 1999), 406–471. https://doi.org/10.1145/344588.344618

[19] Jonathan Lifflander, Sriram Krishnamoorthy, and Laxmikant V. Kale. 2012. Work Stealing and Persistence-based Load Balancers for Iterative Overdecomposed Applications. In *Proceedings of the 21st International Symposium on High-Performance Parallel and Distributed Computing (HPDC '12)*. ACM, New York, NY, USA, 137–148. https://doi.org/10.1145/2287076.2287103

[20] Jonathan Lifflander, Sriram Krishnamoorthy, and Laxmikant V. Kale. 2013. Steal Tree: Low-overhead Tracing of Work Stealing Schedulers. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*. ACM, New York, NY, USA, 507–518. https://doi.org/10.1145/2491956.2462193

[21] J. Lifflander, S. Krishnamoorthy, and L. V. Kale. 2014. Optimizing Data Locality for Fork/Join Programs Using Constrained Work Stealing. In *SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 857–868. https://doi.org/10.1109/SC.2014.75

[22] Seung-Jai Min, Costin Iancu, and Katherine Yelick. 2011. Hierarchical work stealing on manycore clusters. In *Fifth Conference on Partitioned Global Address Space Programming Models (PGAS11)*, Vol. 625.

[23] Jun Nakashima and Kenjiro Taura. 2014. *MassiveThreads: A Thread Library for High Productivity Languages*. Springer Berlin Heidelberg, Berlin, Heidelberg, 222–238. https://doi.org/10.1007/978-3-662-44471-9_10

[24] OpenMP Architecture Review Board. 2018. OpenMP Application Program Interface Version 5.0. https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf

[25] J. Paudel, O. Tardieu, and J. N. Amaral. 2013. On the Merits of Distributed Work-Stealing on Selective Locality-Aware Tasks. In *2013 42nd International Conference on Parallel Processing*. 100–109. https://doi.org/10.1109/ICPP.2013.19

[26] Antoniu Pop and Albert Cohen. 2013. OpenStream: Expressiveness and Data-Flow Compilation of OpenMP Streaming Programs. *ACM Trans. Archit. Code Optim.* 9, 4, Article Article 53 (Jan. 2013), 25 pages. https://doi.org/10.1145/2400682.2400712

[27] James Reinders. 2007. *Intel threading building blocks: outfitting C++ for multi-core processor parallelism.* O'Reilly Media, Inc.

[28] A. Robison, M. Voss, and A. Kukanov. 2008. Optimization via Reflection on Work Stealing in TBB. In *2008 IEEE International Symposium on Parallel and Distributed Processing*. 1–8. https://doi.org/10.1109/IPDPS.2008.4536188

[29] S. Seo, A. Amer, P. Balaji, C. Bordage, G. Bosilca, A. Brooks, P. Carns, A. Castelló, D. Genet, T. Herault, S. Iwasaki, P. Jindal, L. V. Kalé, S. Krishnamoorthy, J. Lifflander, H. Lu, E. Meneses, M. Snir, Y. Sun, K. Taura, and P. Beckman. 2018. Argobots: A Lightweight Low-Level Threading and Tasking Framework. *IEEE Transactions on Parallel and Distributed Systems* 29, 3 (March 2018), 512–526. https://doi.org/10.1109/TPDS.2017.2766062

[30] Warut Suksompong, Charles E. Leiserson, and Tao B. Schardl. 2016. On the efficiency of localized work stealing. *Inform. Process. Lett.* 116, 2 (2016), 100 – 106. https://doi.org/10.1016/j.ipl.2015.10.002

[31] K. Taura, J. Nakashima, R. Yokota, and N. Maruyama. 2012. A Task Parallel Implementation of Fast Multipole Methods. In *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*. 617–625. https://doi.org/10.1109/SC.Companion.2012.86

[32] Yizhuo Wang, Yang Zhang, Yan Su, Xiaojun Wang, Xu Chen, Weixing Ji, and Feng Shi. 2014. An adaptive and hierarchical task scheduling scheme for multi-core clusters. *Parallel Comput.* 40, 10 (2014), 611 – 627. https://doi.org/10.1016/j.parco.2014.09.012

[33] Yi Guo, R. Barik, R. Raman, and V. Sarkar. 2009. Work-first and help-first scheduling policies for async-finish task parallelism. In *2009 IEEE International Symposium on Parallel Distributed Processing*. 1–12. https://doi.org/10.1109/IPDPS.2009.5161079

[34] Richard M. Yoo, Christopher J. Hughes, Changkyu Kim, Yen-Kuang Chen, and Christos Kozyrakis. 2013. Locality-aware Task Management for Unstructured Parallelism: A Quantitative Limit Study. In *Proceedings of the Twenty-fifth Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '13)*. ACM, New York, NY, USA, 315–325. https://doi.org/10.1145/2486159.2486175

[35] Han Zhao, Quan Chen, Yuxian Qiu, Ming Wu, Yao Shen, Jingwen Leng, Chao Li, and Minyi Guo. 2018. Bandwidth and Locality Aware Task-stealing for Manycore Architectures with Bandwidth-Asymmetric Memory. *ACM Trans. Archit. Code Optim.* 15, 4, Article 55 (Dec. 2018), 26 pages. https://doi.org/10.1145/3291058