修士論文

Master’s Thesis

Efficient Thread Scheduling Strategies for Nested Parallel Programs

（ネストした並列プログラムに対する
効率的なスレッドスケジューリング手法）

椎名 峻平

Shumpei Shiina

学籍番号　48-196423

東京大学 情報理工学系研究科 電子情報学専攻

Department of Information and Communication Engineering
Graduate School of Information Science and Technology
The University of Tokyo

2021.01.28　提出

指導教員　田浦 健次朗　教授

Advisor　Prof. Kenjiro Taura

Abstract

Nested parallelism is a natural and straightforward representation of parallelism. Parallelism of many algorithms (e.g., divide-and-conquer algorithms) are nested by their nature, and invocation of parallel subroutines from parallel programs yields nested parallelism. In order for nested parallel programs to run efficiently, the underlying parallel runtime should efficiently handle nested parallelism; however, it is challenging as massive and fine-grained parallelism can be dynamically created. User-level threading is a promising approach for this purpose because of its extremely lightweight operations and flexible scheduling policy, whereas kernel-level threading is too heavyweight to handle fine-grained parallelism much more than the number of cores. Nevertheless, there are still challenges to efficiently run nested parallel programs over user-level threads in practice.

The first challenge addressed in this thesis is an interoperability issue between existing parallel programs and user-level threads. Many existing parallel programs are finely tuned under the assumption that they can exclusively use all of the machine resources, and thus they sometimes have synchronization mechanisms based on busy loops for the sake of performance. When such parallel programs get nested, threads more than the number of cores can be created. Because most of the today's user-level threads are nonpreemptive, busy loops in each thread can be blocked forever under oversubscription of threads, resulting in a deadlock. To resolve this issue, this thesis investigates preemption techniques for user-level threads as a workaround for preventing deadlocks. The primary result from the evaluation is that preemptive user-level threads proposed in this thesis achieved 27% performance improvement to kernel-level threads in Cholesky decomposition with nested parallelism, in which otherwise a deadlock occurs over nonpreemptive user-level threads.

The second challenge is a data locality issue for scheduling nested parallel computations on modern architectures with deep memory hierarchy. In these days, as the number of cores on a chip increases, the memory hierarchy is getting deeper and more complicated. As existing scheduling strategies cannot fully exploit data locality in nested parallel computations and in deep memory hierarchy, this thesis proposes a novel scheduling algorithm called Almost Deterministic Work Stealing (ADWS). The idea of ADWS involves deterministic thread scheduling to match the hierarchy of nested parallelism with the memory hierarchy in the machine, improving data locality. Furthermore, ADWS is capable of dynamic load balancing for better core utilization, which makes it "almost" deterministic. The evaluation revealed that ADWS was up to nearly six times faster than the traditional work stealing strategy in 5-point stencil computation.

However, it was also found that the performance of ADWS dropped when the total data size was too large to fit into the last level cache. Thus, this thesis further proposes "multi-level scheduling" to extend ADWS to be cache-size-aware for improving performance of ADWS with larger data. The basic idea of multi-level ADWS, an improved version of ADWS, is to recursively apply ADWS to caches at each level rather than just to cores. It guarantees that the data size accessed by a set of threads scheduled under a cache does not exceed the cache capacity, preventing excessive shared cache misses. The detailed evaluation results show that multi-level ADWS achieved 37% performance improvement to the original single-level ADWS in decision tree construction with 2 GB data, reducing many shared cache misses.

Acknowledgments

First of all, I express my deepest appreciation to Professor Kenjiro Taura for his invaluable support. Without his guidance, I would not have been involved in this exciting research field. I have learned much more than textbook knowledge from him, like what research is all about and how to approach a problem. He also gave me a change to join Argonne National Laboratory as an internship student, which became a valuable experience for me.

During my stay at Argonne National Laboratory, I received generous support from my supervisor Pavan Balaji. Especially, he taught me the basics of how to construct a good research paper and helped me revise our paper a lot. It was undoubtedly a precious experience for me. I would like to thank Shintaro Iwasaki, who was my mentor at Argonne and a *senpai* of Taura Laboratory at the same time. He gave me tremendous support for everything, from the research topic to how to live in the U.S. His sound advice leaded our research in the right direction, and I have learned a lot about writing and presentation skills from him. Doing research with him was quite fun and exciting for me.

I am also thankful to staff members at Argonne, including Yanfei Guo, Hui Zhou, Min Si, and Giuseppe Congiu for their support and encouragement. I had a great time with internship students at that time, Shintaro Iwasaki, Rohit Zambre, Kaiming Ouyang, Hengjie Wang, Poornima Nookala, and Shu-Mei Tseng. They are so friendly and talented, and I enjoyed chatting and a lot of activities in the U.S. with them. I would like to thank Gail Pieper for proofreading our paper, which definitely made our paper in good shape.

I also want to express my gratitude to all of the members of Taura Laboratory. Especially, Assistant Professor Shigeyuki Sato showed me a broad range of research topics, which expanded my horizons. Research papers he introduced to me were always interesting and impressed me a lot. During my life in the lab, I spent a great time with Takuya Fukuoka, my colleague since we joined the lab three years ago. We have talked a lot about research, technical things, and daily life drinking alcohol. Also, I deeply appreciate his tremendous effort put into server management in our lab, which helped all of the members of Taura Laboratory without any doubt. Wataru Endo helped me a lot since I was a newcomer to the lab. Especially, it was really fun to develop a useful logging tool (MassiveLogger), which is still helping me investigate performance issues. I had a great time in chatting and having lunch with Yukio Siraichi, and I want to go eat Brazilian BBQ with him again. Christian Helm gave me very useful information about performance analysis, and Tomokazu Higuchi bought me great steak meat using his salary. Advice and comments by Noboru Tanabe at progress report were insightful and beneficial to my research. I also thank Shogo Matsuda, Taisei Takahashi, Takato Hideshima, and Genki Kimura for chatting with me about research and daily things. I would like to thank the secretary Sachie Ikeya for handling a lot of paperworks and encouraging me.

Finally, I would like to sincerely thank my family for supporting me until now.

A part of this thesis related to preemptive user-level threads was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration, in particular its subproject on Scaling OpenMP with LLVM for Exascale performance and portability (SOLLVE). We gratefully acknowledge the computing resources provided and operated by LCRC and JLSE at Argonne National Laboratory.

Contents

List of Figures

List of Tables

List of Algorithms

1. Introduction

1.1. Background and Motivation

Parallelism of many algorithms is nested by their nature. For example, in recursive divide-and-conquer algorithms such as quicksort, a problem is recursively decomposed into smaller subproblems, which can often be computed in parallel; in this way, parallelism is created as the recursion proceeds. Furthermore, nested composition of parallel programs yields nested parallelism. For example, if a parallel program calls a parallelized subroutine provided by a third-party library, such as Basic Linear Algebra Subprograms (BLAS), there is nested parallelism. Such nested parallelism frequently appears in complicated software stacks, where parallel libraries are hierarchically composed. Considering these cases, nested parallelism can be seen as natural and straightforward representation of parallel computation.

Efficiently handling nested parallelism is generally difficult, as the degree of parallelism grows exponentially as the level of nesting increases. At the same time, parallelism gets much more fine-grained, in which case the overheads caused by parallel runtimes largely affect the overall performance of parallel programs. User-level threads are expected to be the key to efficiently handle massive and fine-grained parallelism posed by nested parallel programs. User-level threads are thread contexts that can be created, dynamically switched, and destroyed, totally in user space without the involvement of the kernel. Threading operations of user-level threads are extremely lightweight compared with kernel-level threads because they can avoid heavyweight context switching to kernel space. Also, the scheduling policy of user-level threads can be defined in user space, enabling specializing thread schedulers for specific workloads. The lightweight threading operations and flexible scheduling policy of user-level threads should largely benefit nested parallel programs, but there are still challenges in utilizing user-level threads for nested parallel programs.

The first challenge addressed in this thesis is an interoperability issue between existing parallel programs and user-level threads. Existing parallel programs sometimes have their own thread synchronization schemes based on busy-looping or spinning for the sake of performance, under the assumption that they can exclusively use all the machine resources (e.g., cores). However, when such parallel programs get nested and threads more than the number of cores are created, busy loops in them can be blocked for a long time, waiting for some other threads to reach synchronization points. It can cause a deadlock when running over user-level threads, because most of the user-level threads today are nonpreemptive; threads busy-looping on cores do not get interrupted and evicted, occupying the cores forever. Ideally, it is better to remove such busy loops in existing programs; however, in practice, it requires much effort. Even worse, it is nearly impossible for closed-source software. Alternatively, supporting preemption for user-level threads would resolve this issue, by interrupting busy-looping threads to schedule other runnable threads instead. However, preemption techniques for user-level threads have not been well studied until now, which motivated us to investigate preemption techniques for user-level threads in this thesis.

The second challenge addressed in this thesis is a data locality issue for nested parallel computations on modern architectures with deep memory hierarchy. As intensively studied in many literatures, nested parallel programs often have good data locality properties, even if they are not tuned for specific memory hierarchy. *Cache-oblivious algorithms* [3], [4] are algorithms that are asymptotically optimal for any cache hierarchy, even though they do not explicitly program specific cache hierarchy and cache sizes. Cache oblivious algorithms are based on recursive formulation of algorithms, such as divide-and-conquer. As recursion proceeds, data in subcomputation fits into the cache at some level of recursion, improving cache utilization without explicit cache blocking. Parallel variants of cache oblivious algorithms have been also intensively studied [5], [6], as many divide-and-conquer algorithms are nested parallel by nature. Nevertheless, scheduling parallel cache-oblivious algorithms on modern architectures is challenging, as the number of cores on a chip is increasing and the memory hierarchy is getting deeper and more complicated. This motivated us to investigate locality-aware thread schedulers for nested parallel programs, that will work well on any memory hierarchy.

1.2. Contributions of This Thesis

The contributions of this thesis are twofolds. First, it investigates preemption techniques for user-level threads to efficiently resolve the deadlock issue arising from nested composition of existing parallel programs. It enables many existing parallel programs to run over user-level threads, exploiting lightweight threading operations and flexible scheduling policies of user-level threads. Beyond the deadlock issue, it is also found that preemptive user-level threads are also beneficial to computations other than nested parallel computations; the contributions of this work are not limited to nested parallelism. Second, it investigates locality-aware scheduling algorithms for recursively nested parallelism, targeting for well-structured parallel programs which have limited assumption on specific machine configuration. By designing a highly efficient thread scheduler for nested parallel computations, we can achieve both of high performance and good portability while keeping natural and straightforward representation of algorithms, even without much effort on performance tuning by the application programmer.

To list the contributions about preemptive user-level threads:

1. It explores preemption techniques for user-level thread libraries which can be realized without any modification of modern OSes and proposes a new preemption technique for user-level threads, called *KLT-switching*, to address the *KLT-dependence* issue posed by an existing preemption technique [7]–[9] (*signal-yield*). While KLT-switching is costlier than signal-yield, it enables safer preemption which can be applied to a broader range of programs, which is quite important for real-world usage of preemptive user-level threads.

2. It shows that a naive implementation of preemption timers (which generate timer interrupts periodically) can cause significant performance degradation with a large number of cores and proposes scalable implementations of preemption timers. This optimization will become more important as the number of cores increases.

3. It demonstrates that the performance of highly optimized preemptive user-level threads can be as fast as nonpreemptive user-level threads; the overhead of preemptive user-level threads is less than 1% compared to nonpreemptive user-level threads on modern CPU architectures with a preemption interval of 1 ms (which is reasonably frequent considering preemption intervals of the modern OS are usually in the millisecond range). Although the evaluation shows that the absolute preemption cost of user-level threads is higher than that of kernel-level threads, the cost of explicit threading operations (e.g., thread creation, yield calls) in user-level threads is much faster than that in kernel-level threads. Generally speaking, the

cost of explicit threading operations is much more important, especially when threads are fine-grained, considering the cost of implicit preemption is less than 1%.

4. It shows that nested composition of parallel libraries over nonpreemptive user-level threads can sometimes cause a deadlock and demonstrates that preemptive user-level threads can prevent the deadlock while outperforming kernel-level threads. The evaluation uses Cholesky decomposition picked from SLATE [10], in which two levels of parallelism exist: the outer one is parallelized by task dependencies of submatrix computations, and the inner one is parallelized by a third-party BLAS library (Intel MKL [11]) in each submatrix computation. In this nested parallel computation, a deadlock results from a busy-loop-based barrier implementation in Intel MKL when scheduled nonpreemptively. The evaluation result shows that preemptive user-level threads are up to 27% faster than kernel-level threads, because user-level threads incur much smaller overheads for explicit threading operations than kernel-level threads.

5. It empirically shows that supporting preemption gives more freedom of scheduling to user-level threading libraries, improving the performance of applications under complicated settings close to real-world usage, in which appropriate prioritization of threads is needed to improve core utilization. The evaluation includes HPGMG [12] under *thread packing* [13], in which the number of available cores is dynamically reduced for the purpose of power capping, and LAMMPS [14], [15] with *in situ analysis*, in which prioritization of threads plays a critical role in performance. The empirical result shows that preemptive user-level thread schedulers specialized for each application outperform traditional nonpreemptive user-level threads and the general-purpose kernel-level scheduler, by efficiently prioritizing threads.

To list the contributions about locality-aware scheduling for recursively nested parallelism:

1. It proposes a novel scheduling algorithm, called *Almost Deterministic Work Stealing (ADWS)*, to efficiently map nested parallel computations to machines with deep memory hierarchy. ADWS is designed to map hierarchical computation to hierarchical memory subsystems, respecting data locality in hierarchical parallel computation. The thread mapping is done deterministically based on the programmer's hints on the relative amount of work for each thread. ADWS is also equipped with a dynamic locality-aware load balancing scheme, which enables high core utilization while preserving good data locality.

2. It explains in detail the implementation of ADWS over user-level threads.

3. It evaluates ADWS with other scheduling strategies and shows that ADWS outperforms them in memory-bound applications. It shows that ADWS can be up to nearly six times faster than work stealing [16] in 5-point stencil computation. It is also indicated that ADWS does not perform well when the total data size of computation is larger than the capacity of the last level cache.

4. It proposes "multi-level scheduling" to reduce shared cache misses when the overall data of computation does not fit into the cache. In multi-level scheduling, only subcomputation that fits into level-l cache is scheduled under level-l cache at the same time, reducing shared cache misses. Multi-level scheduling is composed of scheduling between a level-l cache and its child caches at level $l+1$, and subcomputation that fits into level-$l + 1$ cache is recursively scheduled under level-$l + 1$ cache. Any composition of scheduling algorithms at each level of cache can be considered.

5. It empirically demonstrates that *multi-level ADWS*, a variant of multi-level scheduling where each level is scheduled by ADWS, outperforms other scheduling strategies in many benchmarks with various data

sizes. For example, compared with (single-level) ADWS, multi-level ADWS achieves 37% performance improvement in decision tree construction with 2 GB data, which is much larger than the last level cache size. It also achieves 16% performance improvement to multi-level WS, a multi-level variant of work stealing.

1.3. Outline of This Thesis

This thesis is organized as follows.

- Chapter 2 explains the background knowledge for this thesis, including the basics of parallel computation, various threading techniques including the current situation of preemption supports for user-level threads, and schedulers for nested parallel computations.
- Chapter 3 explores preemption techniques for user-level threads. This chapter includes the proposal of KLT-switching, optimizations for preemption techniques, and evaluations of the effectiveness of preemptive user-level threads.
- Chapter 4 proposes Almost Deterministic Work Stealing (ADWS) and explains its design and implementation. Evaluation of various schedulers with multiple applications follows.
- Chapter 5 extends ADWS to be cache-size-aware and compares the improved version of ADWS (multi-level ADWS) with the original ADWS (single-level ADWS) and other schedulers.
- Chapter 6 lists existing studies regarding preemptive user-level threads and task schedulers for nested parallel computations, together with discussion of the difference to the approaches proposed in this thesis.
- Chapter 7 summarizes this thesis and gives concluding remarks and future insights.

2. Background

2.1. Flat Parallelism and Nested Parallelism

2.1.1. Basics of Parallel Computing

Modern processors have multiple levels of parallelism, including instruction-level parallelism, single-instruction-multiple-data (SIMD) instructions, core-level parallelism, and node-level parallelism. This thesis focuses on core-level parallelism, in which multiple cores within a node are utilized for increasing the throughput of computation. In order to exploit core-level parallelism, programs have to expose parallelism to the hardware in some way. Programming models for core-level parallelism can be classified into two models: distributed memory model and shared memory model.

In distributed memory model, parallelism is typically exposed as processes, which do not share virtual memory addresses with each other. Message Passing Interface (MPI) is one of the most major standards for parallel computing in the distributed memory model, which provides a set of APIs to exchange data among processes. Programs based on distributed memory model are easy to exploit node-level parallelism too, but distributed memory programming is generally hard and sometimes inefficient for core-level parallelism because of the restriction of address sharing.

In shared memory model, parallelism is typically exposed as threads, which share the same virtual memory address within a process. Thus, threads within a process can simultaneously access to the same memory location. Because of address sharing, the programmer does not need to write explicit data exchange as in the distributed memory model, while it sometimes requires a special care for avoiding race condition, where simultaneous accesses to the same memory location by multiple threads result in an unexpected state. Shared memory programming with threads is the primary focus of this thesis.

The application programmer can utilize threads either directly, by using low-level APIs provided by the OS or threading libraries, or indirectly, by letting the parallel runtime map parallelism to threads. In the latter case, typically, the programmer exposes parallelism to the parallel runtime using a convenient and high-level notation, and the parallel runtime efficiently handles the parallelism by using low-level APIs for threads. OpenMP [17] is an example of high-level parallel programming models, which is explained in the next section, and Pthreads [18] define low-level threading APIs which we will cover in Section 2.2. The following sections explain how the programmer exposes parallelism to the parallel runtime, using some examples of widely-used parallel programming models.

2.1.2. Flat Parallelism

The most dominant programming style in the shared memory model is to expose flat parallelism to a parallel runtime. For example, OpenMP [17], one of the most major standards for shared memory computing, provides parallel loop constructs to express flat parallelism. Fig. 2.1 shows an example code to calculate addition of vector *a* and *b* and store the result in vector *c* in parallel using OpenMP. The directive `#pragma omp parallel for`

```
#pragma omp parallel for
for (int i = 0; i < N; i++) {
  c[i] = a[i] + b[i];
}
```

Fig. 2.1.: Computation of vector addition using a parallel loop in OpenMP.

```
node* tree = /* The root node of a tree */;
vector<node*> leafs; /* Array of leaf nodes to be computed by 'create_leaf_array' function */

void create_leaf_array(node* t) {
  if (t->is_leaf) {
    leafs.push_back(t);
  } else {
    for (node* child : t->children) {
      create_leaf_array(child);
    }
  }
}

create_leaf_array(tree);

#pragma omp parallel for
for (node* leaf : leafs) {
  computation_for_leaf(leaf);
}
```

Fig. 2.2.: Example of parallel computation for each leaf node of a tree using flat parallelism.

tells the compiler that the following loop can be computed in parallel; i.e., each iteration of the parallel loop can be computed in parallel. Typically, the OpenMP runtime creates as many threads as cores at the entrance of a parallel loop, and each thread computes over an iteration space assigned to itself. In OpenMP, the programmer can specify how the entire iteration space of the parallel loop is divided and assigned to each thread by `schedule` clause. For example, in the *static* schedule, each thread is assigned an equally-sized iteration space statically divided, whereas in the *dynamic* schedule, the iteration space is divided into a set of chunks, which are dynamically assigned to each thread on demand.

A parallel loop is a convenient parallel construct to express flat parallelism, as it is easy to parallelize an existing serial program just by replacing a loop with a parallel loop (putting a directive in OpenMP). However, a parallel loop is sometimes inflexible to express more irregular parallelism, such as parallelism based on tree structures and divide-and-conquer algorithms. Even in those cases, in practice, programmers use flat parallelism by transforming the algorithm to create a parallel loop, which often requires much effort and makes the program hard to understand. Fig. 2.2 is an example of the increased complexity in the program by transforming a tree structure into flat parallelism. In this example, suppose that we need to do some computation for each leaf node of a tree, which can be computed in parallel. In order to express this parallelism by parallel loops, we first need to collect leaf nodes into a single array (`leafs`) by traversing the tree (`create_leaf_array()` function). Then, we can apply a parallel loop construct into the array. In the next section, we will see that this parallel program is much simplified by introducing nested parallel constructs.

```
node* tree = /* The root node of a tree */;

void compute_tree(node* t) {
  if (t->is_leaf) {
    computation_for_leaf(leaf);
  } else {
    for (node* child : t->children) {
      spawn compute_tree(child);
    }
    sync;
  }
}

compute_tree(tree);
```

Fig. 2.3.: Example of parallel computation for each leaf node of a tree using recursively nested parallelism.

```
quicksort(data) {
  if (data.size() <= 1) {
    return data;
  } else {
    pivot = select_pivot(data);
    (lo_elems, hi_elems) = partition(data, pivot);
    spawn lo_sorted = quicksort(lo_elems);
    spawn hi_sorted = quicksort(hi_elems);
    sync;
    return lo_sorted ++ [pivot] ++ hi_sorted;
  }
}
```

Fig. 2.4.: Pseudocode of quicksort using recursively nested parallelism.

2.1.3. Recursively Nested Parallelism

Parallelism of many algorithms, such as computations for a tree and divide-and-conquer algorithms, is recursively nested by their nature. To expose recursively nested parallelism in programs, the *fork-join model* is often used. A *fork* operation (often denoted as *spawn*) creates a parallelism, and a *join* (often denoted as *sync*) operation waits for the completion of forked parallel computations. Fig. 2.3 is the nested parallel version of Fig. 2.2. Parallelism is recursively created by spawn keyword at each level of recursion, and sync waits for the completion of all parallel computations spawned at this level of recursion. In this way, computations for leaf nodes are straightforwardly parallelized by traversing the tree while specifying computations for each child node as parallel. It does not require allocation for an additional array of leaf nodes (leafs in Fig. 2.2), reducing the burden on the programmer and simplifying the program.

We give another example of recursively nested parallelism in Fig. 2.4, which is pseudocode of quicksort parallelized by nested fork-join constructs. Quicksort is a well-known divide-and-conquer algorithm, in which input elements are partitioned into ones smaller than the pivot (lo_elems) and ones larger than the pivot (hi_elems), each group is sorted recursively, and the results of them (lo_sorted and hi_sorted) are concatenated. Sorting for lo_elems and hi_elems can be done in parallel since they are not overlapped. Fig. 2.4 is natural and straightforward parallelization of quicksort; the basic code structure is not changed from the serial code. We note that it is not trivial to express divide-and-conquer algorithms such as quicksort only with flat parallelism.

2.1.4. Nested Composition of Parallel Programs

Nested parallelism comes not only from algorithms themselves but also from multi-layered software stacks. If a parallel program calls another parallel program within a parallel code block, there is nested parallelism, even when each program exposes flat parallelism. The issue of composition of parallel programs was also pointed out and studied by some literatures [19]–[21].

In practice, many applications abandons inner levels of parallelism to make it flat; only the outer level of the parallelism is exposed to the parallel runtime. This is for the sake of performance; nested parallelism is often inefficient mainly because of the implementations of existing parallel runtimes and the assumption of exclusiveness in existing programs. Existing programs often assume that they can exclusively use all the resources in the machine when executed in parallel, resulting in contention for the machine resources when inner parallelism is enabled.

However, abandoning inner parallelism is problematic when outer level parallelism is insufficient. For example, in a program which calls a parallel subroutine at a shallow level of parallel recursion, there is not enough parallelism in the outer parallelism (recursion). Without using the inner level of parallelism (parallel subroutine), the program suffers from lack of parallelism and performs poorly. We can also use only the inner parallelism and abandon the outer parallelism, but even in this case, at deep level of recursion, the problem size is often small, resulting in poor parallelism in a parallel subroutine. In these kind of algorithms, utilizing nested parallelism is becoming important today, as the number of cores increases.

Also, from the viewpoint of productivity, collapsing nested parallelism into flat parallelism is not desirable. Considering the case where the programmer uses a subroutine provided by a third-party library, the implementation of the library, including whether it is parallelized or not, should be a black box to the user. Thus, the current practice of explicit parallelism management in external libraries is suboptimal.

2.2. Threading Techniques

2.2.1. Basics of Threads

A thread is *a single flow of control within a process* according to the definition in POSIX [18]. A thread has an execution context to save the current call stack and registers (such as an instruction pointer) to be interrupted and resumed at any time.

Multiple threads concurrently run within a process, enabling latency hiding and high throughput. Latency hiding is a technique to mitigate the idle CPU time while waiting for blocking operations, such as I/O operations, by doing useful work during the wait time. For example, when a thread issues an I/O request and gets blocked, the thread is evicted from the core so that another ready thread can be scheduled to the same core, doing some meaningful work until the I/O request is completed and the issuing thread is unblocked. On multicore processors, multiple threads can be simultaneously executed by multiple cores, achieving high throughput of computation. In parallel computing, threads are often used to exploit core-level parallelism.

Users can in principle create an arbitrary number of threads, even more than the number of cores. A thread scheduler is responsible for dynamically mapping threads to cores, which largely affects the overall performance of multithreaded programs. Since it is generally hard to design a general thread scheduler that will efficiently work for every workload, many thread schedulers have been proposed in various contexts. The purpose of thread scheduling often differs in different contexts; for example, in real-time systems, delay of high-priority threads is crucial for their performance, whereas in high-performance computing, throughput of computation is more important. Thread schedulers designed for recursively nested parallelism, which are the main focus of this thesis, are described in Section 2.3 in detail.

In the following, we introduce various low-level implementations of threads, which are the basis of shared memory parallel computing.

2.2.2. Kernel-Level Threads

Kernel-level threads are thread implementations provided by the kernel. Sometimes kernel-level threads are referred to as *lightweight processes*, as historically threads are introduced to OSes as process-like scheduling entities that share the same virtual address space. This thesis defines a kernel-level thread as a scheduling entity managed by the kernel and a process as a resource group which can have multiple kernel-level threads sharing a single virtual address space. Thus, when a program is launched, it is assigned a process with one kernel-level thread on which the program runs. After that, new kernel-level threads have to be allocated within the process to utilize multiple cores. In other words, in order for a process to utilize N cores, it has to have at least N kernel-level threads.

Kernel-level threads are scheduled by the kernel. Kernel-level threads may be interrupted by the kernel at any point in time and evicted from the core; this type of scheduling is called *preemptive scheduling*, which is described in Section 2.2.4 in more detail. Under preemptive scheduling, the program is usually unaware of preemption; even if there are kernel-level threads more than the number of cores, the kernel-level threads appear to run simultaneously (at least under a fair scheduling policy). Thus, kernel-level threads can be seen as virtualized cores.

POSIX threads (Pthreads) [18] are one of the most prevalent threading standards. Although the Pthreads specification does not require threads be implemented as kernel-level threads, most of the today's Pthreads implementations use kernel-level threads as Pthreads. This threading model is often called 1:1 model, as a thread visible from the user are mapped to a kernel-level thread. In the following of this thesis, we assume that Pthreads are implemented as kernel-level threads (based on 1:1 mapping).

One of the benefits of kernel-level threads is that kernel-level threads can be aware of kernel events, such as I/O blocking and page faults. For example, when a kernel-level thread is blocked by a system call that requires I/O, the kernel-level thread can be immediately evicted from the core so that another thread is scheduled on the core. This is, however, impossible for user-level threads, as discussed in Section 2.2.3.

Downsides of kernel-level threads are their large overheads of threading operations and inflexible scheduling policy. The scheduler of kernel-level threads is implemented in the kernel; thus threading operations, including thread creation, context switching, and yielding, require involvement of the kernel, which often incurs large overheads because of context switching costs to the kernel and sometimes resource contention in the kernel. The kernel-level scheduling policy is usually not customizable; the user cannot control thread scheduling even within a process, although some parameters may be controllable from user space.

2.2.3. User-Level Threads

User-level threads have been proposed as lightweight threads that are implemented mostly with user-space operations. User-level threads are dynamically mapped to kernel-level threads; typically as many kernel-level threads as cores are allocated at initialization, and many user-level threads are switched around the kernel-level threads. This threading model is often called M:N model, as M user-level threads are mapped to N kernel-level threads (typically $M > N$). In Section 3, we clearly distinguish user-level threads and M:N threads; user-level threads are threads visible to the user and M:N threads are thread implementations based on many-to-many mapping from user-level threads to kernel-level threads. Following this definition, even widely-prevalent Pthreads implementations based on the 1:1 model can be called user-level threads, although Pthreads are essentially kernel-level threads. Throughout of this thesis, however, we sometimes use the term ”user-level thread“ to represent M:N threads depending on the context, following the convention in many literatures.

As with kernel-level threads, a user-level thread have its own stack. In a user-level thread implementation, when a thread context switches, the running thread's stack pointer and registers are saved in a separate memory location, and the control is handed back to the scheduler. The scheduler then picks another thread to be scheduled (depending on scheduling policy it uses), loads the new thread's stack pointer and registers, and executes it—an approach that costs only about one hundred cycles in total. User-level context switching is key to efficiently implementing several threading operations including fork, join, and yield. Users can develop their own schedulers, thus allowing them to customize their scheduling algorithm for specific workloads.

While numerous user-level thread implementations exist, this thesis assumes the following threading model, which is common to most such implementations. On initialization, the runtime creates as many workers as the available cores, and each worker is associated with one kernel-level thread. Each worker runs a “scheduler thread” (which is also a user-level thread) on the kernel-level thread associated with it at scheduling points. The scheduler thread runs an infinite loop that tries to pop a thread from its associated *thread pools* that store ready threads and runs it by context switching. When a thread explicitly yields or finishes its thread function, it switches to the scheduler thread by context switching again.

Some parallel runtimes also provide user-level-thread-like entities, called run-to-completion threads or “tasks,” but these entities do not support generic yield; tasks in Cilk [22]–[24] and Intel TBB [25] have no concept of yielding while tasks in LLVM/Intel OpenMP support *stack-based yield*, which has highly restrictive scheduling constraints [26]. This thesis does not deal with tasks but only with threads, although the ideas of the proposed schedulers for recursively nested parallelism in Chapter 4 and Chapter 5 can also be applicable to tasks.

In contrast to kernel-level threads, user-level threads are not aware of kernel events, such as blocking I/O, page faults, and preemption, because most of the today's OSes do not notify such events to user space as discussed in Section 6.1. As most OSes do not explicitly support user-level threads, user-level threads may waste a large amount of time while threads are effectively blocked by the kernel, hindering replacement of the threading layer in existing multithreaded programs with user-level threads.

Traditionally, *cooperative scheduling* or *nonpreemptive scheduling*, where threads are not evicted from cores unless threads voluntarily yields their cores, has been considered a fundamental feature of user-level threads. This is because most user-level thread implementations have not provided preemption until now, as explained in the next section.

2.2.4. Preemption

Kernel-level threads usually adopt both explicit and implicit yielding capabilities. In a kernel-level thread implementation, threads can explicitly call a yield function (such as `sched_yield()`) or use implicit OS preemption if a thread does not yield for a “long time” (an OS scheduler time slice). In either case, control is handed back to the scheduler, which can then pick the next thread to schedule based on priority or fairness metrics. On the other hand, most user-level threads are not preemptive. Context switching between threads happens only at explicit scheduling points, namely, when a thread explicitly calls yield or yield-like operations (e.g., fork, join, or barrier) or when it completes. Thus, if threads occupy cores for a long time without issuing explicit scheduling operations, nonpreemptive user-level threads can cause core starvation, loss of prioritization, and deadlock.

Most user-level threads are nonpreemptive since they cannot count on the OS preemption functionality, which utilizes hardware timer interruption, because user-level threads are not visible from the OS. To realize preemption for user-level threads, we have to control interruption to user-level threads from user space, using some kernel interfaces provided by the OS. In Section 3, we describe how to interrupt the execution of threads without kernel modification.

2.3. Schedulers for Recursively Nested Parallelism

2.3.1. Nested Parallel Computations and Task Schedulers

Section 2.1.3 introduced recursively nested parallel computations, such as tree computations and divide-and-conquer algorithms (e.g., quicksort). Fig. 2.5 is a visualization of such a nested parallel computation in the form of a directed acyclic graph (DAG), where nodes are units of serial computations (often called "tasks") and edges represent dependencies between tasks. Tasks in the DAG can be computed in parallel as long as their incoming dependencies are met. In the following of this thesis, we assume that a spawned task is placed on the left side of figures and the continuation of the parent task is placed on the right. Parallelism expressed in the form of a DAG is often called "task parallelism" in general. Recursively nested parallelism is a subset of task parallelism; task parallelism can deal with any parallelism as long as its dependencies can be expressed in the form of a DAG. However, it is often hard for the general task parallel model to exploit the data locality properties in the DAG as explained later; this thesis focuses only on recursively nested parallelism.

A task scheduler is an underlying runtime component that maps tasks in the DAG to cores on physical hardwares, maintaining dependencies between tasks. The DAG is usually not known in advance; the structure of the DAG is revealed dynamically (online scheduling). Sometimes the shape of the DAG can be determined in advance (offline scheduling), but it is not the case for irregular computations. For example, the shape of the DAG of quicksort is input-dependent and cannot be determined until execution. This thesis focuses on online scheduling, which can deal with a broader range of parallel computations.

Because the shape of the DAG is not known in advance, the task scheduler should make a decision on how to map tasks to cores on the fly as quickly as possible. Generally speaking, it is difficult to create an optimal scheduler for arbitrary workloads; there are several properties to evaluate task schedulers. First, the overhead of the task scheduler itself should be minimized. Second, load balancing among cores is important; if there is a ready task and an idle core, the task should be executed on the idle core (greedy scheduler). Third, the scheduler should be aware of data locality in a DAG. The property of data locality is the main focus of this thesis; respecting data locality is practically important for many applications because many today's applications are memory-bound and the memory hierarchy on modern architectures is getting deeper and deeper. The next section explains the data locality properties in nested parallel computations in detail.

2.3.2. Data Locality in Nested Parallel Computations

Roughly speaking, close tasks in a DAG share some data locality. Recall the quicksort example and its divide-and-conquer approach. As the input array is partitioned and child tasks compute on a smaller portion of the data, close tasks in the DAG access close (or often overlapping) data. If close tasks are scheduled on distant processors, the cost of data movement across processors becomes large, worsening the performance. To exploit

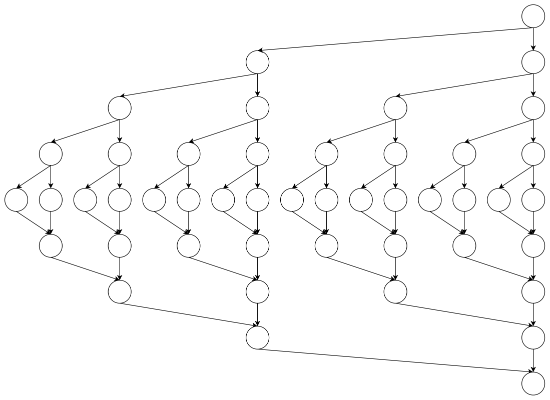


Fig. 2.5.: Directed acyclic graph (DAG) of a nested parallel computation.

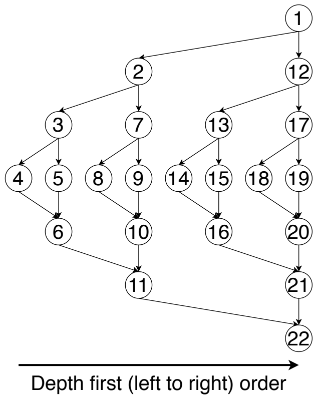


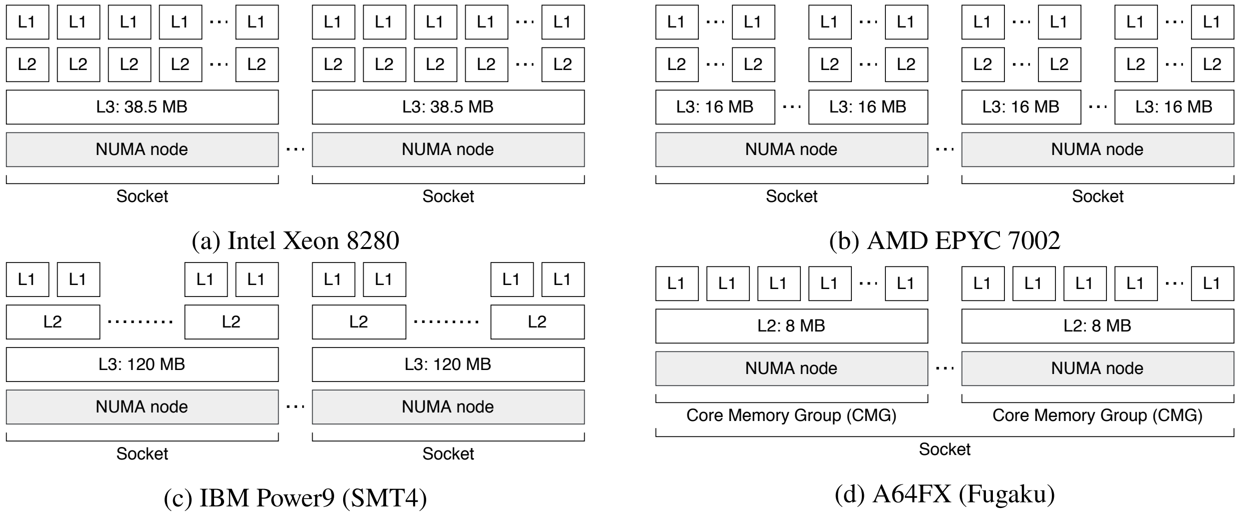
Fig. 2.6.: Depth-first execution order of a nested parallel computation.

data locality in nested parallelism, the depth-first execution order is generally considered to be a good execution order. Fig. 2.6 shows the depth-first execution order in a DAG. It is the same execution order as serial execution by one core, because in the serial execution the function call is executed first and the continuation is executed later. Since function calls (spawned tasks) are placed on the left and continuations are on the right in the figure, this order is sometimes called "left-to-right" execution order. This execution order is good for nested parallel programs because it can promote cache reuse across close tasks.

The data locality properties of recursive algorithms have been formally studied in many literatures. *Cache-oblivious algorithms* [3], [4] are algorithms that do not require cache hierarchy or cache sizes be programmed explicitly but perform optimally on any cache hierarchy. Informally, recursive algorithms tend to have cache-obliviousness. As recursion proceeds, a problem is decomposed into smaller subproblems, and at some level of recursion, a subproblem fits into a cache; data access to the descendants of the subproblem are all cache hit. Since this applies to any level of cache in the cache hierarchy, cache-oblivious algorithms perform well on any cache hierarchy without assuming specific cache hierarchy. Cache-oblivious algorithms have advantages over cache-aware algorithms, in which specific cache hierarchy and cache sizes are explicitly programmed for the sake of performance (e.g., cache blocking). First, cache-oblivious algorithms are simple and straightforward to implement, reducing the burden on the programmer. Also, they have good portability; cache-obliviousness largely reduces the burden on tuning for specific architectures. As multicore processors become prevalent, parallel variants of cache-oblivious algorithms have been also studied [5], [6]. Most of them are based on recursively nested parallelism, following the divide-and-conquer approach. For parallel cache-oblivious algorithms to perform well on multicore processors, it is crucially important for the task scheduler to respect the data locality in the nested parallel DAG.

Also, on the hardware side, the demand for good data locality is increasing. Modern architectures tend to have deep memory hierarchy for exploiting data locality in programs. As the number of cores increases, the memory subsystem is expected to be more hierarchical and deeper. However, it requires parallel programs to utilize deep memory hierarchies for achieving good performance, increasing the burden on the programmer. Even worse, the structure of memory hierarchy is various across different architectures as shown in Fig. 2.7; portability of parallel programs is hard to achieve today.

The memory hierarchy of modern architectures can be divided into two parts: hierarchical cache and Non-Uniform Memory Access (NUMA) nodes. Modern CPUs have multiple levels of cache (typically up to L3 cache). Cache is either private or shared, depending on whether it is shared by multiple cores or not. We can



(a) Intel Xeon 8280

(b) AMD EPYC 7002

(c) IBM Power9 (SMT4)

(d) A64FX (Fugaku)

Fig. 2.7.: Various memory hierarchies on modern architectures.

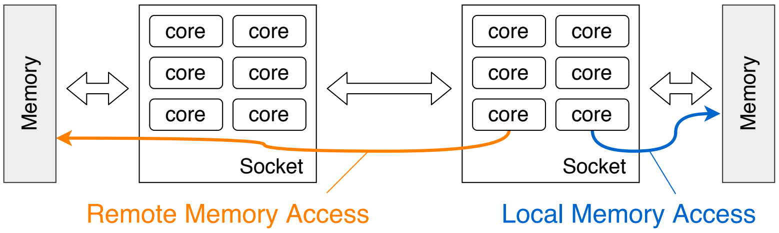


Fig. 2.8.: Illustration of Non-Uniform Memory Access (NUMA).

have multiple sockets (CPU chips) in a single node, deepening the cache hierarchy. In the parallel memory hierarchy model [27], the cache hierarchy is modeled as a tree of caches, in which the main memory (DRAM) is the root node.

Today, the main memory is often composed of physically distributed DRAMs with different access costs; this configuration is called Non-Uniform Memory Access (NUMA). Fig. 2.8 illustrates NUMA architectures and different memory access costs. DRAMs are physically distributed to multiple NUMA nodes; the memory access cost is different depending on which memory stores the target data. For example, if the target data is stored on the local NUMA node which the core belongs to, the memory access is called “local memory access” and has relatively low cost. On the other hand, if the data is stored on a remote NUMA node, the memory access is called “remote memory access” and has relatively high cost. Therefore it is important to increase the number of local memory accesses to utilize data locality on NUMA architectures. Unlike hardware cache, data on NUMA nodes are fixed to the physical memory and cannot be automatically moved to different NUMA nodes by hardware. For the task scheduler to perform well on arbitrary memory hierarchy, it must handle both of the different characteristics of caches and NUMA.

We must note that there is also data locality among tasks that are not close in a DAG for iterative parallel programs, as pointed out in [28]. An iterative parallel program is a program that repeats the same or similar parallel computation for many times. An example of iterative programs is computational simulation which calculates time evolution of a physical system step by step. Fig. 2.9 illustrates data locality of iterative parallel programs. Tasks enclosed in the dotted lines share data locality. As iterative programs usually sweep data in the same order at each iteration, data locality exists vertically in the DAG. In other words, the task at the

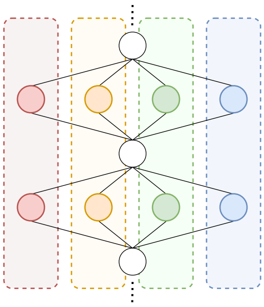
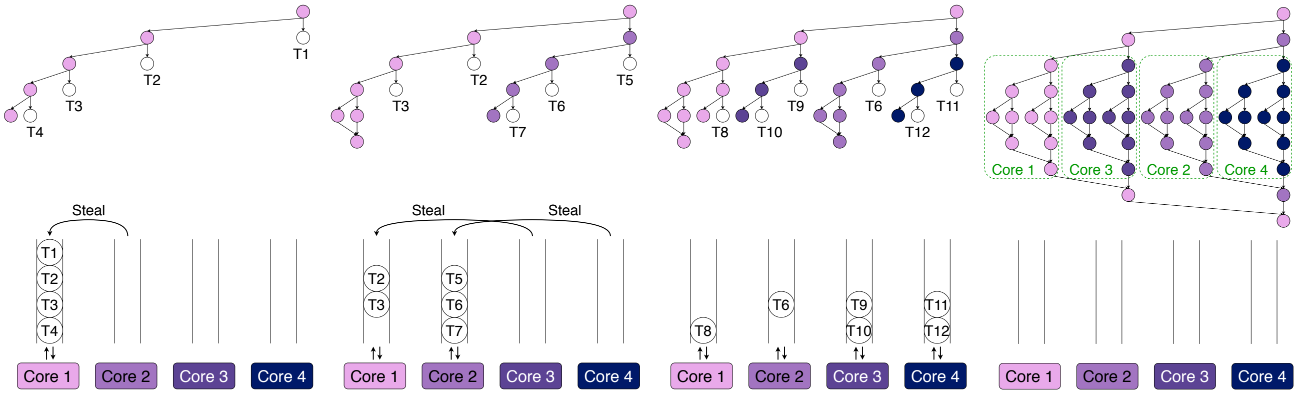


Fig. 2.9.: Data Locality in Iterative Parallel Programs.



(a) Core 1 executes tasks in depth first order. Generated tasks are pushed to its local queue.

(b) Core 2 steals the oldest task created in Core 1. Core 1 pops a task from its local queue in LIFO order.

(c) Core 3 and 4 steals tasks from Core 1 and 2, respectively. They execute tasks locally.

(d) Cores execute as distant tasks as possible. Task execution order within a core is depth-first order.

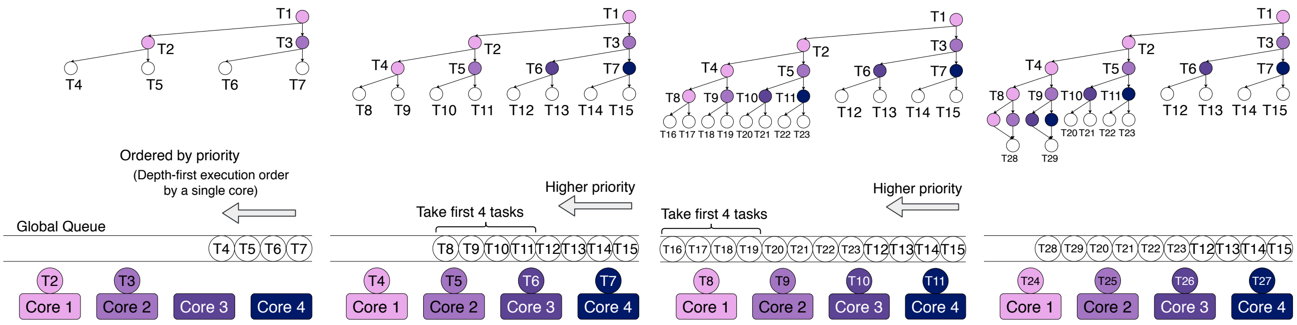
Fig. 2.10.: Illustration of work stealing.

same location as in the previous iteration tends to touch the same data. Exploiting this type of data locality can promote cache reuse across iterations. Furthermore, in NUMA architectures, if it is known in advance which part of the data is accessed by which NUMA node at every iteration, data can be efficiently distributed to appropriate NUMA nodes to increase local memory accesses. Deterministic scheduling (such as OpenMP's `static` schedule) can easily exploit this type of data locality, but in nested parallelism it is not obvious. Later, this thesis introduces an *almost* deterministic approach for scheduling nested parallel programs in Chapter 4.

To address data locality in task parallel programs, two major task scheduling algorithms have been proposed in the 1990s: *work stealing* [16] and *parallel depth first (PDF) scheduler* [29]. Work stealing has good data locality for private caches, whereas PDF scheduler has good data locality for a shared cache. Because of their efficiency, these schedulers have been adopted by many task parallel runtimes, but their scheduling is not optimal for deep memory hierarchies on modern hardwares. The following sections explain the working of work stealing and PDF scheduler as the basis of task scheduling, as well as the difficulty in exploiting data locality for arbitrary memory hierarchy.

2.3.3. Work Stealing

This section introduces *work stealing* [16], a widely-used task scheduling algorithm. Fig. 2.10 illustrates how work stealing works on four cores. In work stealing, each core has its own local task queue. Basically, a core



(a) A global queue is shared by all cores. Each task in the queue is prioritized by the depth-first execution order.

(b) High priority tasks are popped from the queue and assigned to cores.

(c) Since the tasks are prioritized by the depth-first execution order, the tasks are executed in the depth-first order.

(d) Close tasks in the DAG are executed at a time by all cores in parallel.

Fig. 2.11.: Illustration of PDF scheduling.

executes tasks in the depth-first order, pushing the continuations into the local queue in Last-In First-Out (LIFO) order. Task execution order within a core is depth-first order because, when the executing task completes or suspends, a core tries to pop a task from the local queue in LIFO order. An idle core tries to steal a task from another core's queue by choosing a victim randomly. It steals a task in First-In First-Out (FIFO) order, which means that the stolen task is the oldest task generated in the victim core. This is a good characteristic in work stealing because the oldest task in the queue locates at shallow parts in the DAG and is expected to have larger amount of computation. Thus, in work stealing, the granularity of steals is large, improving data locality within a core.

Since cores execute as distant tasks as possible and execute tasks in the depth-first order within a core, data locality in private caches is good in work stealing. In other words, the amount of communication among cores is small and cores execute tasks as locally as possible; additional communication happens only when stealing. Acer et al. **locws** gave a theoretical bound on the number of additional private cache misses in work stealing as $O(C\tau)$, where C is a private cache size and τ is the number of steals. The number of steals is also bounded by $O(PT_\infty)$, where P is the number of processors and T_∞ is the critical path of the DAG; therefore the number of additional cache misses is bounded by $O(CPT_\infty)$. It theoretically shows that data locality of work stealing is good for private caches.

On the other hand, data locality of work stealing in a shared cache is not good. Roughly speaking, because work stealing executes the entire DAG at a time, shared cache misses frequently occur when the total data size of the DAG is larger than the shared cache size. More strictly, space requirement is a good property to explain the shared cache behaviour. Blumofe et al. [16] bounded the space requirement of work stealing by $O(S_1P)$, where S_1 is the space requirement by the serial execution by one core and P is the number of cores. This indicates that work stealing consumes P times more memory than the serial execution, oppressing the limited size of a shared cache.

2.3.4. Parallel Depth First (PDF) Scheduler

Parallel depth first (PDF) scheduler [29] has good data locality for a shared cache. Fig. 2.11 explains how PDF scheduler works. In PDF scheduling, ready tasks are maintained in the global queue and prioritized by the depth-first execution order. At each step, higher priority tasks are picked from the global queue and assigned to each core. Since the ready tasks in the queue are prioritized by the global depth-first order, the execution order by P cores also follows the depth-first execution order. Roughly speaking, since PDF scheduler executes close tasks at a time by all cores, data locality for the cache shared by all cores is good.

Blelloch et al. [29] bounded space requirement by PDF scheduler by $S_1 + O(PT_\infty)$, which is usually less than that of work stealing. Blelloch et al. [30] also stated that the number of shared cache misses by PDF scheduler is theoretically the same as that by the serial execution if the shared cache size is more than $C_1 + O(PT_\infty)$, where C_1 is the shared cache size used by the serial execution. This theoretically verifies the optimality of PDF scheduler on a cache shared by all cores, but PDF scheduler cannot handle data locality in private caches effectively. This is because, in PDF scheduler, tasks executed within a core can be distant and close tasks are distributed to multiple cores, increasing private cache misses.

Summarizing, work stealing works well on private caches but not on a shared cache. On the other hand, PDF scheduler works well on a shared cache but not on private caches. Therefore, both scheduling algorithms are not expected to work well on architectures that have both private and shared cache. To make things worse, modern architectures tend to have more complex and deeper memory hierarchy, demanding smart task schedulers that perform well on any memory hierarchy.

3. Lightweight Preemptive User-Level Threads

This chapter investigates preemption techniques for user-level threads (specifically, M:N threads) primarily for resolving the deadlock issue arising from nested composition of parallel programs. The content of this chapter is based on the conference paper [2] written with Shintaro Iwasaki, Kenjiro Taura, and Pavan balaji. Some additional explanation and evaluation are included in this thesis.

3.1. Introduction

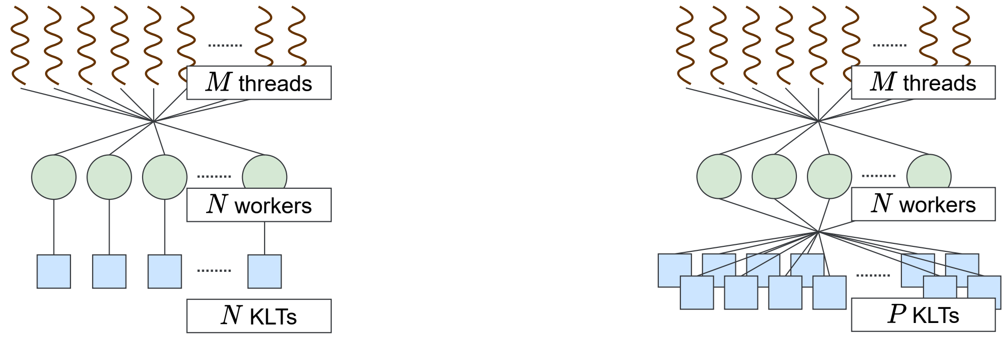
Many-to-many thread mapping models ("M:N threads")—where "M" user-level threads (ULTs) that are visible to the application programmer are mapped to "N" kernel-level threads (KLTs) that are managed by the OS—have been an area of active research for the past several decades. Several implementations of M:N threads have been proposed during this time [31]–[37]. This is in contrast to the simple one-to-one mapping model that current Pthreads implementations use ("1:1 threads"). The many-to-many mapping model used by M:N threads allows them to perform context switching and scheduling almost entirely in user space with no kernel involvement. This capability is the source of the high performance that most M:N thread implementations can achieve. In fact, modern M:N thread implementations such as Argobots [37]–[39], Qthreads [35], and MassiveThreads [36] can achieve several orders of magnitude lower overhead compared with current implementations of Pthreads, thus allowing for more fine-grained parallelism and schedulers optimized for it [16].

As beneficial as the kernel-bypass feature of M:N threads is, however, it has its own shortcomings. In particular, bypassing the kernel for scheduling and context switching causes M:N threads to lose the kernel-provided ability of *implicit OS preemption*. That is, M:N threads have to explicitly yield control for other threads to be scheduled. Since scheduling of M:N threads happens only at explicit *scheduling points*, programs that do not call scheduling operations in a timely manner cause M:N threads to occupy cores for a long time, resulting in load imbalance or loss of thread prioritization. Programs can even deadlock if nonpreemptive M:N threads fall into a busy loop waiting for the progress of other threads. Manually inserting explicit scheduling points in proper places would be a solution but is often impractically burdensome considering today's complex applications built on top of numerous dependent frameworks and libraries. Closed-source software exacerbates this situation since users cannot rewrite their implementations. Lack of implicit OS preemption narrows the applicability of M:N threads.

For simplicity, in the rest of this thesis we will refer to ULTs simply as "threads," as is common practice in the literature. Kernel-level threads will be referred to as KLTs, to clearly distinguish them from ULTs. As we mentioned, *ULTs can be implemented as either M:N threads or 1:1 threads*.[1] In fact, major thread specifications, including Pthreads [18] and OpenMP [17], do not require a one-to-one mapping of threads to KLTs.

In this thesis we investigate two techniques for M:N threads to achieve implicit OS preemption similar to 1:1

[1] We note that some literature uses the term "ULT" to denote M:N threads. In order to clearly distinguish KLTs and ULTs, a ULT in this section denotes a thread visible to user programs. By definition, Pthreads and OpenMP threads are ULTs, and their common implementations adopt 1:1 mapping.



(a) Nonpreemptive/signal-yield

(b) KLT-switching

Fig. 3.1.: Thread mapping to workers and KLTs in M:N threads.

threads: *signal-yield* and *KLT-switching*. The first technique—signal-yield—has been previously proposed [7]–[9] and also integrated into the Go language [40], [41]. This technique preempts a thread by interrupting the execution with a preset timer signal and context switching from within the signal handler. While this technique is simple, it is not directly applicable to several real-world applications because of two reasons.

1. Timer signals are not scalable and can be expensive on systems with large core counts. In this thesis, we present new optimizations to mitigate this overhead by coordinating the signals across different KLTs.
2. Signal-yield requires user functions to be "KLT-independent" and have the ability to execute correctly even if the underlying KLT is switched dynamically during execution. This requirement, unfortunately, is highly restrictive for many applications. For instance, the `malloc()` implementation in Glibc is KLT-dependent because it uses KLT-local data.

Our second technique—KLT-switching—overcomes the KLT-dependence issue in signal-yield by virtualizing the concept of a KLT into what we call a "worker." In other words, "M" threads would be mapped to "N" workers, which in turn would internally be mapped to a pool of "P" KLTs. Like signal-yield, KLT-switching also preempts a thread by interrupting the execution with a signal. Unlike signal-yield, however, instead of context switching the thread from within the signal handler, the entire KLT is suspended, and the worker is remapped to another KLT from the KLT pool. KLT-switching incurs higher overhead than signal-yield does, but it covers a wider range of applications. For consistency, we will use the term "worker" for both signal-yield and KLT-switching. In signal-yield, a worker would be equivalent to a KLT (Fig. 3.1a), while in KLT-switching there is no static mapping between workers and KLTs (Fig. 3.1b).

Both techniques involve kernel calls only for interruption; they require neither kernel modification (e.g., scheduler activations [42]) nor compiler support to insert preemption points [43], [44] because they rely only on OS features provided by today's mainstream OS implementations. Thus, the overhead of our techniques is incurred only at interruption, which is less than 1% when the preemption interval is 1 ms (OS preemption is typically in the millisecond range). We note that our implementation allows combining all three types of M:N threads within the same application: traditional nonpreemptive threads, signal-yield threads, and KLT-switching threads. Users can, therefore, further optimize programs by spawning the most suitable type of threads based on their needs for performance and threading features.

We evaluate three applications to evaluate the benefits of preemptive M:N threads. We first demonstrate the applicability of our preemption techniques with the Cholesky decomposition kernel in SLATE [10], which has nested parallelism and internally uses Intel MKL as a BLAS library. While using nonpreemptive M:N threads, instead of Pthreads, can improve the performance by reducing threading overheads in nested parallelism, they are not reliable and can deadlock in some cases. With our proposed preemptive M:N threads, we can achieve up to 27% performance improvement over Intel OpenMP but without deadlocks. The second evaluation is with HPGMG-FV [12], a high-performance geometric multigrid application, under *thread packing* [13]. Core starvation incurred by thread oversubscription under thread packing is efficiently alleviated by designing a scheduler specialized for thread packing, which outperforms traditional implementations that use 1:1 threads or nonpreemptive M:N threads. Our evaluation of the LAMMPS [14], [15] molecular dynamics application with in situ analysis shows that lightweight preemptive threads can help with thread prioritization.

3.2. Designing Preemption for M:N Threads

This section presents the design, implementation, and detailed analysis of two preemption techniques for M:N threads—signal-yield and KLT-switching.

3.2.1. Preemption Techniques for M:N Threads

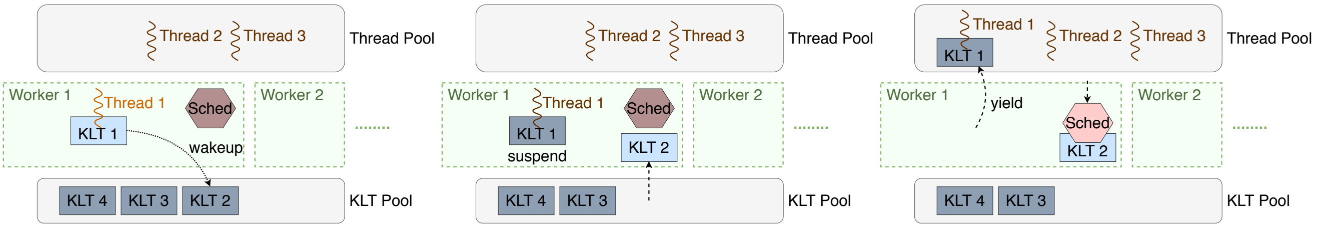
Signal-Yield

The first preemption technique that we study in this thesis—signal-yield—was proposed in [7]–[9], [41]. The central idea of signal-yield is that a running thread can be interrupted by using OS timer signals, thus emulating OS time slices. Once interrupted, the running thread will context switch to the scheduler from within the signal handler. Since the stack frame of the signal handler is located on top of the running thread's stack frame, the context switch saves contexts for both the signal handler and the running thread. When the thread is resumed, the execution context is still in the signal handler; then the control exits from the signal handler and returns to the thread's context. We note that some systems, including POSIX (by default), do not invoke a signal handler while the previous signal handler associated with the same signal is unresolved. For signal-yield, this would mean that only one thread can be preempted on each worker. To avoid this issue, we unblock the signal for the signal handler right before context switching from within the signal handler. This strategy allows us to preempt as many threads as needed within the same worker.

While signal-yield allows us to achieve OS time-slicing-like preemption capability, it has two shortcomings.

1. Because calling a signal handler involves taking a lock in the kernel, concurrent invocation of signal handlers on multiple cores can cause severe lock contention. Thus, if timer signals are issued at nearly the same time, some threads will spend more time to handle signals. For systems with large core counts this situation can cause significant overhead. In Section 3.2.2 we will describe optimization techniques to address this issue.

2. The signal-yield technique is relevant only for threads that execute KLT-independent functionality, namely, functions that can execute correctly even if the underlying KLT is switched dynamically during execution. This, essentially, restricts threads from using any KLT-local storage. Unfortunately, using KLT-local storage is common practice in many applications and libraries, making this a significant restriction. For instance, the `malloc()` implementation in Glibc uses KLT-local storage to save flags and cache data. If preemption happens in a KLT-dependent function, the next thread running on the same KLT may modify the KLT-local data by, for example, calling the same function again.

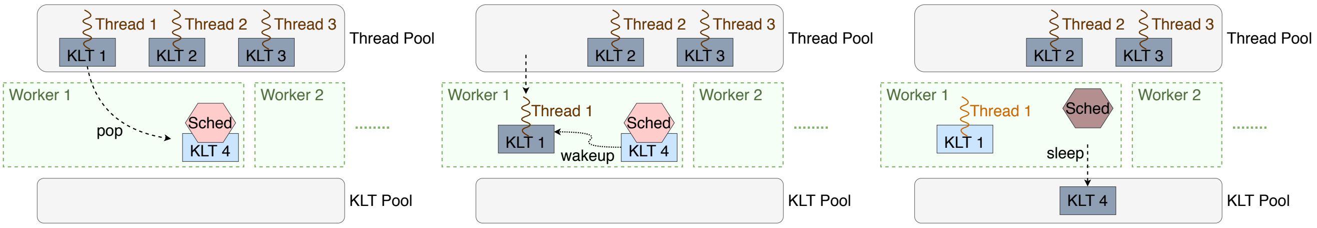


(a) When a thread (thread1) is preempted and a signal handler is called on it, it wakes up a KLT (KLT2) in the KLT pool.

(b) The new KLT (KLT2) wakes up, and the preempted thread (thread1) associates the previous KLT (KLT1) with itself and suspends its execution.

(c) The woken-up KLT (KLT2) runs the scheduler and puts the preempted thread (thread1) into the thread pool as if it had called a yield function.

Fig. 3.2.: Illustration of KLT-switching when preemption happens.



(a) The scheduler running on a KLT (KLT4) pops a preempted thread (thread1) from the thread pool.

(b) The running KLT (KLT4) wakes up the KLT (KLT1) associated with the preempted thread (thread1).

(c) The preempted thread (thread1) resumes on the same KLT (KLT1). The previous KLT (KLT4) exits from the scheduler and sleeps in the KLT pool.

Fig. 3.3.: Illustration of KLT-switching when rescheduling a preempted thread.

KLT-Switching

Our second preemption technique—KLT-switching—is designed to overcome the KLT-dependence issue in signal-yield. The core idea of this technique is to extend the concept of a worker so there is no static mapping between workers and KLTs. While a thread is not preempted, it works as a normal M:N thread does. When a thread is preempted, however, the suspended thread temporarily gets 1:1 mapping to the underlying KLT of the worker. The assigned KLT is restricted from executing other threads, so the KLT-local data remains unchanged until the suspended thread resumes. We dynamically allocate additional KLTs during suspension and remap the worker to a different KLT.

The overall working of KLT-switching is illustrated in Fig. 3.2 (showing the suspend path) and Fig. 3.3 (showing the resume path). In this example, worker1 is initially mapped to KLT1 and is executing thread1. When the time slice expires, thread1 receives a signal to suspend itself. At this point, three actions are done: (1) a new KLT (KLT2) is allocated and assigned to worker1; (2) the suspended thread (thread1) and the KLT on which it was executing (KLT1) are both pushed to the thread pool for later execution; and (3) the execution context switches back to the scheduler, which will now execute on the newly allocated KLT2. Similarly, when the thread needs to be resumed, again three steps need to be done: (1) the thread (thread1) and its corresponding KLT (KLT1) are popped back from the thread pool by the scheduler; (2) the execution context switches to thread1; and (3) the KLT on which the scheduler was executing (KLT4) is returned to the KLT pool. We note that even though the KLT pool can have more KLTs than the number of cores, not all of them are active at the same time. In fact, we keep active only as many KLTs as there are cores available.

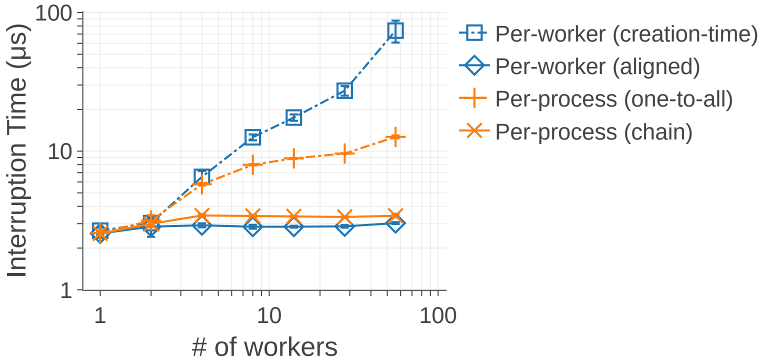


Fig. 3.4.: Average time for an OS timer interruption with 1 ms timer interval on Skylake.

An interesting detail to note is that KLT-switching needs a special mechanism to create KLTs. To allocate a new KLT upon preemption, the worker first checks the KLT pool to see whether any unused KLTs are available. If no unused KLTs are available, a new KLT will need to be created. Unfortunately, we cannot simply create new KLTs inside a signal handler because most KLT-creation functions (such as `pthread_create()`) are not *async-signal-safe*; in other words, they cannot be called from within signal handlers [18]. To work around this issue, we delegate the creation of KLTs to a dedicated KLT called "KLT creator." When a new KLT needs to be created, we send a request to the KLT creator, which then creates a KLT and adds it to the KLT pool.

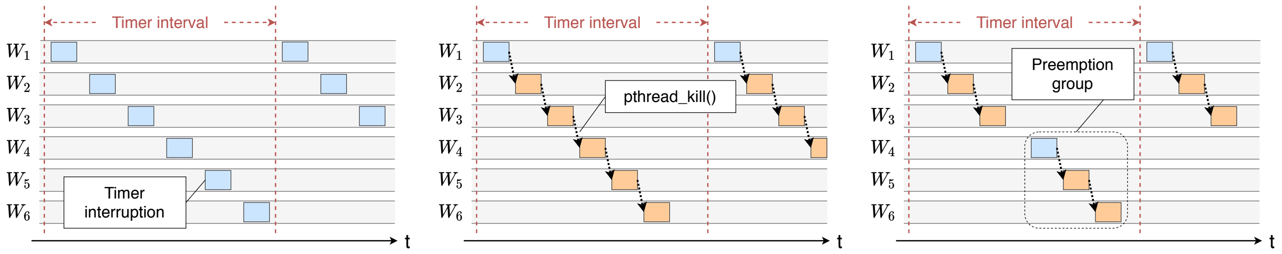
Another interesting detail to note is that once a request is sent to the KLT creator, the signal handler cannot wait for the completion of the KLT creation. The reason is that the KLT creation function itself might try to acquire a global lock, which would cause a deadlock if the preempted thread had already taken the lock. Therefore, the running thread must exit from the signal handler and wait for the next signal interruption to check whether a new KLT is available. When the running thread receives its next signal interruption, however, the newly created KLT may have gotten assigned to another worker. In that case, it would have to issue another request for a new KLT creation and go through the same cycle again. We note that while this situation may cause delay in some cases, there is no risk of livelock because, in the worst case, we would allocate as many KLTs as threads, thus simply deteriorating to a 1:1 threading model.

3.2.2. Optimizations for Preemption Timers

Both signal-yield and KLT-switching require a *preemption timer* to periodically send a signal to the running thread in order to force implicit preemption on preemptive threads. This is accomplished by using the `timer_create()` function. We consider two approaches to implement preemption timers: *per-worker* and *per-process*. With a per-worker timer, every worker has its own OS timer. In contrast, with a per-process timer, all workers in the process share one OS timer: one worker receives the signal and forwards it to other workers.

Per-Worker Timer

A naive implementation of a per-worker timer in which the timer of each worker is set independently (e.g., on worker creation) does not scale well on systems with large core counts. Fig. 3.4 shows the average time spent in a timer interrupt when all workers are interrupted every 1 ms. The plot shows an average of 1,000 interrupts with standard deviations. System settings are described in Section 3.3. The naive implementation (*Per-worker (creation-time)*) shows poor scalability, taking as much as 100 μs for large core counts (in contrast, thread context switching for M:N threads costs tens of nanoseconds). This behavior is because of signal contention. In Linux,



(a) Per-worker timer (aligned) (b) Per-process timer (chain) (c) Hybrid timer

Fig. 3.5.: Illustration of optimized preemption timers. A timeline of six workers is shown; the rectangles represent interruptions.

calling a signal handler involves taking a lock in the kernel, thus causing lock contention when multiple signals are issued at the same time.

To avoid such signal contention, we propose an approach called "timer alignment," where the timer interrupts across the different workers are explicitly aligned in order not to overlap (see Fig. 3.5a). This ensures that the signal handling cost does not increase with increasing core counts (*Per-worker (aligned)* in Fig. 3.4).

Nevertheless, per-worker timers have two shortcomings.

1. They are not portable. The feature of sending periodic timer signals to a specific KLT is not a part of the POSIX specification, although it is implemented in Linux.

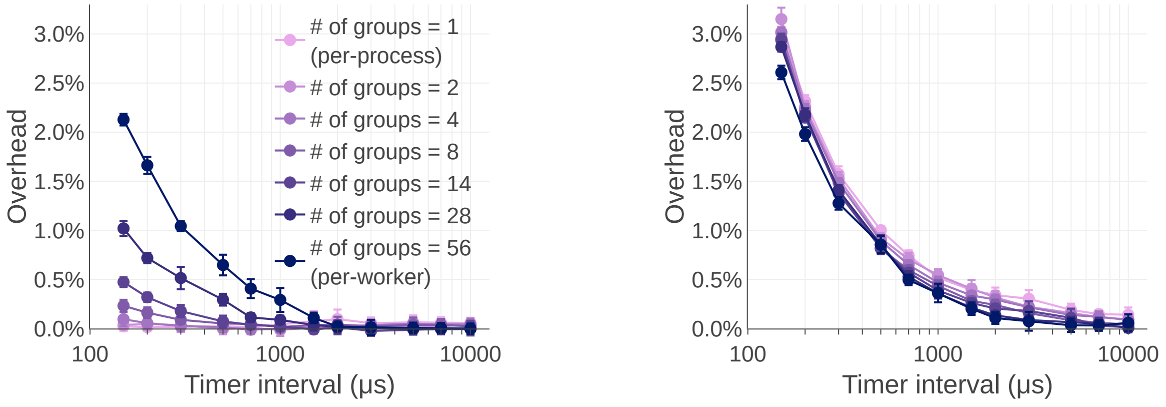
2. They do not distinguish between workers that have preemptive threads and workers that do not. Consider an application that has both preemptive and nonpreemptive threads: the per-worker timer would signal all workers, even if none of the currently running threads are preemptive, thus adding overhead with no benefit.

Per-Process Timer

To overcome the shortcomings in per-worker timers, we investigate "per-process timers" as an alternative signaling technique. In this technique, only one worker receives the periodic timer signal from the OS. It then checks whether the running thread on another worker is preemptive, and only if it is preemptive does it send a new signal to that worker by `pthread_kill()`. It repeats this procedure for the remaining workers. If none of the running threads are preemptive, no additional signals are issued.

While per-process timers can significantly reduce signal overhead in cases where there are few preemptive threads, they are not particularly beneficial when most threads are preemptive. In fact, as shown in Fig. 3.4 (*Per-process (one-to-all)*), the average interruption time continues to scale linearly when all threads are preemptive, taking tens of microseconds for large core counts. The reason for this increase is also signal contention because issuing a `pthread_kill()` call is much cheaper than signal handling. Hence, all workers try to process their received signals at nearly the same time, thus causing contention.

To avoid this signal contention, we propose a new optimization to per-process timers, called "chained signals." In this optimization, the worker that received the timer interrupt handles the signal and then issues a signal to at most one other worker, depending on which worker has a preemptive thread running. That worker in turn handles the received signal and again forwards the signal to at most one other worker. Thus, the workers are interrupted one by one like a chain reaction (see Fig. 3.5b). As shown in Fig. 3.4 (*Per-process (chain)*), this optimization



(a) Nonpreemptive threads

(b) Preemptive threads (signal-yield)

Fig. 3.6.: Overhead of preemption timers for a compute-intensive benchmark with different number of preemption groups on Skylake.

can significantly improve the average interruption time. We note that the performance of *Per-process (chain)* is slightly worse than that of *Per-worker (aligned)* because of the additional `pthread_kill()` calls.

Hybrid Timers

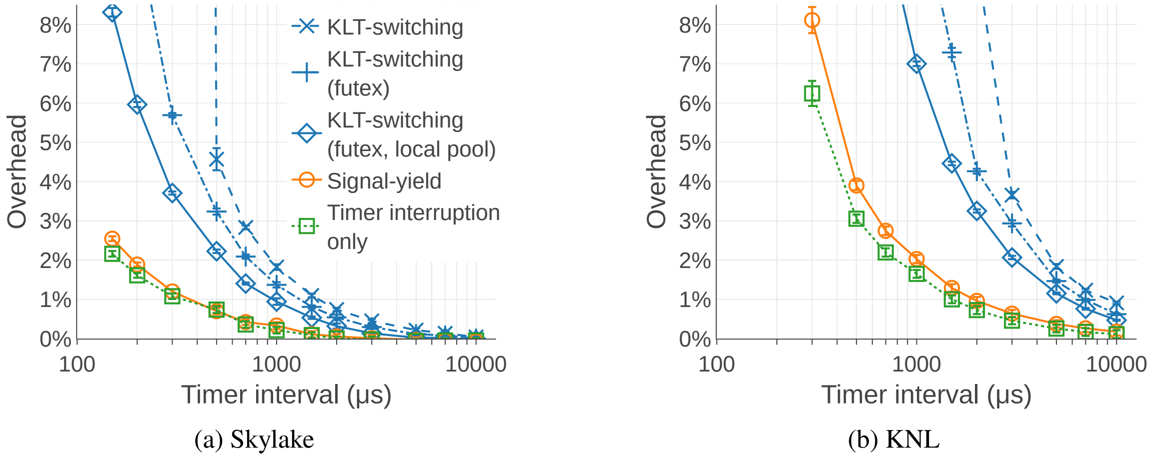
Per-worker and per-process timers both have their own advantages and disadvantages. Per-process timers are good for minimizing the overhead on nonpreemptive threads because, in the entire process, only one signal is issued every preemption interval. But they lose some performance compared with per-worker timers when most threads are preemptive. To address this trade-off, we propose a hybrid model (Fig. 3.5c) that collects the available workers into groups and issues one timer signal to each group: each worker forwards the signal only within its group. A single group that contains all the workers would be equivalent to the per-process model, while small groups that contain a single worker per group would be equivalent to the per-worker model.

To compare the different group sizes in the hybrid timer implementation, we use a simple microbenchmark in which each worker runs ten threads that just consume CPU cycles in a loop. Fig. 3.6a shows the overhead in the case where all threads are nonpreemptive, while Fig. 3.6b shows the overhead in the case where all threads are preemptive. They plot the average and standard deviation of 100 runs on Skylake, whose configuration is described in Section 3.3. As expected, for nonpreemptive threads the overhead is the lowest when the number of groups is one (i.e., per-process timers), while for preemptive threads the overhead is the lowest when the number of groups is the largest (i.e., per-worker timers).

3.2.3. Optimizations for KLT-Switching

While signal-yield is a low overhead preemption technique, as described in Section 3.2.1, it is relevant only for threads that execute KLT-independent functionality. KLT-switching, on the other hand, is a more generally applicable technique, although it can be more expensive than signal-yield. In this section, we analyze the performance of KLT-switching and present several optimizations to minimize its overhead.

Fig. 3.7 compares the overhead of KLT-switching with that of signal-yield with a compute-intensive microbenchmark in which each of 56 workers runs ten threads that just consume CPU cycles in a loop. In this experiment we use the optimized per-worker timer. The plot shows the average and standard deviation of 100 runs. Experimental environments of Skylake and KNL are summarized in Tab. 3.2 in Section 3.3. Based on the



(a) Skylake

(b) KNL

Fig. 3.7.: Relative overhead of preemptive M:N threads compared with nonpreemptive M:N threads over a compute-intensive benchmark. A per-worker timer is used.

experimental data, we note that the overhead of signal-yield is virtually identical to that of a pure timer interrupt (*Timer interruption only*). In other words, signal-yield does not add any additional overhead compared with that of the basic timer signal. We also note that a naive implementation of KLT-switching can add significant overhead compared with that of signal-yield.

Suspending and Resuming KLTs

Since we perform KLT-switching in a signal handler, suspending and resuming KLTs require using *async-signal-safe* functionality, which restricts usable functions in practice. A portable implementation of such functionality could use `sigsuspend()` to suspend a KLT at preemption and `pthread_kill()` to resume a KLT because both calls are *async-signal-safe*. While correct and portable, `sigsuspend()` involves additional signal handling, which leads to high overhead.

In this optimization, we replace the `sigsuspend()` and `pthread_kill()` calls with a futex-based suspend/resume implementation. Specifically, once a running thread receives a signal, it simply suspends its underlying KLT on a futex variable. When the KLT needs to be resumed, the resuming thread would simply wake up the suspended KLT with a `FUTEX_WAKE` operation. We note that futex is Linux-specific, but other OSs provide similar functionality. As shown in Fig. 3.7, *KLT-switching (futex)* can reduce the overhead of KLT-switching somewhat, but the overhead is still non-negligible.

Worker-local KLT Pool

Section 3.2.1 describes a model where KLTs are allocated and cached in a global pool. While this model reduces the average cost of KLT allocation, using a global pool can hurt data locality when the KLTs get resumed on different cores. In addition, in cases where we enable binding of workers to specific cores for performance, whenever a KLT is mapped to a different worker, that KLT's affinity needs to be reset. This can be expensive. To alleviate these overheads, we introduce "worker-local KLT pools," where, together with the global pool, each worker also maintains its own local pool. This optimization further reduces the overhead of KLT-switching, as shown in Fig. 3.7.

Our two optimizations together achieve approximately two times performance improvement, bringing KLT-switching's overheads within a range comparable to that of signal-yield. When the timer interval is short, Tab. 3.1.: Overhead of preemption.

	1:1 threads (Pthreads)	Signal-yield	KLT-switching
Skylake	2.8 μs	3.5 μs	9.9 μs
KNL	15 μs	18 μs	62 μs

KLT-switching can still add significant overhead; but once the timer interval is in the millisecond range, its overhead is less than 1% on Skylake. On KNL the overhead of preemption is relatively large because of its less powerful CPU architecture; to make the overheads less than 1%, we need to set about 10 ms as an interval.

We note that even with all the optimizations above, implicit preemption of M:N threads is costlier than that of 1:1 threads. Tab. 3.1 shows the median of preemption overheads obtained by repeating preemptions 1,000 times. We set the preemption interval to 10 ms, which is almost the same as the OS preemption interval in our system. The result shows that, on both Skylake and KNL, overheads of signal-yield and KLT-switching are approximately 1.2x and 4x higher than that of 1:1 threads, respectively. The merit of preemptive M:N threads is a combination of implicit preemption, scheduling flexibility, and other lightweight threading operations such as fork-join operations and synchronization primitives. Section 3.3 shows how preemptive M:N threads can take advantage of the preemption capability.

3.2.4. Choice of Thread Types

As mentioned earlier, our implementation allows three types of M:N threads to coexist within a single application: traditional nonpreemptive threads, signal-yield threads, and KLT-switching threads. We have a few recommendations on which thread type to choose. First, if a thread function requires no preemption or has explicit yield calls, we recommend using nonpreemptive threads because they are the most efficient. If preemption is needed and users know that the target thread function never calls a KLT-dependent function, signal-yield is recommended because, as shown in Fig. 3.7, the overhead of signal-yield is smaller than that of KLT-switching. If preemption is necessary and the thread function contains a KLT-dependent function, KLT-switching should be chosen. If users do not know what a thread function does (e.g., when replacing Pthreads with M:N threads in an existing large program), we recommend KLT-switching for correct execution.

3.2.5. Safe Use of Preemptive M:N Threads

Although preemptive M:N threads work similarly to 1:1 threads, they have some programming constraints.

System Calls and Signals

Unlike interruptions generated by a kernel, some system calls fail if a signal handler is triggered while executing them. We can resolve this problem by automatically restarting system calls by setting the `SA_RESTART` flag for the timer signal in `sigaction()`. But some system calls cannot be restarted automatically, and appropriate error handling is required. Moreover, users need to be aware that too short a timer interval would cause many restarts of system calls, which would affect the performance of blocking system calls that take a long time, such as I/O.

Replacement of 1:1 Threads with Preemptive M:N Threads

Implementing a complete substitute for existing 1:1 threads implementations (e.g., widely used Pthreads implementations) with preemptive M:N threads would significantly ease programmer effort in running existing applications over M:N threads. In reality, however, people use threading features beyond the pure specification of standardized threads, so developing a wrapper for a certain thread package is insufficient. Arguably, our preemptive M:N threads lack several nontrivial threading features. For example, on some architectures, programs utilize a register value that is unique to a thread (e.g., fs register on x86/64), so this value must be properly maintained. Without compiler modification, initialization of thread-local storage (TLS) would be problematic since a compiler assumes TLS of KLT and thus often relies on the initialization mechanism on KLT creation. As discussed above, our signal-based preemption affects some system calls, so this impact must be evaluated. Investigating a complete replacement of current 1:1 thread implementations is our future work.

Coexistence of Preemptive and Nonpreemptive Threads

Users must pay attention to how they manage their locks when both preemptive and nonpreemptive threads coexist in their application. If a preemptive thread is interrupted while holding a lock and a nonpreemptive thread scheduled later tries to take the same lock, a deadlock can occur. Users should be particularly careful when third-party functions use locks that are out of their control. For example, Glibc `malloc()` can take a global lock. Hence, preemptive threads and nonpreemptive threads should not be mixed unless one is sure that no lock is shared by both types of threads. When in doubt, it is safer to rely on preemptive threads instead of mixing thread types.

Writing a Scheduler

One of the strengths of M:N threads is the ability to create custom schedulers. In a preemptive thread runtime, a scheduler needs to be carefully implemented since it is also subject to the issue of async-signal-safety. Specifically, a scheduler must not call any non-async-signal-safe function because it can be switched from within the signal handler. Even if one uses KLT-switching, and a preempted thread and the scheduler are associated with different KLTs, non-async-signal-safe functions may take a global lock, which can cause a deadlock because a preempted thread may have already taken the lock. This situation can happen if both the scheduler and the application threads call non-async-signal-safe functions in the same external library (e.g., Glibc). In most scheduling algorithms, non-async-signal-safe functions are not essential, so we believe this is not a major hindrance.

3.3. Evaluation and Analysis

In this section, we experimentally evaluate and analyze the performance of preemption techniques for M:N threads with several microbenchmarks and three real-world applications. Tab. 3.2 shows the experimental environment used in Section 3. Although our preemption techniques are generic and applicable to other implementation of M:N threads, we chose Argobots (v1.0rc1) [37] for evaluation primarily because it has an OpenMP wrapper called BOLT (v1.0rc3) [45]. For comparison we used Intel MPI for inter-process communication and Intel OpenMP as a 1:1 thread implementation. To maximize the performance of Intel OpenMP, we tweaked its settings as follows. When more threads than cores are created, we set 0 to `KMP_BLOCKTIME` and disabled thread affinity; otherwise, `KMP_BLOCKTIME` and `OMP_PROC_BIND` were set to 200 ms (the default value) and `true`, respectively. We enabled nested hot teams [46] when parallel regions are nested. We ran each microbenchmark (Section 3.3.1) 100 times and each application reported in the succeeding sections 10 times; we plotted the mean and the standard deviation. In all experiments workers in Argobots were pinned to cores.

Tab. 3.2.: Experimental environment.

Name	Skylake	KNL
CPU Model	Intel Xeon Platinum 8180M	Intel Xeon Phi 7250
Frequency	2.5 GHz	1.4 GHz
# of Sockets	2	1
# of Cores	56	68
# of HWTs	112 (56 × 2)	272 (68 × 4)
L1 Data Cache	32 KB/core	32 KB/core
L2 Cache	1 MB/core	1 MB/2cores
L3 Cache	38.5 MB/socket	-
Memory	396 GB	204 GB
OS (Kernel)	Red Hat Linux 7.5 (3.10.0-862.14.4.el7)	
Compiler	Intel C/C++ compiler 19.0.4.243 (with the –O3 flag)	

3.3.1. Microbenchmarks

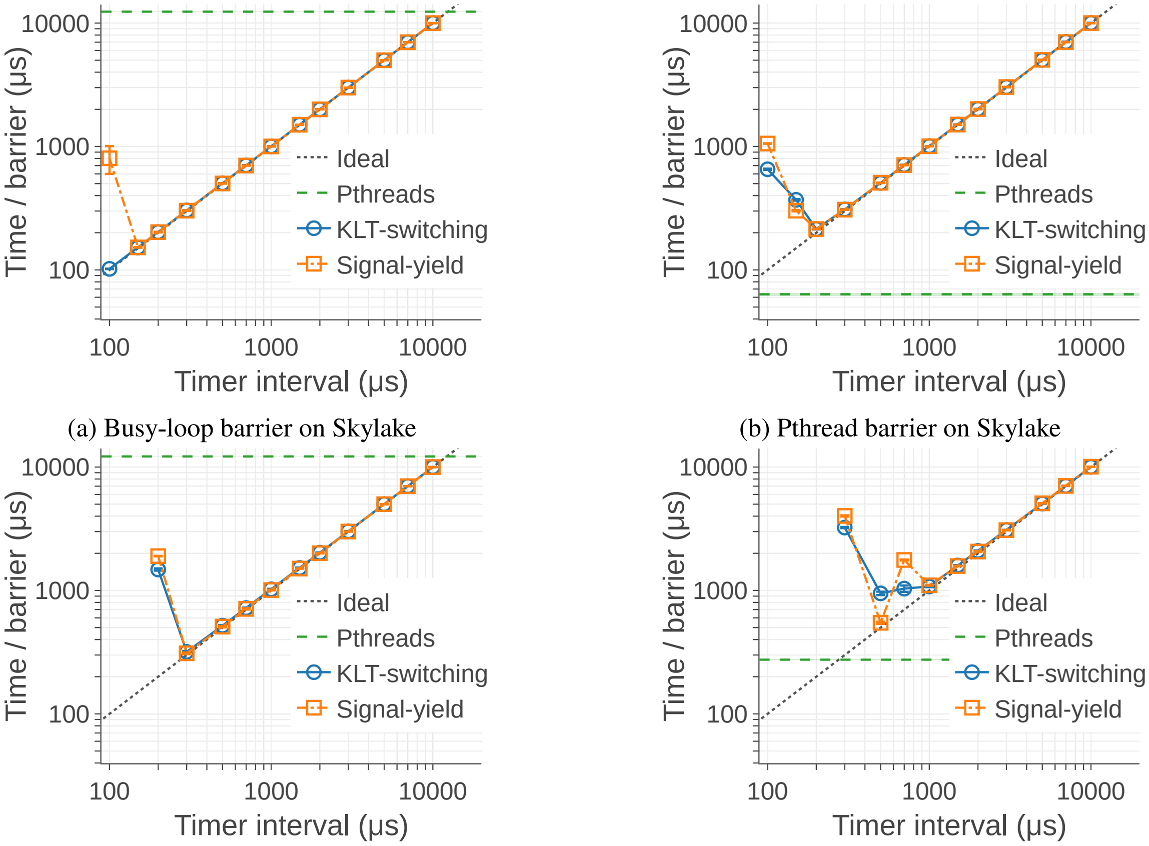
In this section, we use several microbenchmarks to evaluate the performance and overheads associated with the proposed preemption techniques in various targeted environments.

Barrier Benchmark

First we discuss an experiment in which nonpreemptive threads cannot be used because of the risk of deadlocks. In this microbenchmark, all threads perform 1,000 thread barriers. We use two versions of the thread barrier: (1) a busy-loop barrier that loops on an atomic variable; and (2) a Pthread barrier that uses `pthread_barrier_wait()`. Threads are equally distributed to each worker's FIFO queue and thus are scheduled in a round-robin fashion within a worker. In this experiment, we assign 10 threads to each worker. Since we create as many workers as the number of cores, this microbenchmark creates a total of *# of cores* × 10 threads. We also compare the performance with Pthreads by creating *# of cores* × 10 Pthreads while each thread is assigned to a core in order to mimic the affinity setting of M:N threads.

Fig. 3.8 shows performance results for different timer interval lengths. Y-axis represents a time to resolve one barrier, calculated by dividing the total execution time by $(\textit{\# of threads per worker} - 1) \times \textit{\# of barriers per thread}$. We note that the performance of Pthreads does not change in our experiment because its preemption is based on the OS-specified time slice which is unaffected by our experimental settings. In both, the busy-loop barrier (Fig. 3.8a and Fig. 3.8c) as well as the Pthread barrier (Fig. 3.8b and Fig. 3.8d), when the timer interval is large enough, signal-yield and KLT-switching perform close to the ideal execution time. But when the timer interval is small, the preemption overheads become visibly larger than the ideal one. This is because of increased signal contention in such cases. In particular, signal handlers usually take a few microseconds to execute. On Skylake, with 56 cores and a timer interval of around 100 μs, unless each signal handler can execute in less than $100/\textit{\# of workers} = 1.78$ μs, we cannot avoid signal overlap, and thus contention, even with alignment. On KNL, the effect of signal contention appears with larger timer intervals because of the larger number of cores and the less powerful CPU with lower frequency. In the busy-loop barrier benchmark, we already see some impact of this contention. But in the Pthread barrier benchmark, this effect is further exacerbated because, when a thread calling a futex system call receives a signal, the system call is first forced to exit before calling a signal handler. This additional overhead disturbs the timer alignment.

The performance of Pthreads in the busy-loop barrier benchmark is close to that of M:N threads with 10



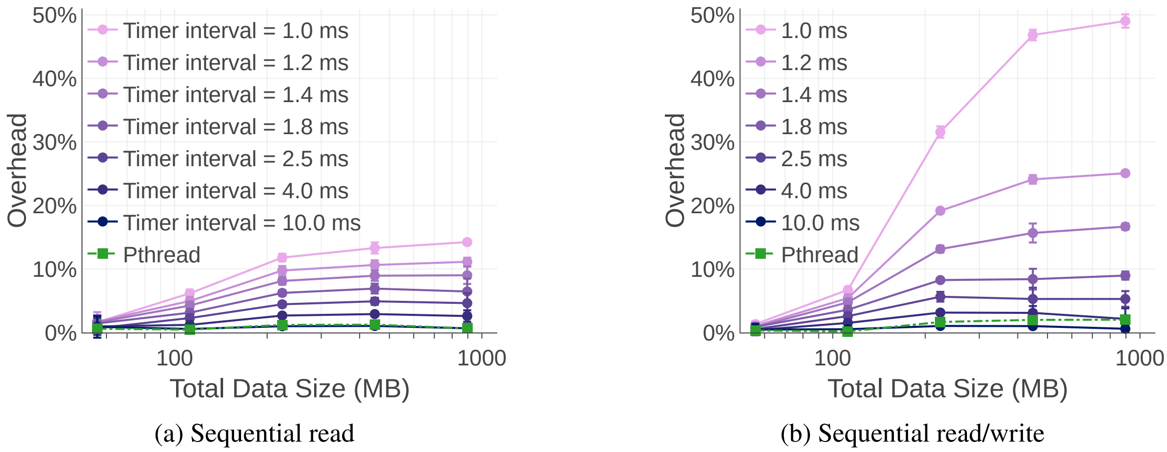
(a) Busy-loop barrier on Skylake

(b) Pthread barrier on Skylake

(c) Busy-loop barrier on KNL

(d) Pthread barrier on KNL

Fig. 3.8.: Performance of the barrier benchmark.



(a) Sequential read

(b) Sequential read/write

Fig. 3.9.: Overheads caused by preemption in cache-intensive benchmarks on Skylake. The baseline is nonpreemptive M:N threads.

ms timer interval, since the preemption interval in our OS setting is 10 ms. On the other hand, Pthreads in the Pthread barrier benchmark perform better than M:N threads because M:N threads are not aware of Pthread barrier (specifically, futex operations in `pthread_barrier_wait()`) and thus have to wait for timer interruption for context switching.

Effect of Data Locality

In this experiment, we evaluate the negative impact of preemption with respect to data locality. As such, this evaluation represents the worst case for preemption. Each thread runs a loop that sequentially accesses all elements of its own array that has the exact size of the L2 cache (i.e., 1 MB). If no preemption happens, most memory accesses would hit L2 cache. Preemption, however, can disturb this execution order because when a thread is preempted, it would be rescheduled after all other threads are scheduled (which would evict its array from the L2 cache). We use two microbenchmarks: *sequential read*, in which a thread repeatedly reads an array with sequential access, and *sequential read/write*, in which a thread updates every value of an array. The number of repetitions of data access is set to 10,000 so that each thread is preempted several times even with a timer interval of 10 ms. In the experiment we change the total number of threads, and therefore the total data size, while keeping the data size per thread fixed at the L2 cache size. The performance of signal-yield and KLT-switching was virtually identical in our experiments, so we chose to show only one of them (KLT-switching) here.

Fig. 3.9 shows overheads of preemption compared with a baseline of nonpreemptive M:N threads. When the total data size is small (56 MB), each worker has only one thread. Thus, preemption does not hurt data locality, adding at most 2% overhead compared with nonpreemptive threads. When the total data size increases to 112 MB, there is some overhead but it is somewhat reduced because the data still fits in the L2 and L3 caches.[2] Once the data size grows larger than that, cache misses would require memory access thus increasing the overhead. Our results indicate that frequent preemption can degrade performance if an application is cache-intensive. With a preemption interval of 10 ms, preemptive M:N threads have almost the same overhead as Pthreads (whose preemption time slices are in the same range).

[2] Skylake uses noninclusive L2 and L3 caches summing up to 133 MB.

3.3.2. Deadlock Prevention in Cholesky Decomposition

OpenMP [17] is one of the most popular multithreading programming models. Although most production runtimes such as GCC, LLVM, and Intel OpenMP adopt 1:1 threads for OpenMP threads, numerous papers have reported substantial performance benefits from mapping OpenMP threads to lightweight M:N threads [45], [47]–[50]. ABI compatibility of such M:N thread-based implementations with major OpenMP runtimes has also been studied [45], [49], [50]. Thus, in theory, such M:N thread-based OpenMP libraries can run existing OpenMP applications without recompilation. In practice, however, a deadlock can occur when programs run on nonpreemptive M:N thread-based OpenMP systems since numerous OpenMP programs assume preemptive threads. Preemptive M:N threads are expected to enhance the performance without deadlocking.

To evaluate the performance, we focus on the Cholesky decomposition kernel extracted from SLATE [10], a modernized linear algebra library. This kernel uses an OpenMP backend with nested parallelism. The outer parallelism uses OpenMP tasks with data dependencies to decompose its computation into operations of submatrices (*tiles*). Computation of each tile calls BLAS routines including DGEMM, TRSM, HERK, and POTRF in an external BLAS library, Intel MKL in our case, in which the inner parallelism exists. OpenMP-parallel Intel MKL, however, assumes implicit preemption during thread synchronization by having threads busy-loop on a memory flag, which causes a deadlock when running on nonpreemptive M:N threads. We used BOLT, an Argobots-based OpenMP library that is fully ABI compatible with LLVM OpenMP 9.0 [45]. We modified BOLT to work together with KLT-switching and per-worker timers for preemption. Our thread scheduler is based on BOLT's default work stealing [16] scheduler, in which each worker prioritizes execution of threads in its own FIFO queue and steals a thread from a randomly chosen remote queue when its local queue is empty. Upon preemption, the scheduler pushes the preempted thread into its local FIFO queue and pops the next thread. This preemption mechanism guarantees that all threads are scheduled within a finite time period, preventing a deadlock caused by busy loops.

As a point of comparison, we reverse engineered the closed-source Intel MKL library, by looking through its generated assembly code, to create a version of the code that explicitly yields the thread while waiting for the flag to be set, so it would work with nonpreemptive M:N threads too. Obviously, such reverse engineering is a hack and is only meant to demonstrate the best possible performance that is achievable with nonpreemptive M:N threads. Such modification might not be possible for other applications and does not cover all architectures or all MKL routines, but it is sufficient for our experiment.

Because Cholesky is too complicated to analyze the basic performance characteristics of OpenMP over preemptive M:N threads, we first focus on parallel DGEMM, which is the heaviest subroutine in Cholesky. We then show the performance of Cholesky decomposition.

Evaluation of MKL DGEMM

In this simplified evaluation setting, we ran multiple DGEMM calls in a parallel region, where each DGEMM routine uses 4000×4000 matrices. We fixed the number of outer OpenMP threads to 32 and changed the thread count of the inner parallel regions (i.e., the number of threads spawned in each DGEMM routine).

Fig. 3.10 shows the relative performance of Intel OpenMP and BOLT with preemptive M:N threads, compared against the baseline of BOLT with nonpreemptive M:N threads. As noted above, BOLT with nonpreemptive M:N threads uses our reverse-engineered hack to run to completion without deadlocking. For Intel OpenMP we did not bind threads to cores because it performed the best under oversubscription. We note a few observations in the graph.

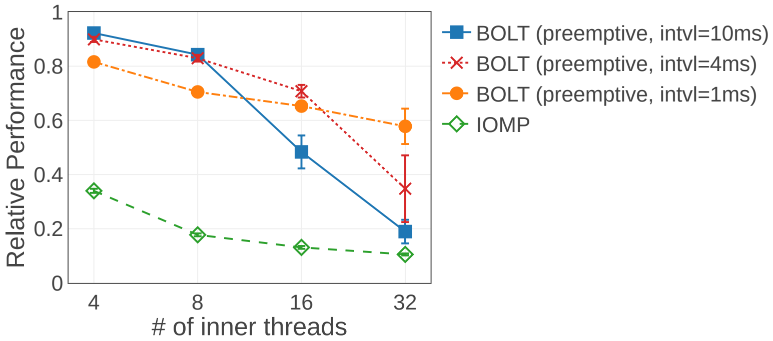


Fig. 3.10.: Performance relative to BOLT (nonpreemptive threads, with a reverse-engineered hack) in MKL DGEMM on Skylake.

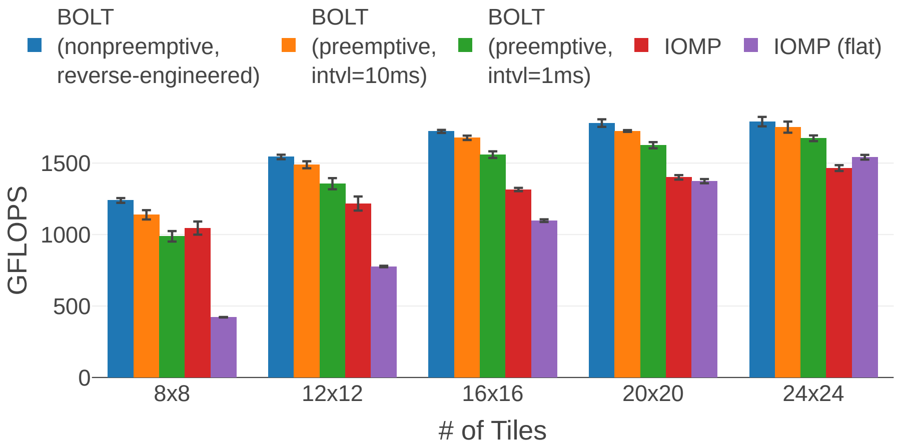


Fig. 3.11.: Performance of Cholesky decomposition.

1. Intel OpenMP performs significantly worse than BOLT with nonpreemptive threads, achieving somewhere from 10–40% of its performance. This is mainly because Intel OpenMP wastes considerable amount of time for *inter-thread synchronization in the OpenMP runtime* (i.e., not one embedded in MKL). In contrast, BOLT provides efficient OpenMP implementations that utilize lightweight threading operations of M:N threads (e.g., fork-join and inter-thread synchronization), which can significantly reduce the synchronization cost in the OpenMP runtime. We note that synchronization embedded in MKL requires implicit preemption, so preemptive M:N threads are required to run this benchmark over BOLT.

2. BOLT with preemptive threads loses some performance compared with BOLT with nonpreemptive threads, primarily because of the loss of data locality, but is still able to outperform Intel OpenMP significantly.

3. There is a trade-off that occurs with the preemption time interval—when the timer interval is large, the effect for data locality is small, but more time is wasted during thread synchronization internally implemented in Intel MKL. When the number of inner OpenMP threads is large there are more dependencies among inner threads, thus the wasted time in thread synchronization is more pronounced.

Evaluation of Cholesky Decomposision

In the evaluation of Cholesky decomposision, we fixed the tile size to 1000×1000 and changed the number of tiles. The outer parallelism exists across computations over tiles and the inner parallelism exists inside each tile. Fig. 3.11 shows the results. *IOMP (flat)* shows the performance of Intel OpenMP when inner parallelism is disabled and the outer parallelism is set to the number of cores (56). In all other cases, inner parallelism is enabled, and both inner and outer parallelism are set to 8. In these settings, nested parallel versions, including BOLT and Intel OpenMP, perform better than the flat version because the outer parallelism is not sufficient to make all cores active. In almost all cases, BOLT with preemptive M:N threads outperforms Intel OpenMP thanks to the lightweight threading operations in M:N threads [45]. In this application, larger timer intervals achieve better performance because short preemption intervals incur non-negligible cache misses.

3.3.3. Thread Packing with HPGMG-FV

OS schedulers designed for general purposes, such as completely fair scheduler (CFS) in Linux, do not perform well for some workloads. We found performance degradation caused by OS schedulers in *thread packing* [13], [51], [52], where threads are dynamically packed into fewer cores for power capping or resource partitioning. With 1:1 threads, CPU affinity masks of threads are dynamically changed so that they are executed on a limited set of cores. This causes significant core starvation because of poor load balancing of OS schedulers for parallel workloads [53]–[55]. Nevertheless, users often do not have permission to change the scheduling algorithm of OS schedulers on large-scale computing platforms. Preemptive M:N threads are promising since their user-level scheduling and preemption capability give chances to schedule other threads for better load balancing at the preemption interval.

In this evaluation, we demonstrate that preemptive M:N threads with a user-level scheduler can minimize the performance degradation caused by thread packing. To dynamically change the number of active workers, our preemptive M:N thread-based runtime dynamically wakes up workers by signals or suspends workers upon preemption. Threads that were executed by suspended workers will be executed by the remaining active workers. Let us assume typical HPC workloads where a fixed number of threads as many as cores are created and the computation load is equally balanced among the threads. Under thread packing, the key to good load balancing is how to schedule extra threads that suspended workers have in addition to local threads originally assigned to currently active workers. The idea of our proposed scheduler is to equally balance the load of extra threads across all the active workers by prioritizing extra threads. Preemption allows our runtime to schedule extra threads in a round-robin fashion among active workers, slicing extra threads by a preemption interval.

Algorithm 3.1 shows the pseudocode of the scheduler. Each worker has a unique integer ID, called *rank*, within the range $[0, N_{total})$, where N_{total} denotes the initial number of workers. Upon thread packing, workers with larger ranks are suspended. The scheduling algorithm consists of two phases: scheduling of threads in *private* pools (lines 7–10) and *shared* pools (lines 11–14). The private pools are exclusively assigned to a worker while the shared pools are shared by all the active workers. Initially only the local pool of each worker is private to the worker while no pools are shared. Under thread packing, the shared pools include local pools of suspended workers. Each worker repeats executing a thread in one of its private pools (lines 7–10) and then a thread in one of its shared pools (lines 11–14) alternately at the preemption interval. Since the preemption interval is the same among all workers, threads in the shared pools are scheduled in a round-robin fashion among all active workers, as active workers peek the shared pools in turn. This strategy, however, hurts data locality when fewer workers are active because most threads are in the shared pools and scheduled by different workers. To avoid this, when the number of active workers (N_{active}) becomes half of N_{total} or less, we exclusively assign more than one pool ($N_{private}$ pools in line 6) to the private pools of every active worker. Thus the number of shared pools are always less than the number of workers, which makes shared pools prioritized against private pools.

Algorithm 3.1: Scheduler specialized for thread packing.

```
$N_{total} \leftarrow$ # of workers that are initially created
$pools \leftarrow$ List of $N_{total}$ thread pools
$rank \leftarrow$ Unique ID of this worker within the range $[0, N_{total})$
while true do
    $N_{active} \leftarrow$ # of current active workers
    $N_{private} \leftarrow N_{active} \times \lfloor N_{total}/N_{active} \rfloor$
    for $i \leftarrow rank$ to $N_{private}$ step $N_{active}$ do
        if $thread \leftarrow pop(pools[i])$ then
            run($thread$)
            break
    for $i \leftarrow N_{private} + 1$ to $N_{total}$ step 1 do
        if $thread \leftarrow pop(pools[i])$ then
            run($thread$)
            break
```

This achieves good load balancing for threads in the shared pools while keeping good data locality for threads in the private pools.

For evaluation, we chose the finite volume version of High Performance Geometric Multigrid (HPGMG-FV) [12] (version 0.4), which solves linear equations using a full multigrid method. We set eight as the log base 2 of the dimension of each box on the finest grid and allocate eight boxes per process. We created two MPI processes in a single node and bound each process to a NUMA node for better locality. As an M:N thread-based OpenMP runtime we used BOLT [45], but we replaced the scheduler with the one we propose (Algorithm 3.1). We used KLT-switching and per-worker timers for preemption. For comparison to the OS scheduler, we also evaluated the performance of Intel OpenMP (IOMP).

Fig. 3.12 shows performance evaluation of HPGMG-FV when we first create 28 threads per process and reduce the number of active cores from 28 to n (x-axis) in each process. It shows relative overheads compared to a baseline where n threads are created from the beginning, together with the absolute execution time of the baseline to solve a linear equation. We arbitrarily chose the performance of BOLT over nonpreemptive M:N threads as the baseline since Intel OpenMP and BOLT showed almost the same performance. For *IOMP*, we use `taskset` command to bind 28 threads to n cores as done in [51], while for *BOLT* we suspend $28 - n$ workers. The results show that the performance of *IOMP* is far from the ideal performance especially when the number of active cores is close to 28. This is because of inefficient load balancing of CFS in Linux [53]–[55]. On the other hand, the performance of preemptive BOLT is close to the ideal one. It is interesting to note that 1 ms preemption interval performs better than 10 ms; this result indicates that 10 ms interval is insufficient to balance the load of extra threads in this case. *BOLT (nonpreemptive)* shows good performance when the number of cores is a divisor of 28, but in other cases the performance is poor because, without preemption, there are not enough scheduling chances for load balancing. This evaluation shows the benefits of user-level scheduling with preemptive M:N threads and configurable preemption intervals.

3.3.4. In Situ Analysis with LAMMPS

Preemption gives more control over thread scheduling, enabling non-voluntary priority-based scheduling, for example. With preemption, low-priority threads can be immediately evicted in favor of high-priority threads even if low-priority threads do not voluntarily yield. As a case study for priority-based scheduling, we picked *in*

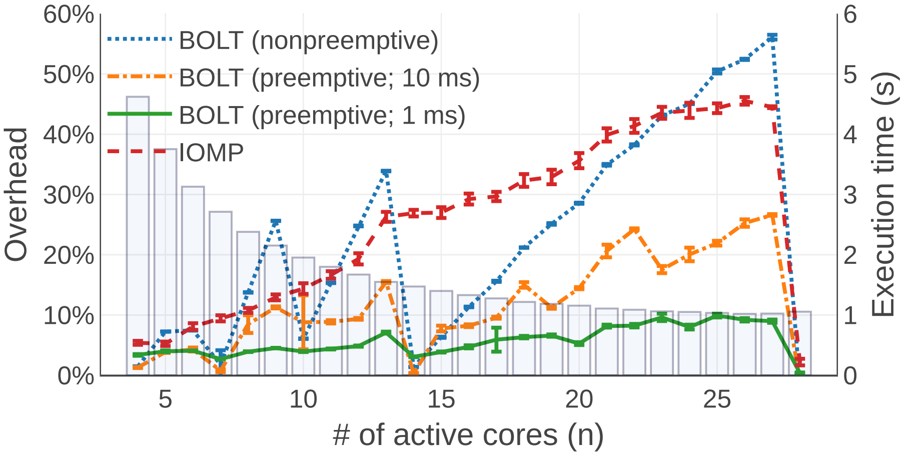


Fig. 3.12.: Relative overhead of threading packing in HPGMG-FV. The number of active cores per process is reduced from 28 to *n* while creating 28 threads. The baseline is BOLT (nonpreemptive) when spawning *n* threads from the beginning. The baseline's execution times to solve a linear equation are shown as the bar plots.

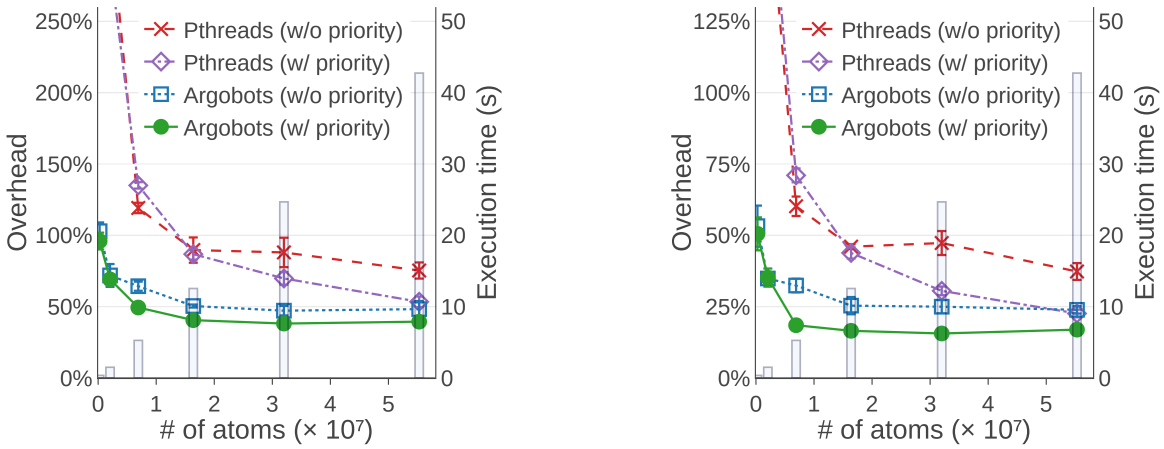
situ analysis in LAMMPS, where thread prioritization is favored.

LAMMPS [14], [15] is a molecular dynamics simulator developed by Sandia National Laboratories. For our experiments, we used version `stable_7Aug2019` together with the in situ analysis discussed in [56]. We used the Kokkos [57], [58] package for shared-memory parallelism in the simulation code and implemented an Argobots backend in Kokkos which spawns as many simulation threads as the number of workers in every parallel region. The analysis code copies all atoms to a separate buffer and performs analysis on this buffer in parallel, while the simulation is going on, by spawning dedicated analysis threads. In the simulation, we calculate the 3D Lennard-Jones potential for 100 time steps.

Simulation threads have higher priority than analysis threads have because analysis threads are created based on the progress of simulation threads. Too eager scheduling of analysis threads will delay the execution of simulation threads, which can degrade the overall performance. Thus, analysis threads should be executed only when no simulation threads exist (e.g., during MPI communication). The Pthreads version of the code achieves such prioritization by setting a higher "niceness" (lower priority) to the analysis threads than the simulation threads. The Argobots version of the code achieves prioritization by having the scheduler first check whether any simulation threads exist before checking for analysis threads. Only analysis threads are set to be preemptive, and every worker has a LIFO queue for analysis threads in order not to hurt data locality during preemption. We use signal-yield for preemption because the analysis functions are KLT-independent. To minimize the overhead of timer interruption for nonpreemptive simulation threads, we use the per-process timer. The preemption timer interval is fixed to 1 ms. We use four nodes, each of which has one MPI process that consists of 56 workers.

Fig. 3.13 shows the performance comparison of Pthreads and Argobots (with and without prioritization) compared with a baseline of a simulation-only (no analysis) execution. Fig. 3.13a shows the overhead when analysis is executed in every iteration and Fig. 3.13b shows the overhead when it is executed every two iterations. In our experiments, we create one less analysis thread than the available cores because the main thread rarely becomes idle—it is either computing or communicating; the remaining threads, on the other hand, become idle when the simulation is going through sequential portions of the code. For the Pthreads version of the code, simulation threads are created by Intel OpenMP and analysis threads are spawned via the Pthreads interface.

The result shows that Argobots-based execution achieves better performance than Pthreads-based execution



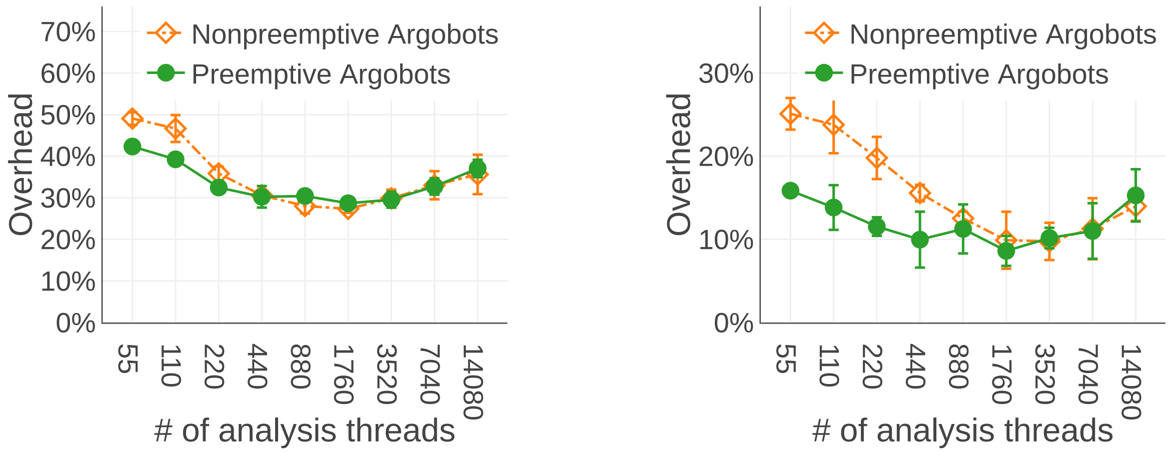
(a) Analysis interval = 1

(b) Analysis interval = 2

Fig. 3.13.: Relative overhead of in situ analysis with LAMMPS compared with simulation-only execution when the problem size is changed on four Skylake nodes. The execution times of simulation-only execution for 100 steps are shown as the bar plots.

because of its lower-overhead threading ability. Especially when the total number of atoms is small, the improvement in performance is significant. Prioritization helps both Pthreads- and Argobots-based executions to reduce core idleness and improve performance when the number of atoms is large. For example, in Fig. 3.13b, when the number of atoms is 5.6×10^7, prioritization reduced 30% of the core idle time in the case of Pthreads and 18% of the core idle time in the case of Argobots. The impact of prioritization is more pronounced when analysis is performed once every two iterations (Fig. 3.13b) compared with when analysis is performed in every iteration (Fig. 3.13a). This is because, when the analysis interval is 2, the amount of analysis work is small enough to complete while no simulation threads exist (e.g., during MPI communication). We note that, because nice values do not impose strict scheduling order, the execution of the Pthreads-based simulation is still uncoordinated. We could have further improved performance by setting strict priority, for example, by using real-time scheduling policies defined in POSIX, such as SCHED_FIFO. However, these policies require root privilege which is often unavailable to users. As a result, Argobots with prioritization using preemptive threads performs the best amongst all of the different approaches—this is the version that is enabled by the work in this thesis.

Fine-grained thread decomposition can also achieve a goal of priority-based scheduling as also investigated in [59]. This fine-grained version divides computation for in situ analysis into small pieces of work (i.e., more than the number of cores), so a scheduler has a chance to schedule the simulation thread when each analysis thread completes even without preemption. Fig. 3.14 shows the performance with various numbers of analysis threads. The number of atoms is fixed to 1.6×10^7. The number of nodes is four as we set in the previous evaluation. The result shows that overdecomposition improves the performance of both versions because of better load balancing across workers. *Nonpreemptive Argobots* is further improved since fine-grained decomposition also allows timely execution of simulation threads, but in all cases, it performs at best as good as *Preemptive Argobots*. If we carefully tune the thread granularity, even nonpreemptive threads can achieve the same performance as preemptive threads; however, tuning of thread granularity is difficult and burdensome in general. Excessive decomposition adds parallelization overheads and degrades the overall performance, and the optimal thread granularity would depend on the input size, the architecture, and so on. Although in this evaluation we focus only on a single analysis workload, but in practice there are various kinds of analysis;



(a) Analysis interval = 1

(b) Analysis interval = 2

Fig. 3.14.: Overhead of in situ analysis compared with simulation-only execution when the number of analysis threads is changed on four Skylake nodes. Simulation threads are prioritized over analysis threads in both cases. We omit Pthreads because it performs poorly under excessive oversubscription.

tuning for every analysis workload would require significant effort. This result indicates that, although the performance could be further optimized by manual optimizations traditionally applied to nonpreemptive M:N threads, preemptive M:N threads and its priority-based scheduling enable a promising optimization strategy that is robust to thread granularity.

3.4. Summary

We have investigated two lightweight preemption techniques for M:N threads: *signal-yield*, a technique that is simple but cannot run KLT-dependent functions, and *KLT-switching*, a novel technique that addresses the issue of KLT-dependence. These two techniques have different trade-offs: KLT-switching covers a wider range of programs but has higher overheads than does signal-yield. Our analysis shows that preemption of M:N threads can be achieved with nearly no overhead compared with nonpreemptive M:N threads and with high scheduling flexibility. The results of our evaluation with real-world applications are promising. Preemption can successfully improve the performance of nested OpenMP parallel programs without incurring a deadlock, resolve load imbalance under thread packing, and increase core utilization by efficiently handling thread priority.

Although our initial motivation of this work was to resolve the deadlock issue arising from nested composition of parallel programs (especially with Intel MKL), we found that user-level preemptive scheduling for threads was also beneficial for other use cases beyond nested parallel computations, such as thread packing and in situ analysis. We note that using preemption to resolve a deadlock in nested composition of parallel programs is just a workaround; a large amount of time can be wasted in busy-loops and thus there is no guarantee that this workaround always performs well. For applications with nested composition of parallel libraries to perform better, it is desired to remove the assumption that each parallel program can exclusively use all the system resources and not to rely on preemption. The next chapter investigates scheduling algorithms for hierarchical, recursively nested parallelism, which is a promising parallel programming approach to alleviate the assumption of exclusiveness.

4. Almost Deterministic Work Stealing (ADWS)

This chapter explains the design and implementation of Almost Deterministic Work Stealing (ADWS), a task scheduler for nested parallelism that aims to perform well on arbitrary memory hierarchy. The content in this chapter is based on the conference paper [1] with some revision and additional explanation.

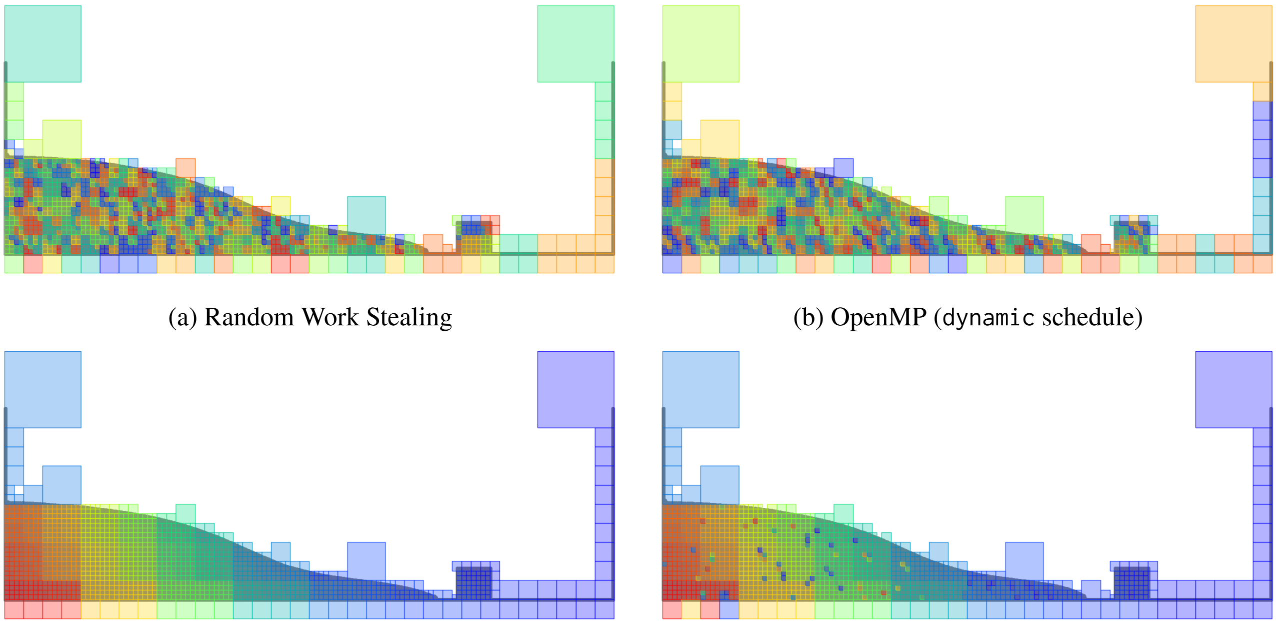
4.1. Introduction

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 [60]. 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 [22]–[24], Chapel [34] and X10 [33]. There are also task parallel runtime libraries which don't require compiler support, including Intel TBB [25], Argobots [37] and MassiveThreads [36]. Task parallelism was also introduced in OpenMP 3.0 [61] and has evolved to support dependencies and affinities [17].

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 [28], 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 [16], 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 [62], [63], which take non-trivial programmer efforts. HotSLAW [64] 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 [65]–[70]. Obviously, they are applicable only to iterative applications.



(a) Random Work Stealing

(b) OpenMP (`dynamic` schedule)

(c) ADWS (no steal)

(d) ADWS

Fig. 4.1.: 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 Fig. 4.21. The color of the cells represents the rank of workers, and the workers are colored gradually from blue to red in numerical order.

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 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 4.2.2, 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 [36] and experimental results show it outperforms other existing scheduling methods.

Fig. 4.1 helps us understand how ADWS works. It visualizes the task mapping overlaid on the figure of 2D particle simulation (Fig. 4.21). Fig. 4.1a shows how tasks are mapped on cores by random work stealing, whereas Fig. 4.1c and Fig. 4.1d show the mapping of deterministic and "almost" deterministic schedulers, respectively. In ADWS (Fig. 4.1d), workers steal tasks to compensate the load imbalance based on the task distribution of

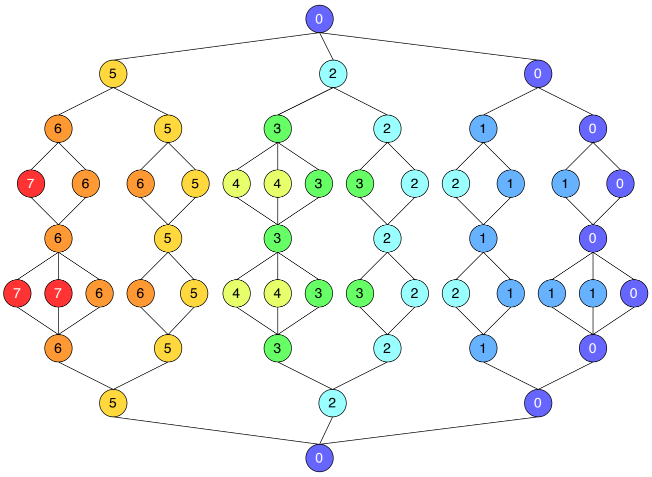


Fig. 4.2.: Example of the desired distribution of tasks we propose. The number in the nodes denotes the worker who executes the task. The tasks are split vertically in the DAG and allocated to workers from right to left in numerical order.

Fig. 4.1c while maintaining most of the data locality.

To summarize the main contributions, this chapter

1. proposes Almost Deterministic Work Stealing (ADWS) scheduler, which consists of
    a) deterministic task allocation, which initially allocates tasks to workers deterministically at runtime (Section 4.2.3) and
    b) hierarchical localized work stealing, which dynamically steals from a limited range of workers (Section 4.2.4), and
2. shows ADWS outperforms other existing scheduling strategies with memory-bound applications (Section 4.4).

4.2. Algorithm and Design

4.2.1. Overview

Efficient task schedulers should exploit both locality in iterative parallel programs 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. For locality in the computation graph, it is important to allocate close tasks in a DAG to workers in the same memory group (e.g., shared cache, NUMA node). Fig. 4.2 shows the desired distribution of tasks we consider. 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 assume 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 Fig. 4.2, we propose *Almost Deterministic Work Stealing (ADWS)*. ADWS consists of two parts. The first part is *deterministic task allocation*, which initially allocates tasks to

[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 4.2.3). Recall that first spawned tasks are placed in the left as noted in Section 2.3.

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

(a) Task group in TBB

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

(b) Task group with a reduced number of tasks

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

(c) Extension for ADWS

Fig. 4.3.: TBB's task group notation and extension for ADWS.

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. In the following, we first explain the programming model assumed in this thesis in Section 4.2.2, followed by explanation of deterministic task allocation (Section 4.2.3) and hierarchical localized work stealing (Section 4.2.4).

4.2.2. Programming Model

In ADWS, programmers have to specify the amount of work for each task as a hint. Even if the amount of work is roughly known and not precise, hierarchical localized work stealing is expected to fix the load imbalance in a locality-aware manner. Although specifying the amount of work requires additional efforts from the programmer, the hints are application-specific and independent of specific hardwares. Thus, these hints do not hurt the potability of codes; the programmer does not need to worry about the underlying memory hierarchy in principle.

For convenience, we adopt TBB-like task group notation [25] as a programming model for ADWS in the following. We note that ADWS should be applicable to other language primitives for nested parallelism such as `spawn` and `sync` in Cilk [22]–[24]. We can spawn tasks and wait for them by using task groups as shown in Fig. 4.3. The last task in the task group can be left unspawned to reduce the overheads for task creation. 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`). The parameter `w4` can be omitted if the last task is not spawned. 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 we did not do that because it makes many programs simple (demonstrated by the example of calculation of particle interactions below).

There is one thing to be noted for ADWS to perform well for nested parallel programs. It is about consideration for the execution order of parallel tasks, which is offered by the scheduling policy of ADWS. Because each worker's execution order of parallel tasks is usually based on the serial execution order, the programmer should write the program so that the data access pattern becomes good for serial execution. Moreover, consecutive task groups are scheduled almost deterministically, which assumption enables further optimization to promote cache reuse across task groups. If a successive task group accesses the data in almost the same pattern as in the previous task group, the same data is likely to be accessed by the same worker. It is demonstrated by the matrix-multiplication example introduced later.

```
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();
  }
}
```

Fig. 4.4.: Example code of calculation of particle interactions with hints on the amount of work.

Example 1. Calculation of Particle Interactions

We introduce particle simulation as an example to demonstrate how to specify the amount of work for each task. Suppose that particles are managed by a tree structure as in FDPS, a framework for developing parallel particle simulation codes [71], which manages a group of particles by using the Barnes-Hut octree structure [72]. Based on this octree structure, let us calculate interactions with neighbor particles. A simple way to do it is to traverse the octree and to perform calculation for particles in each leaf node. With fork-join structures, we can easily parallelize it by spawning each node as a task as traversing the octree.

Fig. 4.4 shows pseudocode of calculation of particle interactions. It specifies the number of particles in an octree node as the amount of work for the corresponding task. Usually, the octree is constructed based on the number of particles in cells; therefore we can assume that each node is attributed with the number of particles. Although the number of particles is just an approximation for the amount of work, we can use a rough estimate of work in ADWS, counting on the dynamic load balancing behind it.

Example 2. Matrix Multiplication

We also introduce 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 equally-sized 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 the whole result by using the products of submatrices, so we can apply the divide-and-conquer algorithm. This algorithm is often called cache-oblivious matrix multiplication.

Fig. 4.5 shows pseudocode of parallel cache-oblivious matrix multiplication. There are two task groups in a task (line 7-12 and line 14-19). `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). As explained above, the memory access order in consecutive task groups should be almost the same; in this case we should respect the order for matrix C because `tg1` and `tg2` do not share submatrices of A and B. As

```
matmul($A$, $B$, $C$) {
  if (size_of($C$) < CUTOFF) {
    $C$ += $AB$;
  } else {
    $\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} = A$, $\begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} = B$, $\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = C$

    task_group tg1(4);
    tg1.run([]{ matmul($A_{11}$, $B_{11}$, $C_{11}$); }, 1);
    tg1.run([]{ matmul($A_{21}$, $B_{11}$, $C_{21}$); }, 1);
    tg1.run([]{ matmul($A_{11}$, $B_{12}$, $C_{12}$); }, 1);
    tg1.run([]{ matmul($A_{21}$, $B_{12}$, $C_{22}$); }, 1);
    tg1.wait();

    task_group tg2(4);
    tg2.run([]{ matmul($A_{12}$, $B_{21}$, $C_{11}$); }, 1);
    tg2.run([]{ matmul($A_{22}$, $B_{21}$, $C_{21}$); }, 1);
    tg2.run([]{ matmul($A_{12}$, $B_{22}$, $C_{12}$); }, 1);
    tg2.run([]{ matmul($A_{22}$, $B_{22}$, $C_{22}$); }, 1);
    tg2.wait();
  }
}
```

Fig. 4.5.: Example code of cache-oblivious matrix multiplication with hints on the amount of work.

a result, workers in ADWS calculate nearly the same location of submatrices of C in consecutive task groups, which leads to good locality across consecutive task groups.

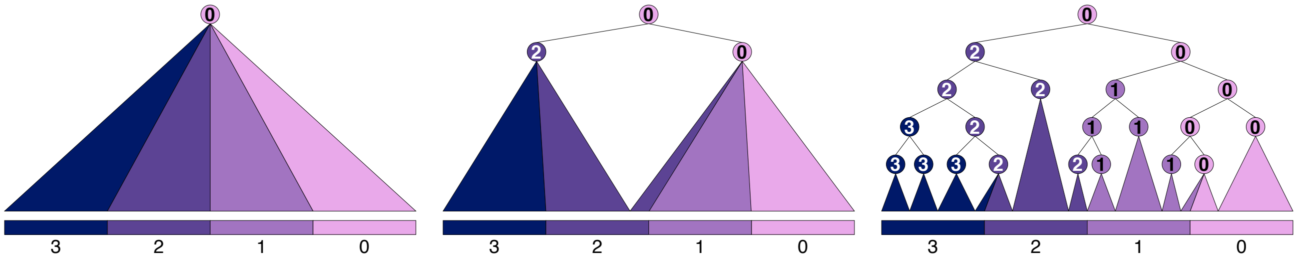
4.2.3. Deterministic Task Allocation

This section describes *deterministic task allocation*, which is the central part of ADWS. In deterministic task allocation, workers first determine the distribution of tasks so that the load is balanced, and then workers execute tasks allocated to themselves. The hints of work provided by the programmer are used to allocate the same amount of work to each worker. We can also view it as a static load balancing algorithm for nested fork-join programs.

Basic Idea

First, let us consider what we should do when a new task is spawned. ADWS schedules the spawned task and the continuation so that they complete in the same execution time; it allocates "an amount of workers" to them proportionally to the amount of work for each task [2]. We use the term "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 for tasks to be distributed, called *distribution ranges*. A distribution range is represented by two floating-point numbers, each of which is a point on the number line of workers (see Section 4.3). This idea of deterministic task allocation is illustrated in Fig. 4.6. The rectangles at the bottom represent the number line of workers, called "work region." We assume that only one task exists at first

[2] 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 nested parallel 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_∞ is the critical path [16]. If the parallelism (T_1/T_∞) is sufficiently larger than P, i.e., $T_1/T_\infty \gg P$, our assumption $T_p \approx T_1/P$ is true.



(a) Worker 0 is executing the root task whose distribution range covers all workers (i.e., descendants of the root task are distributed among all workers).

(b) Upon task creation, the distribution range is divided into two subranges in the ratio of work for the new task (left) and the continuation (right).

(c) Tasks are distributed recursively and in parallel. Each task is assigned to the worker of the smallest rank in its distribution range.

Fig. 4.6.: Overview of deterministic task allocation for four workers. The nodes are tasks, and the number on tasks shows which worker executes them. The triangles below tasks are distribution ranges, which cover the work region of the workers (the rectangles at the bottom).

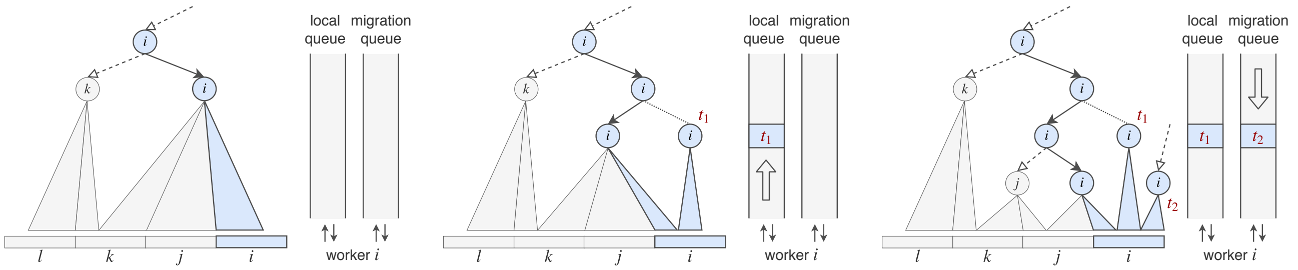
and worker 0 is executing the root task while the others are idle. The initial distribution range covers all the workers, because we want to distribute all descendant tasks evenly to all the workers. When a task is spawned, we divide the current distribution range into two subranges based on the amount of workers assigned to each task, calculated by the hint of work for the spawned task. Subsequently, the distribution range is recursively divided into small subranges.

Algorithm

In the following, we explain in detail the algorithm of deterministic task allocation, focusing on the working of each worker. In deterministic task allocation, each worker distributes tasks to others while executing the DAG. Conceptually, a worker dives into the bottom of the DAG to find the left boundary of its own work region. We call this step "search" as 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 whose distribution range lays across multiple workers. A task that has only one worker in its distribution range is called a "non-search-root task." Fig. 4.7 illustrates the search algorithm. First, worker i receives a search-root task from another worker. Then, worker i starts searching for the left boundary of its own work region. If the distribution range 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 distribution range. A search finishes when the task being executed is completed or when the worker encounters a *wait*.

We have assumed that worker 0 executes the root task initially, but this assumption may not be true in some implementations of parallel runtimes. 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 addition, a task can have multiple consecutive task groups, such the matmul example in Section 4.2.2). Because of that, 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 distribution range. Then, the owner starts a search from the next task group using the same distribution range as the previous task group. Fig. 4.2 helps us understand how tasks in consecutive task



(a) Worker *i* receives a search-root task and starts a search. When the distribution range is divided in the middle of worker *k*'s work region upon task creation, worker *i* migrates the new task to worker *k* and continues to execute the current task.

(b) When the distribution range is divided at the middle of worker *i*'s own work region, worker *i* puts the continuation of the current task into its local queue and starts to execute the new task.

(c) Non-search-root tasks may be migrated to worker *i*'s migration queue. When the *search* finishes, worker *i* first execute tasks in the local queue and then tasks in the migration queue.

Fig. 4.7.: Illustration of how deterministic task allocation works from an individual worker's perspective. The solid arrows represent the *search* path of worker *i*, and the dotted arrows represent task migration.

groups are scheduled.

Management of Tasks

Search-root tasks are put in a special buffer of a worker and can be executed immediately because there must be only one ready search-root task per worker at a time (∵ Theorem 2 in Appendix A). 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 (which are stored in the migration queue) are executed in FIFO order after all tasks in the local queue are completed. While a worker is not searching, tasks are executed by the *work-first* policy [24], in which a spawned task is first executed before the continuation (i.e., depth-first execution order). By following this execution order, we can preserve the serial execution order as much as possible, i.e., the execution order is from left to right in figures.

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. This is because the steal strategy in hierarchical localized work stealing can be simplified by splitting the queue. The local queue is used for the continuations generated during a search and their descendants, and the migration queue is for non-search-root tasks migrated from other workers and their descendants. Because migration of non-search-root tasks to a worker is serialized (∵ Theorem 1 in Appendix A), both of the queues can be implemented without locks. Although locks are needed for hierarchical localized work stealing, it is anticipated that lock contention is unlikely to happen. Thus, lock contention will not be a bottleneck during a search; initial distribution of tasks will be quickly determined.

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). A complex DAG is a DAG in which a nested task has multiple consecutive task groups in it. For example, parallel cache-oblivious matrix multiplication (Section 4.2.2) has a complex DAG. On the other hand, A *simple* DAG has at most one task group in a task (except for the root task),

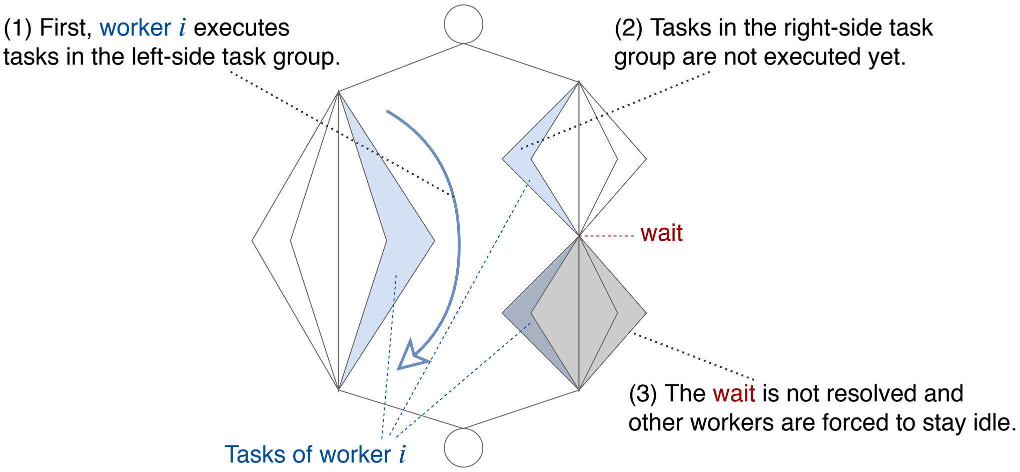

(1) First, worker *i* executes tasks in the left-side task group.
(2) Tasks in the right-side task group are not executed yet.
wait
(3) The wait is not resolved and other workers are forced to stay idle.
Tasks of worker *i*

Fig. 4.8.: Illustration of the problem with a complex DAG in deterministic task allocation.

such as the example of particle interactions in Section 4.2.2.

When dividing distribution ranges during a search, we allow a range to be divided in the middle of a work region, i.e., the amount of workers allocated to a task can be a floating-point number. Because of this, a worker often has to execute tasks from different task groups. This is not a problem with a simple DAG, but it can be a problem with a complex DAG (see Fig. 4.8). 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, searching for the left boundary of its work region. Then it executes the continuations of tasks put in the local queue during the search. The tasks in the right-side task group, which are migrated from other workers and stored in the migration queue, are executed after that. If there is a join point 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 that each worker is assigned almost the same amount of work, but it does not provide any proper execution order without idleness. 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 resolve this problem with hierarchical localized work stealing, which is described in the next section.

4.2.4. Hierarchical Localized Work Stealing

Hierarchical localized work stealing can resolve the idleness issue in deterministic task allocation for a complex DAG and also fix load imbalance dynamically. Load imbalance may appear because of imprecise hints on work for each task provided by the programmer. Even if the hints are precise, many factors at runtime (such as OS noise and CPU frequency scaling) can cause load imbalance.

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

Algorithm

For hierarchical localized work stealing, workers need to construct a *distribution tree* during a search in deterministic task allocation. Each node of a distribution tree contains three elements: a distribution range, a pointer to the parent node, and an active flag. When a worker encounters a task group during a search, the worker appends a new node to the tree; thus each node corresponds to a task group in a search-root task. Fig. 4.9 shows an example of a distribution tree. We denote a distribution range as $[i, j]$ $(i \leq j)$, where i and j are the worker ranks (integer values). In this representation we omit the fractional parts of endpoints of a distribution range for convenience.

During execution, each worker holds a reference for a node in the distribution tree, called *current node* of the worker. At the beginning of the execution, the current node of worker 0 has is the root node with range $[0, P-1]$, where P is the number of workers in the system. Here is the basic rules:

- The current node of each worker (except for worker 0) is first set to the current node of the worker who migrates a search-root task to it.
- A worker appends a new node to its current node when it encounters a task group during a search, updating its current node to the new one.
- During a search, a worker passes its current node to other workers with the migration of search-root tasks.
- When a search finishes, the current node is activated.
- When the task group corresponding to the current node is completed, the current node is deactivated and is set to its parent.
- When a search-root task is completed, the current node is activated.

When a node of a distribution tree is activated, its descendants should be deactivated. For example, in Fig. 4.9, if a node with range $[0, i]$ is deactivated and the parent ($[0, j]$) is activated by worker 0, the node with range $[i, j]$ should be also deactivated even if the node ($[i, j]$) 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 node from the bottom up every time the worker tries to steal. If the worker finds any active ancestors, the worker deactivates the current node and updates its current node to the topmost active ancestor.

When there is no local task, workers try to steal tasks from another worker if the current node is active. Victims are randomly chosen from the distribution range in the current node. Because a worker should not steal tasks outside of the task group of its current node, it must consider which queue they should steal a task from. For example, thieves whose current node has the distribution range $[j, k]$ (in Fig. 4.9) 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 node has the range $[j, k]$ 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 $[j, k]$.

It is not desired to steal tasks at the beginning of a search since it can harm deterministic task allocation, but it rarely happen if we follow the above rules. When a task group is completed, the current node is updated to the parent without activating it. Thus, at the entrance of the next task group, the current node is deactivated, preventing steals within the distribution range of the current node (Section 4.3.3 explains more details on management of a distribution tree). Thus, contention of the migration queue can be avoided (see also Theorem 1 in Appendix A), and the search can be done quickly. Though tasks can be stolen from outside of the distribution

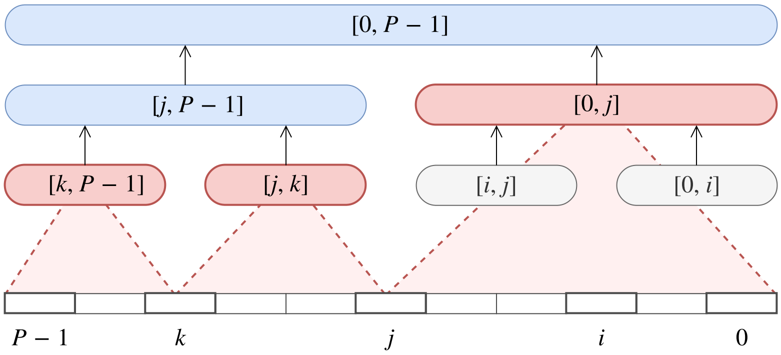


Fig. 4.9.: Example of a distribution tree. The rectangles with the rounded corners with a distribution range represent the nodes of the distribution tree. Nodes are activated from the bottom up, and finally the root node with range $[0, P-1]$ is activated. In this figure, the active nodes are $[0, j]$, $[j, k]$ and $[k, P-1]$.

range of the current node if an ancestor of the current node is activated during a search, it does not occur so frequently.

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 just describes the characteristics of ADWS. Investigating some theoretical proofs on ADWS would be an interesting direction of future work.

By following the hierarchical localized work stealing algorithm, it can be ensured that uncompleted tasks remain in the task group of the current node, and workers are expected to find uncompleted tasks in it. In other words, there are uncompleted tasks within the distribution range of the current node as long as the current node is active. If there is a long-running task in the task group, some of the workers may keep failing to steal tasks within the current distribution range, even though there are ready tasks outside of the current distribution range. Assuming that tasks are sufficiently fine-grained (which is usually true in recursive nested parallel programs), this is not a problem because workers should be able to find tasks within the current distribution range.

The scheduler is not greedy by that alone, because of search-root tasks. A search-root task cannot be stolen by anyone since it must be executed by the owner; hence, even when a worker is idle and there is a ready search-root task, it cannot executed by the worker if the worker is not the owner. It contradicts the definition of the greedy scheduler. 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 in Appendix A, and a worker 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 reaches a scheduling point (e.g., when it completes a task or reaches a join point), it becomes greedy enough in practice.

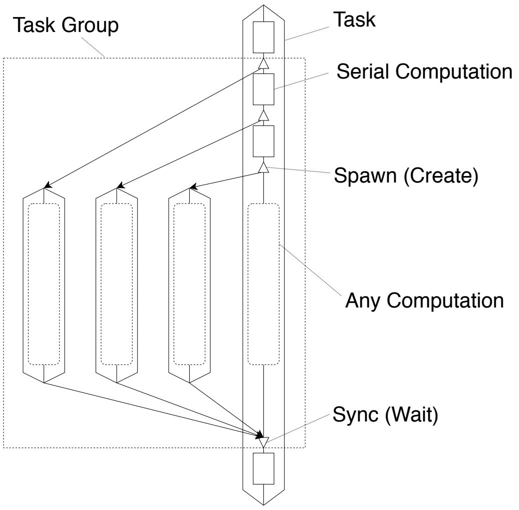


Fig. 4.10.: Notation of parallel computation used in this thesis.

4.3. Implementation over User-Level Threads

This section explains the detailed implementation of ADWS. Here we assume user-level threads as the underlying tasking system, but we believe the implementation explained below is also applicable to other types of task parallel runtimes for nested parallel computations. In the following, a *task* is a series of serial computation that can concurrently run with other tasks; a task can be implemented as a user-level thread.

4.3.1. Notation

Fig. 4.10 shows the notation for expressing nested parallel computations that ADWS targets for. A task can have any number of task groups, each of which consists of multiple *spawn* and *sync*. *Spawn* creates a new task that can be computed in parallel, and *sync* waits for completion of all tasks previously spawned by this task. A group of parallel computations from the first *spawn* to the corresponding *sync* is a task group, including descendant tasks. The *owner* of a task group is the task which creates the task group.

In nested parallel computations, task groups also have a nested structure. Fig. 4.11 shows DAGs which have nested task groups. The left DAG and the right one in Fig. 4.11 do the same computation, but the shape of the DAG is different because the right program does not spawn the last parallel computation as a new task. This is a well-known technique in expressing nested parallel computations to reduce the cost of task creation. However, this optimization can complicate the implementation of ADWS; with a reduced number of task creation, a task can be an owner of multiple task groups at a time, while in the all-spawn computation a task owns at most one task group at a time. Although restricting the programming model to the all-spawn model would much simplify the implementation of ADWS, this thesis explains the implementation that can deal with both cases.

Algorithm 4.1 shows type definitions and some worker-local variables used in ADWS. The main part of the ADWS implementation is shown in Algorithm 4.2 as pseudocode. To clearly distinguish assignment of values and pointers, “←” and “⇒” are used for the assignment notation for values and pointers, respectively. In the following, we will refer to Algorithm 4.1 and Algorithm 4.2 to explain the implementation of ADWS.

```
function A():
  task_group tg1;
  tg1.create([]{ A(); });
  tg1.create([]{ A(); });
  tg1.wait();
  task_group tg2;
  tg2.create([]{ A(); });
  tg2.create([]{ A(); });
  tg2.wait();
```

```
function B():
  task_group tg1;
  tg1.create([]{ B(); });
  B();
  tg1.wait();
  task_group tg2;
  tg2.create([]{ B(); });
  B();
  tg2.wait();
```

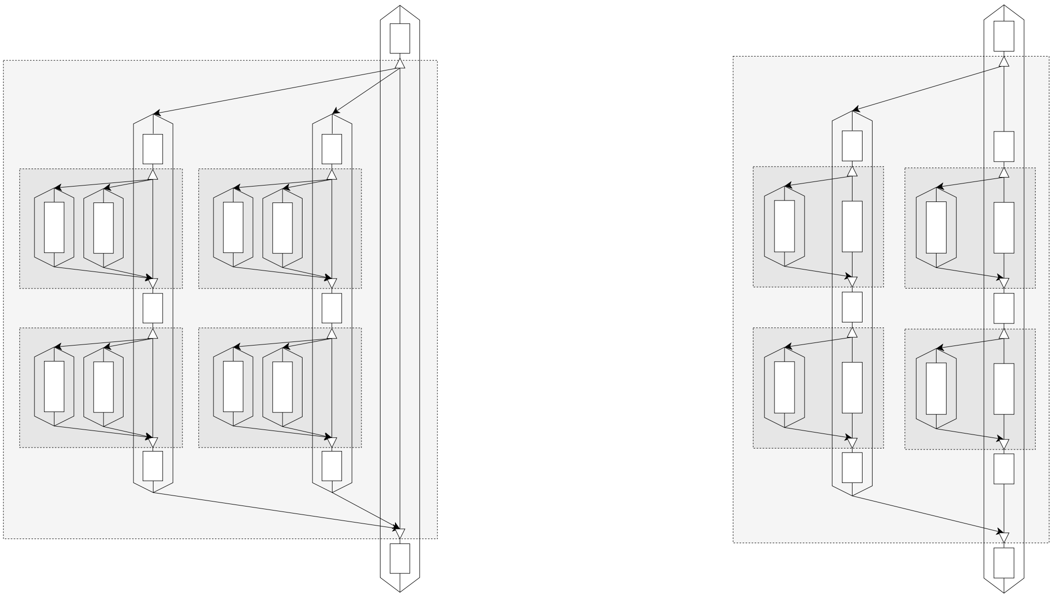


Fig. 4.11.: DAGs with nested task groups. While the left program spawns all parallel computations in task groups as new tasks, the right program does not spawn the last parallel computation in task groups.

Algorithm 4.1: Definitions in single-level ADWS.

```
Struct WSQueue
    Function Push(task :: Pointer to Task) :: nil           ▷ Push a task to the local end of the queue
    Function Pop() :: Pointer to Task                       ▷ Pop a task from the local end of the queue
    Function Pass(task :: Pointer to Task) :: nil           ▷ Push a task to the non-local end of the queue
    Function Take() :: Pointer to Task                      ▷ Pop a task from the non-local end of the queue
Struct DistributionRange
    from :: double
    to :: double
Struct DistributionTree
    drange :: DistributionRange
    parent :: Pointer to DistributionTree
    active ← false :: boolean
Struct Task
    lambda :: lambda
    work :: double
    drange :: DistributionRange
    dtree :: Pointer to DistributionTree
Struct TaskGroup
    tasks ← [] :: List of pointers to Task
    drange :: DistributionRange
    dtree :: Pointer to DistributionTree
    isSearchRoot :: boolean
Struct Worker
    rank :: integer
    dtree :: Pointer to DistributionTree
    searchRootTask :: Pointer to Task
    localQueue :: WSQueue
    migrationQueue :: WSQueue
    useMigrationQueue ← false :: boolean
Worker-Local Variables
    myWorker :: Pointer to Worker                           ▷ A pointer to the worker-local data
    myTask :: Pointer to Task                               ▷ The currently running task on this worker
```

Algorithm 4.2: Algorithm of single-level ADWS.

```
Function TaskGroup::Init($work$)
    $myTask.work \leftarrow work$
    $this.drange \leftarrow myTask.drange$; $this.dtree \Rightarrow myTask.dtree$
    $this.isSearchRoot \leftarrow$ whether $myTask.drange$ is laid across multiple workers (i.e.,
      $\lfloor myTask.drange.from \rfloor \neq \lfloor myTask.drange.to \rfloor$)
    if $this.isSearchRoot$ then
        $dtree \Rightarrow$ new DistributionTree($drange \leftarrow myTask.drange, parent \Rightarrow myTask.dtree$)
        $myTask.dtree \Rightarrow dtree$; $myWorker.dtree \Rightarrow dtree$
Function TaskGroup::Create($lambda, work$)
    $newTask \Rightarrow$ new Task($lambda \leftarrow lambda, dtree \Rightarrow myTask.dtree$)
    $myTask.work \leftarrow myTask.work - work$
    $(myTask.drange, newTask.drange) \leftarrow$ divide $myTask.drange$ in the ratio $myTask.work : work$
    $targetWorker \Rightarrow$ the worker of rank $\lfloor newTask.drange.from \rfloor$
    if $targetWorker = myWorker$ then
        Push $myTask$ into $myWorker$'s local queue and run $newTask$
    else
        if $newTask.drange$ is laid across multiple workers then
            Pass $newTask$ to $targetWorker$ as a search-root task
        else
            Pass $newTask$ to $targetWorker$ as a non-search-root task
    Add $newTask$ to $this.tasks$
Function TaskGroup::Wait()
    if $myTask.drange$ is laid across multiple workers then
        $myTask.dtree.active \leftarrow$ true
    foreach $task \in this.tasks$ do
        Wait for completion of $task$
    if $this.isSearchRoot$ then
        $ownerWorker \Rightarrow$ the worker of rank $\lfloor myTask.drange.from \rfloor$
        Pass $myTask$ to $ownerWorker$ as a search-root task and go to the scheduler
        $myWorker.dtree \Rightarrow myTask.dtree.parent$
        delete $myTask.dtree$
    $myTask.drange \leftarrow this.drange$; $myTask.dtree \Rightarrow this.dtree$
Function OnTaskCompletion($task$)
    if $task.drange$ is laid across multiple workers then
        $task.dtree.active \leftarrow$ true
Function Scheduler()
    while true do
        $task \Rightarrow$ Try to pop a $myWorker$'s local task
        if $task = NULL$ then $task \Rightarrow$ Try to steal a task from others
        if $task \neq NULL$ then Run $task$
```

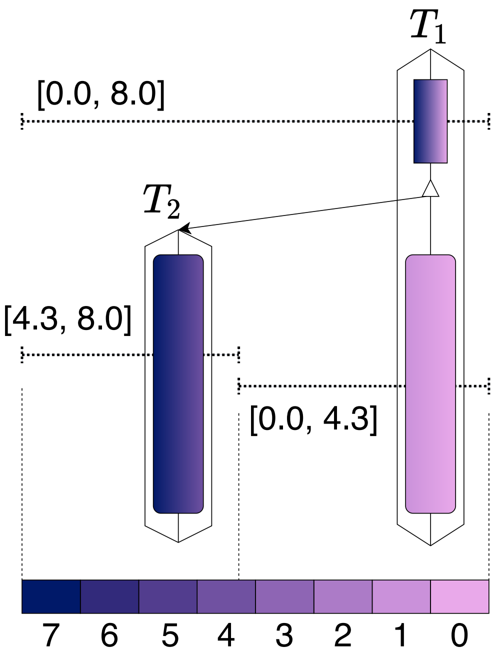


Fig. 4.12.: Example of division of distribution ranges at task creation.

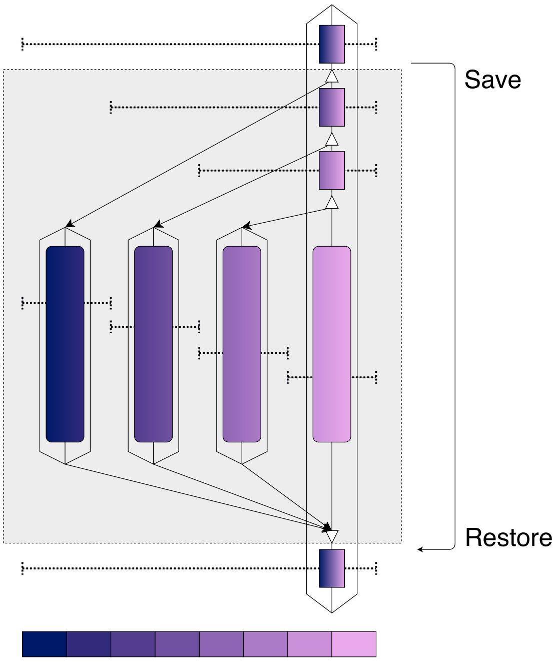


Fig. 4.13.: Distribution ranges in a task group.

4.3.2. Management of Distribution Ranges

First, we describe the management of distribution ranges, which are used for deterministic task allocation. A distribution range consists of two floating-point numbers, $from$ and to (line 6–8 in Algorithm 4.1), to represent the range $[from, to]$. As illustrated in Fig. 4.12, upon task creation, a distribution range of a task is divided into two subranges in the ratio of work for the new task and the continuation of the current task. In this figure, the current task (T_1) originally has the distribution range $[0.0, 8.0]$, which is divided into $[0.0, 4.3]$ and $[4.3, 8.0]$ at task creation. The new task (T_2) is assigned the larger part of the distribution range ($[4.3, 8.0]$) and is migrated to the owner worker of this range (worker of rank $\lfloor 4.3 \rfloor = 4$). Since the distribution range of T_2 is laid across multiple workers ($\lfloor 4.3 \rfloor \neq \lfloor 8.0 \rfloor$), T_2 is passed as a search-root task. If the owner worker of the distribution range of T_2 is the same as that of T_1, the worker puts T_1 into the local task queue and starts to execute T_2 first (the *work-first* policy). Line 12–19 in Algorithm 4.2 corresponds to the above explanation.

Work for tasks is managed as follows. In Algorithm 4.2, the total work for the task group (given by the programmer) is assigned to the owner task at line 2. Work associated with the owner task decreases as new tasks are created; at task creation, work for the new task (given by the programmer) is subtracted from the owner task's work and the distribution range is divided in the ratio of work for the task and the continuation of the owner task (line 10–11). Here we do not assign $work$ to the new task because it will not be used; the new task will soon encounter the next task group in which its total work will be assigned to the task.

Fig. 4.13 is an overall picture of the distribution range management for a task group. The distribution range of the owner task of the task group is getting smaller as new tasks are spawned. The distribution range of a task right before it encounters a task group is saved at the beginning of the task group (line 3 in Algorithm 4.2) and restored at the end of the task group (line 31), so that the next task group is scheduled under the same distribution range.

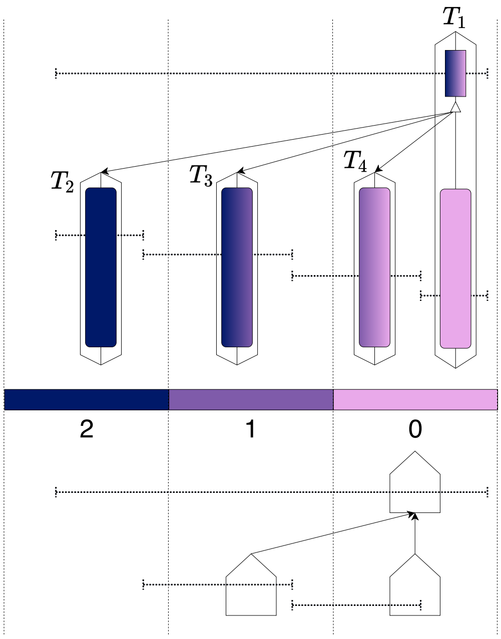


Fig. 4.14.: Illustration of construction of a distribution tree.

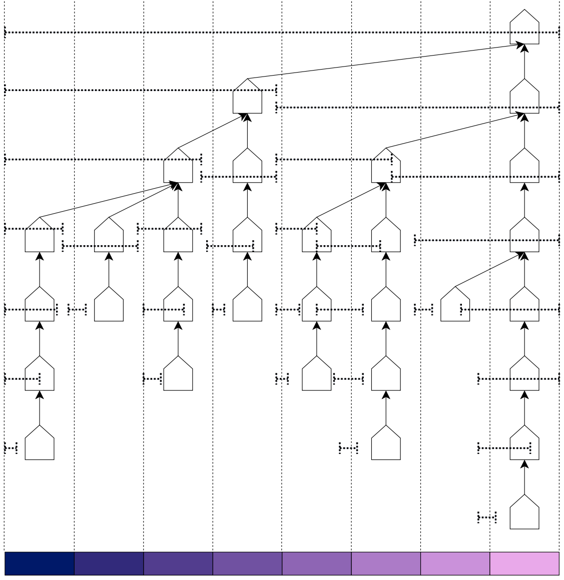


Fig. 4.15.: Overview of a distribution tree.

4.3.3. Management of a Distribution Tree

A distribution tree is referenced when performing hierarchical localized work stealing to limit the range of workers to steal tasks from. Each node in a distribution tree has the range of workers for stealing, which is represented by a distribution range created during deterministic task allocation. A distribution tree does not cover all distribution ranges created on deterministic task allocation but covers only distribution ranges of search-root tasks. Fig. 4.14 shows how a distribution tree is constructed. The owner task of the task group (T_1) creates three tasks T_2, T_3, T_4. In this figure, only T_3 and T_4 are search-root tasks since their distribution ranges are laid across multiple workers. T_1 was initially a search-root task but is no longer a search-root task after spawning three child tasks. When a search-root task encounters a task group, it creates a new distribution tree node associated with the current distribution range, pointing to the parent distribution tree node. In other words, a node in the distribution tree corresponds to a task group in a search-root task. In the figure, T_1 creates a distribution tree node with the initial distribution range, and then T_3 and T_4 create distribution tree nodes when they encounter task groups, pointing to the distribution tree node created by T_1. The rest of T_1 and T_2 do not create distribution tree nodes because they are non-search-root tasks. When a task group in a search-root task is completed, the distribution tree node for the task group is removed from the distribution tree.

Let us explain more details for maintaining the distribution tree. A task is assigned a pointer to a distribution tree (more precisely, a subtree of the overall distribution range) of the task group that it belongs to, regardless of whether it is search-root or not. At the entrance of a task group, a search-root task is assigned a pointer to the newly allocated distribution tree (line 6–7 in Algorithm 4.2), and then it is passed to the child tasks at line 9. At distribution tree creation, because the parent of the new distribution tree is the distribution tree currently assigned to the current task, we simply make the new distribution range point to the current distribution range and overwrite the current distribution range with the new one. By following this procedure, the overall distribution tree will look like Fig. 4.15. The vertical alignment of the distribution tree nodes represents which worker creates them.

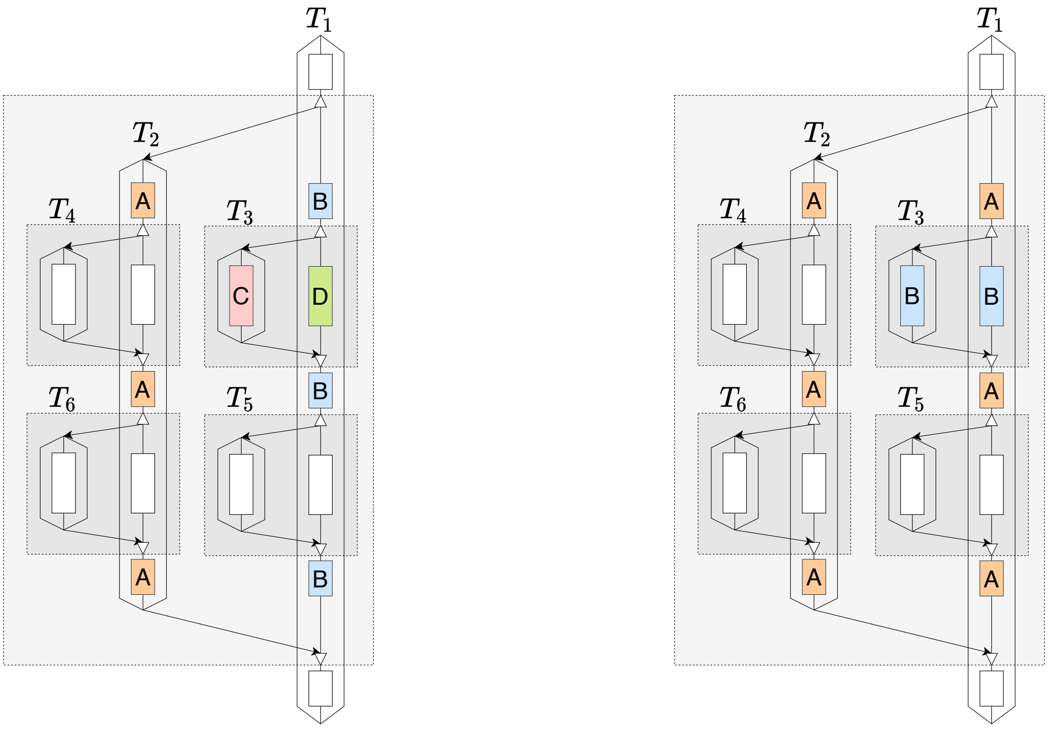
Also, workers should have a reference to distribution tree nodes to be used for work stealing. At the beginning of a task group, the owner worker of a search-root task is assigned the new distribution tree node (line 7 in Algorithm 4.2). When the task group is completed, the pointer to a distribution tree is updated to the parent of the removed distribution tree node (line 29).

Next we explain how distribution tree nodes are activated. As we will describe later, the active node that is the closest to the root of the distribution tree, among nodes that can be reached from the worker's current reference to a distribution tree, has the distribution range from which the worker should steal tasks. Initially, all distribution tree nodes are deactivated upon creation. Distribution tree nodes are activated when a search-root task reaches *sync* (line 22–23 in Algorithm 4.2) or when a search-root task is completed (line 33–34). In Fig. 4.14, the parent distribution tree node is activated when T_3 is completed, T_4 is completed, or when worker 0 reaches *sync* of the corresponding task group in T_1. A distribution tree node is deactivated again when the corresponding task group is completed and the distribution tree node is removed (line 30).

Allocation and deallocation of distribution trees, denoted by *new* (line 6) and *delete* (line 30) in Algorithm 4.2, are not those in the semantics of C++ but are implemented by using *freelists* implemented by our own. This is because deallocated distribution tree nodes can be referenced by other workers even after deallocation, which will result in undefined behaviour. A worker's reference to a distribution tree is updated to the parent when a task group is completed if the task group is created by the worker, but if not, the reference is not updated. In Fig. 4.15, within a worker, a worker-local reference points to only locally-allocated distribution tree nodes or the parent of the top local node. Therefore, every time a worker tries to steal tasks, deallocated distribution tree nodes, that are ancestors of the node pointed to by the worker-local reference, can be referenced. This issue can be solved by reference counting, but it requires atomic operations, which can cause large overheads at runtime. Garbage collection can be another solution, but it would be overkill.

Thus, we decided to simply use a freelist to allocate/deallocate distribution tree nodes within a worker. Distribution tree nodes within a worker are managed as a linked list (called a freelist), which works as a stack. In Fig. 4.15, nodes are allocated from the top down and deallocated from the bottom up (i.e., LIFO order) within a worker. If there is no node in a freelist on allocation, the worker allocates one by using library calls such as `malloc()`. On deallocation, a node is added to a freelist and never returned to the external allocator in the current implementation for simplicity. A distribution tree node is allocated and deallocated by the same worker because at the end of the task group the owner task is migrated to the same worker (line 28 in Algorithm 4.2) and then the distribution tree node is deallocated. A pointer to the parent of a node is not cleared on deallocation so that descendants of the node can reach to the root of the distribution tree even after deallocation. By doing so, the parent-child relationship of nodes within a worker never changes, although the parent of the top node within a worker can change.

The distribution tree is always a tree (does not have a loop) at any time in execution, because (i) nodes allocated in each worker forms a linked list, (ii) the parent-child relationship of nodes within each worker never changes, and (iii) only the top node within a worker has a pointer to nodes in other workers which have smaller ranks. Therefore, the operation of obtaining the active node that is the closest to the root does not fall into a infinite loop. Under severe race conditions (e.g., a deallocated node is allocated again while other workers have a reference to it), a worker might choose a wrong distribution range to steal tasks from, but at least it does not cause a crash or deadlock. It might steal inappropriate tasks, but search-root tasks cannot be stolen because of the task queue management described later, which guarantees that the program should not crash.



(a) Distribution range

(b) Distribution tree

Fig. 4.16.: Illustration of task-local data management for distribution ranges and distribution trees. The same characters (A, B, C, D) represent the same data (or pointer). Both of the data for distribution ranges and distribution trees are saved at the beginning of the task group and restored at the end of the task group.

4.3.4. Clarification of the Difference between Distribution Ranges and Distribution Trees

Distribution ranges and distribution trees might be confusing, as both are used for expressing the range of workers. First, their purpose is different; distribution ranges are used for deterministic task allocation, and distribution trees are for hierarchical localized work stealing, ignoring the fact that a distribution tree node includes a distribution range from which workers steal tasks from. Second, distribution ranges are just stack-allocated values, but distribution trees are heap-allocated objects so that workers can access them even after the corresponding task groups are destroyed. Finally, how they are associated to tasks is different, as we describe in the following.

Fig. 4.16 shows how distribution ranges and distribution trees are associated with tasks. The same characters in the sequential parts in the tasks represent the same data or pointer associated with the tasks. A distribution range associated with a task represents *the range of workers who should execute the continuation of this task and its descendants until the next sync on this task*. Thus, tasks in a task group have different distribution ranges; for example, (A, B) and (C, D) in Fig. 4.16a. B is divided into C and D when T_1 spawns T_3, and B is associated to T_1 again when the task group is completed.

On the other hand, a distribution tree associated with a task corresponds to *the task group at the deepest nesting level among those this task belongs to*. Thus, tasks in a task group are initially associated with the distribution tree created at the beginning of the task group. For example, in Fig. 4.16b, T_1 and T_2 are initially associated with the distribution tree A, and once T_1 creates the nested task group and spawns T_3, it is associated with the new distribution tree B. After the nested task group is completed, A is associated with T_1 again. By doing so, constructing the overall distribution tree is much simplified and activation of distribution trees can be easily done.

4.3.5. Task Queue Management

In the implementation of ADWS, a worker has two kinds of work stealing queues: a *local queue*, which stores stolen tasks and the continuations of tasks left during deterministic task allocation including their descendants, and a *migration queue*, which stores tasks migrated to the worker during deterministic task allocation including their descendants. They are distinguished for preventing inappropriate steals; in other words, stealing tasks outside of the targeting task group. If the distribution range of a distribution tree is $[from, to]$, tasks of the corresponding task group and their descendants exist only in (i) the local queue of worker $\lfloor from \rfloor$, (ii) the migration queue of worker $\lfloor to \rfloor$, and (iii) both queues of other workers within the range, except for the case where one of the ancestors of the distribution tree is activated and tasks in the targeting task group have been already stolen by outside workers. In fact, this exceptional case is not a problem because workers will anyway choose the active distribution tree that is the closest of the root when trying to steal tasks.

Another concern is that workers might steal tasks from the ancestors of the targeting task group. For example, when the distribution range of the targeting task group is $[from1, to]$ and the parent has the range $[from2, to]$, the migration queue of worker $\lfloor to \rfloor$ has tasks from both task groups. Therefore, even if workers are trying to steal tasks under the task group with the range $[from1, to]$, they might steal tasks under the range $[from2, to]$ that are stored in the migration queue of worker $\lfloor to \rfloor$. Nevertheless, we do not consider this problem so seriously because the distribution range of the targeting task group is included in that of the parent ($[from1, to] \subset [from2, to]$), so stolen tasks from the parent task group are anyway to be consumed by them. This problem would be resolved by splitting the task queues more, but for now we leave it for future work. On the other hand, splitting the local queue and the migration queue is way more important because distribution ranges of tasks in the local queue and the migration queue are not overlapped.

Search-root tasks are never stored in the local queue or the migration queue. They are stored in a special buffer for them (line 26 in Algorithm 4.1). Because it is guaranteed that only one search-root task is migrated to a worker before the worker takes it, only one pointer for search-root tasks per worker is sufficient. In the current implementation search-root tasks are not allowed to be stolen, so migration of search-root tasks does not require any lock.

Algorithm 4.3 shows the detailed implementation regarding task queue management that is omitted in Algorithm 4.2. PushTaskToLocal() function (line 1–5) and PopTaskFromLocal() function (line 6–22) are called when pushing the continuation on task creation (line 14 in Algorithm 4.2) and popping local tasks in the scheduler or inside of the task parallel runtime (not shown in this thesis; e.g., inside of *sync*). The flag $useMigrationQueue$ (line 29 in Algorithm 4.1) is used to switch the local queue and migration queue. In PopTaskFromLocal() function, the worker first checks the buffer for search-root tasks because they are the most prioritized. Then it checks the local queue and the migration queue in turn.

PassSearchRootTask() function (line 23–24 in Algorithm 4.3) is called when a search-root task is created (line 17 in Algorithm 4.2) and when a search-root task is returned to the owner worker at the end of a task group (line 28 in Algorithm 4.2). PassNonSearchRootTask() function (line 25–26 in Algorithm 4.3) is called when non-search-root tasks are created and migrated to other workers (line 19 in Algorithm 4.2).

In StealTask() function (line 27–43 in Algorithm 4.3), the worker first gets the active distribution tree that is the closest to the root (line 28–32). If no active distribution tree is found, it does not steal tasks. If the

Algorithm 4.3: Task queue management in single-level ADWS.

```
Function PushTaskToLocal($task$)
    if $myWorker.useMigrationQueue$ then
        $myWorker.migrationQueue$.Push($task$)
    else
        $myWorker.localQueue$.Push($task$)
Function PopTaskFromLocal()
    $task \Rightarrow myWorker.searchRootTask$
    if $task \neq NULL$ then
        $myWorker.searchRootTask \Rightarrow NULL$
        $myWorker.useMigrationQueue \leftarrow$ false
        return task
    if $myWorker.useMigrationQueue$ then
        $task \Rightarrow myWorker.migrationQueue$.Pop()
        if $task = NULL$ then
            $task \Rightarrow myWorker.localQueue$.Pop()
            $myWorker.useMigrationQueue \leftarrow$ false
    else
        $task \Rightarrow myWorker.localQueue$.Pop()
        if $task = NULL$ then
            $task \Rightarrow myWorker.migrationQueue$.Pop()
            if $task \neq NULL$ then $myWorker.useMigrationQueue \leftarrow$ true
    return task
Function PassSearchRootTask($task, targetWorker$)
    $targetWorker.searchRootTask \Rightarrow task$
Function PassNonSearchRootTask($task, targetWorker$)
    $targetWorker.migrationQueue$.Pass($task$)
Function StealTask()
    $topdtree \Rightarrow NULL$
    $dtree \Rightarrow myWorker.dtree$
    while $dtree \neq NULL$ do
        if $dtree.active$ then $topdtree \Rightarrow dtree$
        $dtree \Rightarrow dtree.parent$
    if $topdtree \neq NULL$ then
        $from \leftarrow \lfloor topdtree.drange.from \rfloor, to \leftarrow \lfloor topdtree.drange.to \rfloor$
        $targetRank \leftarrow$ Random number in the range $[from, to]$ except for $myWorker.rank$
        $targetWorker \Rightarrow$ the worker of rank $targetRank$
        if $targetRank \neq from$ then
            $task \Rightarrow targetWorker.migrationQueue$.Take()
            if $task \neq NULL$ then return task
        if $targetRank \neq to$ then
            $task \Rightarrow targetWorker.localQueue$.Take()
            if $task \neq NULL$ then return task
    return $NULL$
```

CPU model	# of sockets	# of cores	frequency	L1 data cache	L2 cache	L3 cache
Xeon Gold 6130 (Skylake)	4	64 (16 × 4)	2.1 GHz	32 KB/core	1 MB/core	22 MB/socket

Tab. 4.1.: CPU information used in the experiments.

distribution range of the top active distribution tree is $[from, to]$, the worker first chooses the victim worker from the range $[\lfloor from \rfloor, \lfloor to \rfloor]$ randomly. As we mentioned earlier, the worker can steal tasks from (i) the local queue of worker $\lfloor from \rfloor$, (ii) the migration queue of worker $\lfloor to \rfloor$, and (iii) both queues of other workers within the range (line 34–42).

4.4. Performance Evaluation

We conducted experiments in a NUMA architecture to compare the performance of ADWS with 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 4.1.

We implemented ADWS on *MassiveThreads* [36], a lightweight user-level threading library. The default scheduler implemented in MassiveThreads was random work stealing with the *work-first* scheduling policy. We also implemented *hierarchical work stealing* (*hierarchical WS*) [64], [73] as an existing locality-aware scheduling strategy for deep memory hierarchy. In hierarchical WS, workers first try to steal tasks from the nearest workers in the same memory hierarchy, and after several attempts, they try to steal from a broader range of workers. This scheduling strategy was proposed in [64], and work in [73] 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 L3 caches 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`.

4.4.1. 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.

Heat2D uses two buffers, *new* and *old*; *new* buffer is used for writing newly calculated values and *old* buffer for reading the values in the last iteration. It swaps *new* buffer and *old* buffer at every iteration. We parallelized heat2D by using the divide-and-conquer algorithm, which divides the region into four square subregions recursively, and the cutoff size was set to 64×64. 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; they used 32 MB, 128 MB, and 512 MB of memory, respectively. We measured the execution time for 1,000 iterations, and the experimental results were the median of ten executions.

For comparison, we implemented *constrained work stealing* (constrained WS) [65], 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 [74]. 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.

Fig. 4.17 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 Fig. 4.17, 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.

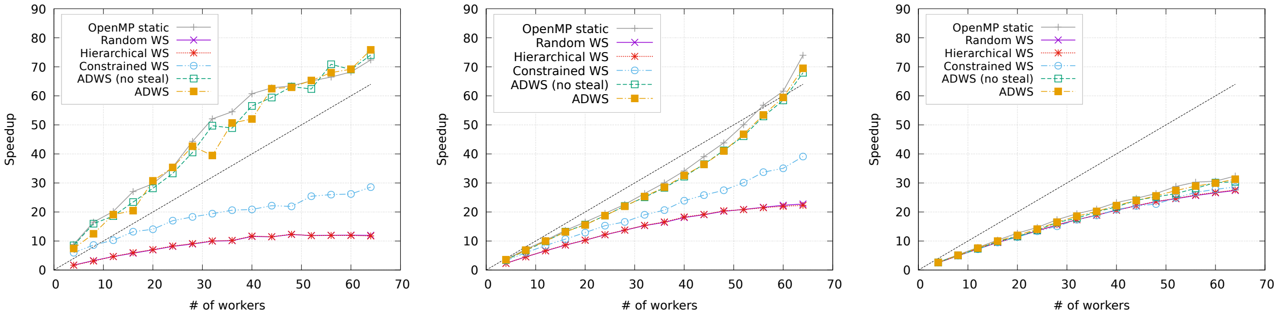
There are two reasons for the performance difference between Constrained WS and ADWS. One is that the execution of the first iteration in constrained WS was scheduled by random WS, and data locality among tasks within a worker or socket was not optimized. The other one is increased idleness of workers in constrained WS. 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 4.2.4). The more detailed discussion of the performance of constrained WS with additional results of performance evaluations is included in Appendix B.

In Fig. 4.17a, 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.

Fig. 4.17b 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.

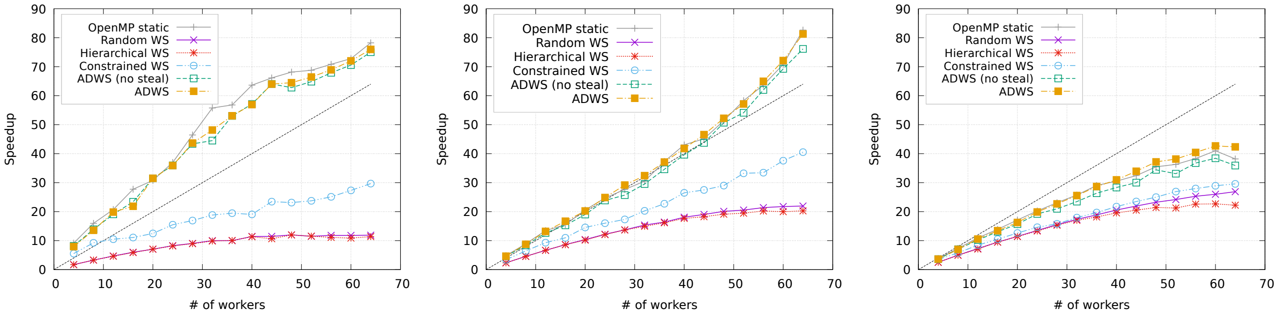
When $N = 8192$, the performance difference became small (Fig. 4.17c). This was because the data (512 MB) did not fit into the caches and accesses to DRAM occurred frequently. In this case, the physical location of data was determined by `numactl -iall` policy, and many remote memory accesses occurred 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.

Fig. 4.18 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 Fig. 4.17. The performance improvement clearly appears in Fig. 4.18c compared to Fig. 4.17c. 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. Concerning the performance improvement of ADWS to random WS, Fig. 4.18a shows that ADWS was up to nearly six times faster than random WS.



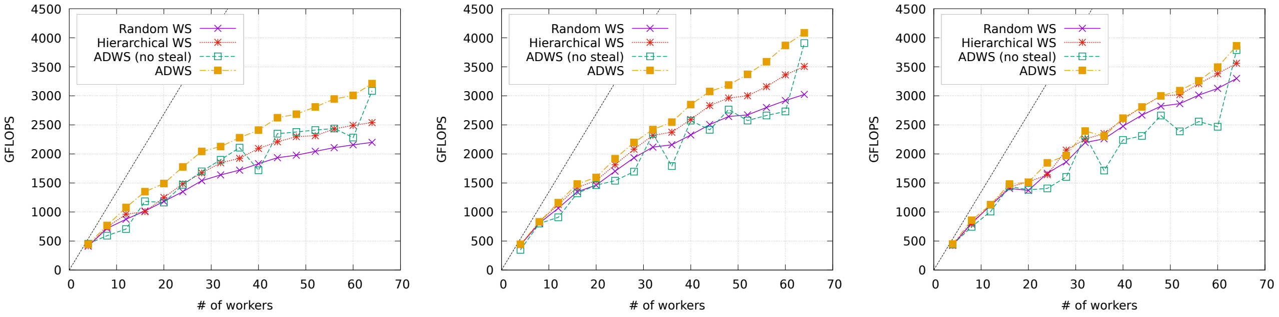
(a) N = 2048 (b) N = 4096 (c) N = 8192

Fig. 4.17.: Comparison of the speedup of heat2D benchmark with `numactl -iall`. The straight dotted line represents the ideal linear speedup.



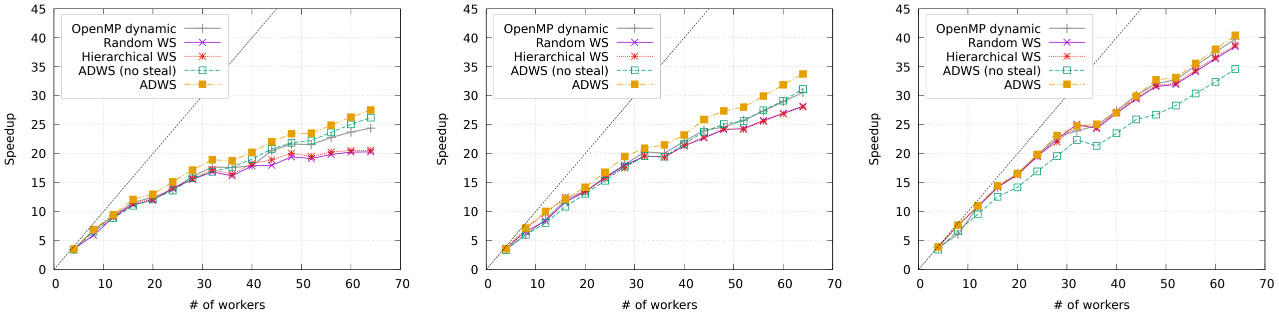
(a) N = 2048 (b) N = 4096 (c) N = 8192

Fig. 4.18.: 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

Fig. 4.19.: 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

Fig. 4.20.: Comparison of speedup of 2D and 3D dam breaking simulation using FDPS. The straight dotted line represents the ideal linear speedup.

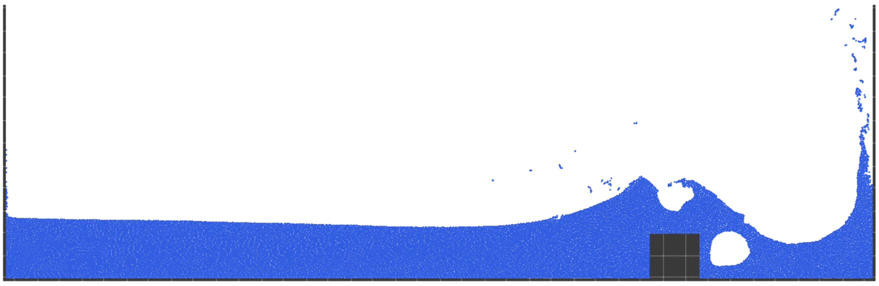


Fig. 4.21.: 2D dam breaking simulation with SPH.

4.4.2. 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 based on the parallel cache-oblivious divide-and-conquer algorithm as described in Section 4.2.2. 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 128×128. We optimized the calculation kernel by using SIMD instructions (AVX-512) to be able to achieve performance comparable to the machine's peak FLOPS. The experimental results were the median of ten executions.

Fig. 4.19 shows a comparison of 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 Fig. 4.19, 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 among them. Notably, ADWS achieved 4085 GFLOPS with $N = 4096$, which is nearly half of the theoretical peak FLOPS with 64 cores (8601.6 GFLOPS).

The performance of ADWS (no steal) was poor because of the problem with a complex DAG. 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 at the shallow parts of the DAG; thus workers were not forced to stay idle for a long time. The result shows that even if deterministic task allocation had a problem, ADWS performed well thanks to hierarchical localized work stealing.

4.4.3. Calculation of Particle Interactions (SPH)

We also compared the performance of ADWS with that of the existing implementation of FDPS [71], a framework for developing parallel particle simulation codes. This evaluation focuses on simulation of *Smoothed Particle Hydrodynamics (SPH)*, which calculates short-range interactions of particles within an effective radius at each time step. In the same way as described in Section 4.2.2, FDPS manages particles as an octree. First, the original implementation of FDPS makes an array of leaf nodes by traversing the octree and then applies OpenMP's parallel `for` loop to the array of leafs. Because the amount of work for each leaf can vary and thus static scheduling can cause significant load imbalance, FDPS uses a dynamic scheduling policy in OpenMP

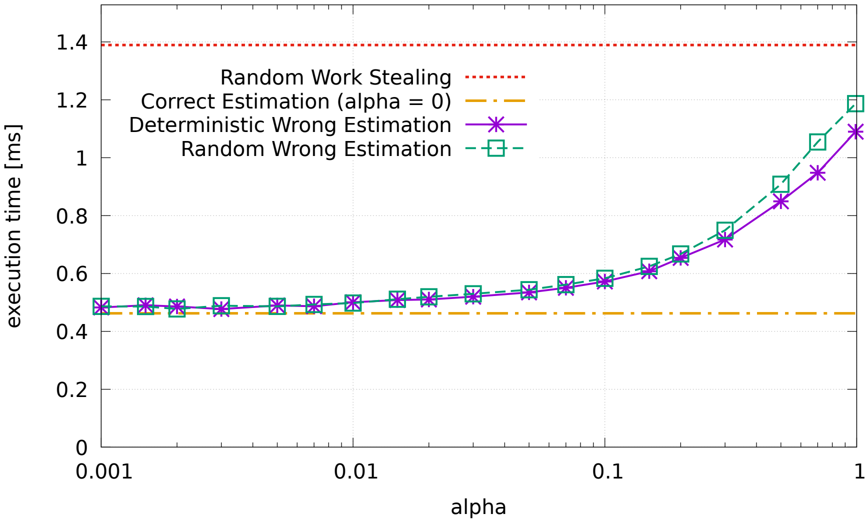


Fig. 4.22.: Mean execution time of one iteration in heat2D benchmark for 10k iterations. *alpha* means the maximum error ratio of an estimated amount of work for each task.

(`schedule(dynamic, 4)`). Section 4.2.2 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 wrote SPH code over FDPS to simulate the dynamics of water (dam breaking), the visualization of which is shown in Fig. 4.21. We referenced Becker and Teschner's work [75] to implement the computation kernel. We also modified the internal code of FDPS (v5.0b) to implement the task parallel version of calculation of particle interactions. In this study, we only compared the performance of the calculation of particle interactions, although we can parallelize other parts such as building octrees by using task parallelism as ExaFMM does [76].

Fig. 4.1 shows visualizations of the task mapping to cores. 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. 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.

Fig. 4.20 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 1,000 iterations, and the result was the median of ten executions. We conducted this experiment with `numactl -iall`, and the base case was the execution time of serial execution without OpenMP nor MassiveThreads.

Fig. 4.20a and Fig. 4.20b show the speedup of the 2D simulation. In both cases, the performance of ADWS was better than that of others. Fig. 4.20c 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.

4.4.4. 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 α, which meant the maximum error

Tab. 4.2.: Mean execution time of fib benchmark for 10 times.

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

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 α. Fig. 4.22 shows the result, showing that the execution time increased as α increased. Within $\alpha < 10\%$, the performance did not degrade more than about 30%. Even when $\alpha = 100\%$, some of data locality remained and the performance was better than that of random WS.

4.4.5. 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 Tab. 4.2. 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 4.2.4). In random WS, the initial distribution of tasks was relatively slow because of lock contention.

4.5. Summary

We introduced Almost Deterministic Work Stealing (ADWS) and showed ADWS outperformed other existing scheduling methods with locality-sensitive applications. Although the applications of ADWS are limited to cases in which the programmer can specify the amount of work for each task, performance was improved significantly if the amount of work was correctly specified. Specifying the amount of work is certainly a burden on the programmer, but it is not hardware-specific and the program can be oblivious to specific machine configurations. Even if the hints are just rough estimates, hierarchical localized work stealing can dynamically fix load imbalance in a locality-aware manner.

One of the remaining problems in ADWS is that it is oblivious to the cache size. As indicated in the evaluation (Section 4.4), ADWS did not perform well for large problem sizes that do not fit into the cache. This is because task groups that do not fit into the cache can be scheduled onto the cache simultaneously, causing many shared cache misses. The next chapter addresses this issue by extending ADWS to perform cache-size-aware scheduling.

5. Extending ADWS to Be Cache-Size-Aware with Multi-Level Scheduling

This chapter introduces an improved design of ADWS, called *multi-level ADWS*, that performs cache-size-aware scheduling.

5.1. Introduction

The previous chapter (Chapter 4) introduced Almost Deterministic Work Stealing (ADWS), which improves data locality of nested parallel computations. ADWS addresses two types of data locality in nested parallel computations: data locality in close tasks in a DAG and data locality in iterative parallel programs, as explained in Section 2.3.2. First, ADWS maps tasks to workers so that workers execute tasks as distant from each other as possible, preventing excessive data sharing among workers. At the same time, tasks are distributed so that each worker locally executes close tasks in a DAG, improving cache locality within a worker. Second, the task mapping is almost deterministic, which enables exploiting data locality in iterative parallel programs because at every iteration workers are expected to touch the same location of the data. One limitation of ADWS is that the programmer has to specify hints on the amount of work for each task. The hints can be a rough estimate since ADWS also performs dynamic load balancing, which is why ADWS is "almost" deterministic.

However, the evaluation results in Section 4.4 indicate that ADWS does not perform well for programs with large data which does not fit into the cache. This is because ADWS is oblivious to the size of shared caches, which often causes many cache misses in shared caches. ADWS performs well for private caches when the overall data fits into the shared cache because workers execute as distant tasks as possible; however, this scheduling is not optimal for shared caches in computations with larger data. For shared caches, it is important to execute subcomputations that fit into the shared cache one by one, promoting shared cache reuse within each subcomputation. ADWS, on the other hand, tries to schedule the entire computation at one time, causing many shared cache misses.

Until now, many task schedulers for improving data locality in shared caches have been studied. Parallel depth first (PDF) scheduler [29], [30] is one of the most earliest schedulers designed for shared cache. PDF scheduler maps close tasks in a DAG to workers, reducing space usage by workers and promoting utilization of the shared cache; more details about PDF scheduler is explained in Section 2.3.4. As PDF scheduler only considers a single shared cache, many shared-cache-size-aware schedulers for more complicated cache hierarchy have been proposed: schedulers for machines with private caches and a single shared cache [77], [78], schedulers for multi-socket multi-core architectures [66]–[68], and schedulers for arbitrary memory hierarchy called *Space-Bounded (SB) schedulers* [6], [79]–[83]. Their common idea is to limit the number of tasks that can be scheduled under a shared cache at the same time by explicitly considering data sizes of tasks, which are obtained via online profiling [66]–[68] or programmer's hints[6], [79], [80]. However, none of them addresses data locality in iterative parallel computations.

We note that data locality in iterative parallel computations cannot be exploited if the data is too large because the data accessed at the previous iteration is usually already evicted from the cache at the next iteration. Such data locality in iterative parallel programs cannot be exploited by any task scheduler, but there are cases where programs have consecutive parallel loops at each recursion level. For example, construction of a decision tree involves consecutive parallel loops at each node of the tree for calculating information gain, as explained in Section 5.2.3 in detail. Such data locality in consecutive parallel computations can be exploited by carefully designing a task scheduler.

To make ADWS perform well with large data, this thesis introduces the concept of "multi-level scheduling." Multi-level scheduling is a generalization of two-level scheduling [66]–[68], [78], [84] for arbitrary memory hierarchy. In multi-level scheduling, we abstract memory hierarchy as a tree of memory groups rooted by the main memory as in [27]. Memory groups at level l are treated as workers at level l and one of the cores under the memory group can be a leader, performing any kind of scheduling such as work stealing at this level. When a task becomes smaller than its cache size, the task is passed to the next level $l + 1$ and its descendants are only scheduled within the memory group. As this procedure is applied recursively, we can ensure that tasks scheduled under each memory group does not exceed the corresponding shared cache size. Scheduling algorithms at each level can be chosen arbitrarily; in this thesis we consider multi-level work stealing (WS) and multi-level ADWS. In the following, we call the original ADWS "single-level ADWS" for clarity.

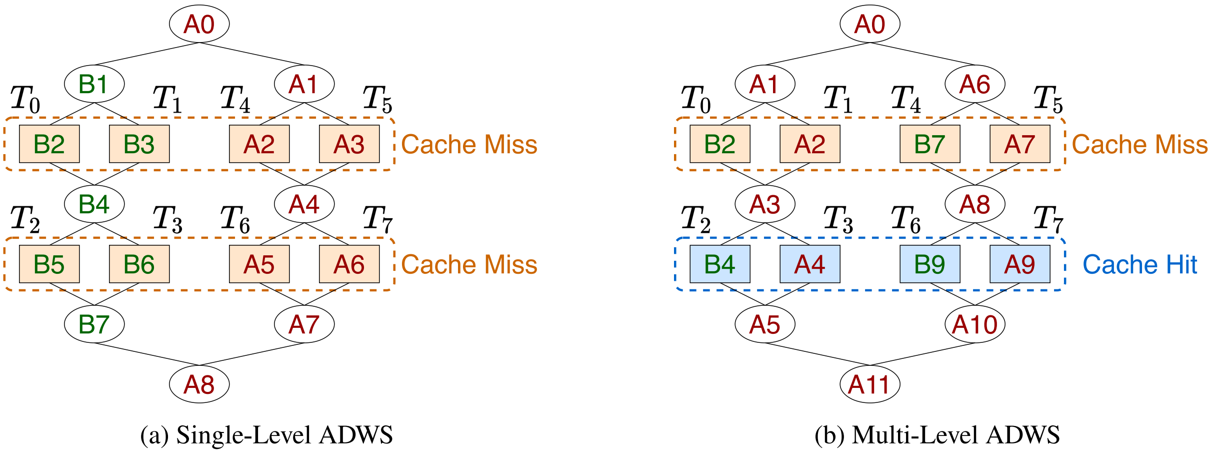
Multi-level ADWS has some benefits to multi-level WS. First of all, it can exploit data locality in consecutive parallel computations as mentioned above. Furthermore, in multi-level scheduling, the computation size at each level becomes smaller, in which case ADWS is expected to perform better because of fast distribution of tasks avoiding lock contention (see Chapter 4). Finally, deterministic scheduling for the level of NUMA nodes can exploit data locality in NUMA architectures for regular computations.

To experimentally show the benefits of multi-level ADWS, we conducted experiments with various benchmarks using a two-socket Cascadelake machine. We compared the performance of five schedulers: single-level WS (traditional random work stealing), single-level ADWS, multi-level WS, multi-level ADWS, and SB scheduler based on the implementation provided in [81], [82]. The results show that multi-level ADWS outperforms other schedulers in highly memory-bound benchmarks with various data sizes. For example, in construction of a decision tree with with large real-world dataset of about 2 GB, multi-level ADWS achieves 37% performance improvement compared with single-level ADWS. Even compared with multi-level WS, multi-level ADWS achieved 16% performance improvement, which supports the benefit of deterministic scheduling at each recursion level. On the other hand, it was found that multi-level scheduling can incur large idle time of workers, especially in irregular computations, showing a general trade-off between idleness and data locality.

5.2. Motivation

5.2.1. Data Locality Issue in Single-Level ADWS

Single-level ADWS schedules nested parallel tasks deterministically so that close workers execute close tasks. Single-level ADWS works well if all shared caches in the system have an infinite size or the data size is smaller than the sum of all shared caches, but if the data do not fit into the shared caches, it does not perform good scheduling in terms of data locality. This is because single-level ADWS tries to schedule the entire DAG in one go, and workers execute tasks as far from each other as possible, resulting in many cache misses on shared caches when the entire data do not fit into the shared caches.



(a) Single-Level ADWS

(b) Multi-Level ADWS

Fig. 5.1.: Example of DAG scheduling under a finite-size cache shared by two workers A and B. This DAG represents a divide-and-conquer computation over a global array, which does not fit into the shared cache but the half of which fits into the cache. The rectangle nodes are leaf tasks that access to the array, while the ellipse nodes do not access to the array. Each node is attributed by the worker identifier (A or B) who executes the node, followed by a global time step. The subarray accessed by T_0 and T_1 is also accessed by T_2 and T_3 again; the same applies to T_4, T_5, T_6, and T_7.

Fig. 5.1 demonstrates that single-level ADWS incurs many cache misses under a shared cache of limited size. This DAG of a typical divide-and-conquer algorithm, in which only leaf tasks touch the data in the global array, is scheduled by worker A and B sharing the same shared cache. The rectangle nodes are leaf tasks which access a part of the global array, while the ellipse nodes are intermediate tasks which logically divide the array into two equally-sized arrays recursively. Each node is attributed by the worker identifier (A or B) who executes the node, followed by a global time step. For example, the nodes with “A1” and “B1” are executed by worker A and B at the same time. The total size of the global array is larger than that of the shared cache, but the half of the global array fits into the shared cache. Thus, the subarray accessed by T_0 and T_1 fits into the shared cache. The subarray accessed by T_0 and T_1 is also accessed by T_2 and T_3 again; the same applies to T_4, T_5, T_6, and T_7. The first access to the global array (T_0, T_1, T_4, and T_5) always incurs cache misses, but the second access (T_2, T_3, T_6, and T_7) does not if the scheduler respects the data locality. However, single-level ADWS incurs cache misses upon the second access. Roughly speaking, in single-level ADWS, worker A executes the right part and worker B executes the left part of the DAG. Because the upper part of the DAG (T_0, T_1, T_4, and T_5) is scheduled almost at the same time, the entire data is loaded into the shared cache first. Then, when workers execute the lower part of the DAG (T_2, T_3, T_6, and T_7), data in the shared cache is likely to have been evicted, causing cache misses again.

On the other hand, scheduling in Fig. 5.1b does not cause cache misses upon the second access because only a part of the array is computed at once. For example, the subarray accessed by T_0 and T_1 is loaded to the shared cache, and then T_2 and T_3 are immediately executed by the same workers to reuse the data in the shared cache. The key difference is that only a part of the DAG that fits into the shared cache is scheduled at one time. It is a demonstration of *multi-level ADWS*, which we will describe in detail later.

5.2.2. Scheduling Principles for Arbitrary Memory Hierarchy

Before getting deeper into the details of multi-level scheduling, let us consider what is the optimal scheduling under arbitrary memory hierarchy. If there are only private caches, workers should try to schedule tasks as far

Algorithm 5.1: Algorithm of decision tree construction.

```
1  Function CreateDecitionTree($rows$)
2      foreach $attr \in attributes$ do
3          $split \leftarrow$ ComputeBestSplit($rows, attr$)
4          if $split$ is better than $bestSplit$ then $bestSplit \leftarrow split$
5      if Splitting at $bestSplit$ is beneficial then
6          ($leftRows, rightRows$) $\leftarrow$ Partition($rows, bestSplit$)
7          $leftNode \leftarrow$ CreateDecisionTree($leftRows$)
8          $rightNode \leftarrow$ CreateDecisionTree($rightRows$)
9          return Node($leftNode, rightNode$)
10     else
11         return LeafNode($rows$)
```

from each other as possible to avoid data sharing across private caches. Random work stealing is known to be asymptotically optimal for private caches [28], and single-level ADWS works well on private caches as well. If there is a cache shared by all workers, they should schedule a part of the DAG that fits into the shared cache at one time, as we discussed above. Generalizing these rules, the principles for arbitrary memory hierarchy are summarized as follows:

1. For distributed caches, tasks executed under each cache should be as far from each other as possible.
2. For a shared cache, tasks executed under the cache should not access larger data than the cache size.

SB schedulers only address the second principle, as they omit data reuse among close tasks in their model [6], [79], [80]. On designing a practically efficient scheduler, it is important to respect both of the above two principles.

5.2.3. Motivating Example: Decision Tree Construction

Some recursive algorithms have consecutive parallel loops at each recursion level, in which cases deterministic scheduling is beneficial to promote cache reuse across consecutive parallel computations. Construction of a decision tree is a good motivating example for it. Construction algorithm of a decision tree is by nature nested parallel; it can be easily and straightforwardly parallelized by using nested fork-join structures, as previously studied in [85]. For simplicity, this thesis uses CART [86] algorithm for constructing a decision tree and targets for only binary-classification tasks using continuous attributes, but the idea is also applicable to other decision tree algorithms.

Algorithm 5.1 shows a simplified algorithm of decision tree construction. It receives as input training data ($rows$), each row of which consists of a class (0 or 1) and multiple attributes of continuous values. The task is to predict the class from multiple attributes. The algorithm is based on a two-way divide-and-conquer algorithm similar to quicksort. At line 9, the data is partitioned into two parts based on $bestSplit$, which consists of $attr_{best}$ and x_{best} where rows are split based on whether a row has an attribute $attr_{best}$ larger than x_{best} or not. After that, two subtrees are recursively constructed using the partitioned two data ($leftRows$ and $rightRows$), which can be computed in parallel using fork-join constructs (line 7–8).

$bestSplit$ is chosen to the split which has the lowest value of Gini impurity, which involves consecutive parallel computations (line 2–4). For each attribute, the best split with the smallest Gini impurity is calculated

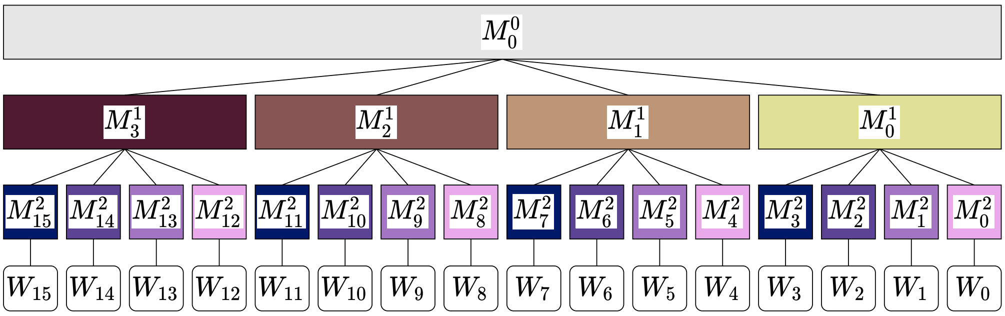


Fig. 5.2.: Example of a tree of memory groups.

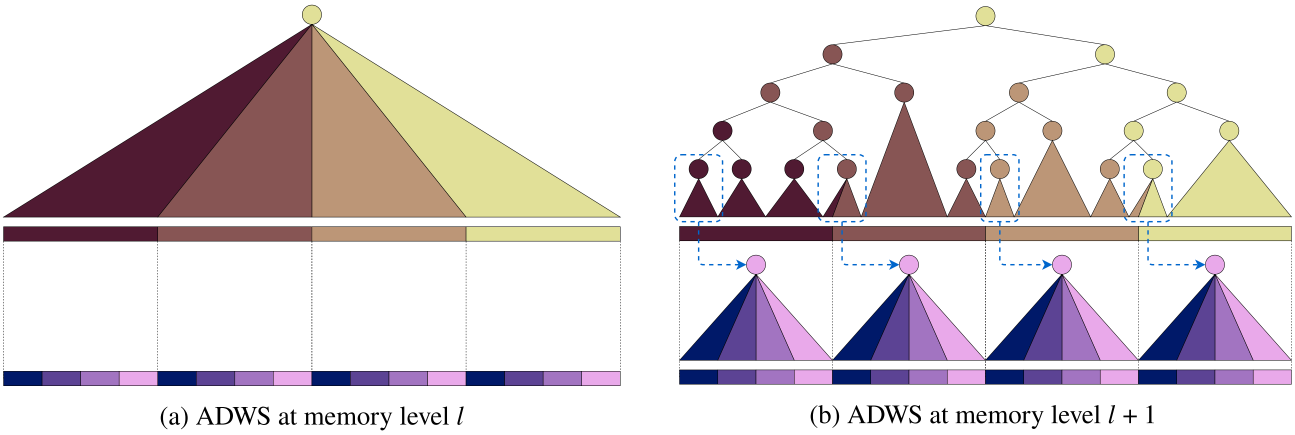
(line 3), and among them *bestSplit* is chosen. The best split for each attribute is usually chosen by sorting rows by the values of the attribute and calculating Gini impurity at every split point. However, sorting for every attribute is costly and thus some improvements for it exist. For example, LightGBM [87], a widely-used gradient boosting framework based on decision trees, adopts histogram-based approaches originally proposed in [88], [89]. In the histogram-based approach, rows with the attribute value within the same interval are counted into the same bin, constructing a histogram. The best split is chosen from the set of bins in the histogram. Thus, ComputeBestSplit function at line 3 can be written as a simple parallel loop (or reduction) to construct a histogram. It forms consecutive parallel loops at each recursion level, whose data locality should be exploited by deterministic scheduling for better performance.

5.3. Algorithm and Design

5.3.1. Multi-Level Scheduling

Multi-level ADWS is designed to follow the two scheduling principles presented in Section 5.2.2. To explain multi-level ADWS, first let us consider a generalized representation of the memory hierarchy [27]. The memory hierarchy can be represented by a tree, in which each node represents a cache or memory. In this thesis we assume that leaf nodes are private caches to cores (e.g., L1 cache) and the root node is a DRAM memory, which has an infinite size. Fig. 5.2 shows an example of the memory hierarchy as a tree. Supposing that a worker corresponds to a core, each worker belongs to multiple memory nodes, which we call *memory groups*. A memory group M_i^l is identified by the memory level l and the index i among memory groups at the same level. The memory level of the root node (DRAM) is 0, thus denoted by M_0^0.

Then we explain the basics of multi-level scheduling. Multi-level scheduling is a generalization of two-level scheduling [66]–[68], [78], [84] for arbitrary memory hierarchy. These existing approaches were proposed for two levels of memory hierarchy (shared and distributed memory [84], machines with private caches and a single shared cache [78], and multi-socket architectures [66]–[68]), but their idea can be extended to arbitrary memory hierarchy. Suppose that a task is being executed in a memory group M_i^l, and the size of the data that will be accessed by the task and its descendants is guaranteed to be smaller than the capacity of M_i^l. M_i^l has multiple child memory groups, denoted by M_j^{l+1}, where j is any index of children. Since the child memory groups can be treated as distributed caches, it is better to apply scheduling algorithms suited for private caches, such as work stealing and ADWS, to child memory groups M_j^{l+1}. Then, when M_j^{l+1} encounters a task that will fit into its own cache while scheduling at level l, it recursively applies this procedure to the next level ($l + 1$). The task passed to the level l is called "level-l root task," which fits into the level-l cache but its parent does not. From


(a) ADWS at memory level l

(b) ADWS at memory level $l + 1$

Fig. 5.3.: Illustration of the idea of multi-level ADWS.

the viewpoint from level-l workers, level-l root tasks are just leaf tasks, which are executed "sequentially" by level-l workers (in fact they are scheduled by child level-$(l + 1)$ workers in parallel).

On scheduling tasks under M_i^l, its children M_j^{l+1} are treated as "workers" at this level of scheduling. In implementation, one of the workers (W_k) under a memory group M_j^{l+1} becomes a leader of M_j^{l+1} upon completion of a level-$(l + 1)$ root task scheduled to M_j^{l+1} and behaves as if it is the worker of M_j^{l+1}. Only one level-l root task can be scheduled to M_i^l at the same time; therefore we can assume that only one root task exists in level-l scheduling, which is usually the case for single-level scheduling starting from the only one root task. Thus, we can apply any scheduling algorithm optimized for nested parallel computations at each level, such as ADWS. In principle, scheduling policies for each memory group are composable; different scheduling policies can be adopted at each memory level. For simplicity, in our evaluation we only consider multi-level schedulers whose scheduling policies are the same among all memory levels. Multi-level work stealing (WS) is a multi-level scheduler whose scheduling policy at each level is traditional work-first random work stealing [16]. Multi-level ADWS is the one that applies single-level ADWS to each memory level, which is illustrated in Fig. 5.3. The memory hierarchy of this figure corresponds to Fig. 5.2. In multi-level ADWS, tasks are scheduled "almost" deterministically at each memory level. We note that in multi-level ADWS, the worker with the smallest rank becomes the leader of memory groups because of the restriction of the algorithm of ADWS, while in multi-level WS any worker can be the leader of memory groups it belongs to (typically the worker who completes the last join of the level-l root task).

5.3.2. Programming Model

In order to perform multi-level scheduling introduced above, the data size of each task should be known in advance. Chen et al. [66]–[68] proposed an online profiling method for estimating data sizes for iterative computations which repeat the same computation for many times, and SB schedulers [6], [79], [80] use programmer's hints on data size for each task. In this thesis, we adopt the latter approach for simplicity.

Combined with the hints on the amount of work for each task in single-level ADWS, the programming style in multi-level ADWS uses the task group notation shown in Fig. 5.4. Compared to Fig. 4.3 in Section 4.2.2, `data_size` parameter is added to the task group initialization. It represents the overall data size that will be accessed during computations of this task group. Although previous approaches mentioned above assign data sizes to tasks, this thesis assigns them to task groups rather than each task, which gives more strict and fine-grained data size limits on nested parallel computations. For example, in the matmul example (Fig. 4.5 in

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

Fig. 5.4.: Extension of the TBB-like task group construct for multi-level ADWS.

Section 4.2.2), for each task group, only the half of matrix A and B is used for computation. However, if we assign the data size to the entire task, the data size should be the sum of sizes of matrix A, B, and C, which is larger than the actual space required.

5.3.3. Memory Group Aggregation

One downside of multi-level scheduling is that even for computations small enough to fit into the total cache, it tries to schedule them in a multi-level way, causing underutilization of cores. For example, if the total data size fits into the cache of memory group M_i^l, only the workers under M_i^l work on it, leaving other workers idle. As previously studied, when the overall data fits into the cache, single-level ADWS should work well compared to other scheduling methods. Thus, we introduce an optimization for multi-level ADWS called "memory group aggregation," in which when the data size is small enough to fit into the cache, multi-level memory groups are aggregated into single-level memory groups and single-level ADWS is applied to them.

For example, in Fig. 5.2, if the overall data size is smaller than the sum of cache capacities of M_0^1, M_1^1, M_2^1, and M_3^1, the memory groups at this level is omitted and the child memory groups of them are aggregated into a single level. Thus, single-level ADWS is applied to memory groups $M_0^2, \ldots, M_{15}^2$. More formally, memory level aggregation in multi-level ADWS works as follows. When a memory group M_i^l encounters a task group, it first checks if the data size of the task group is smaller than the sum of cache capacities under the current distribution range ($[i, j]$): $M_i^l, \ldots, M_j^l$. If true, M_i^l further checks the sum of capacities of children of $M_i^l, \ldots, M_j^l$. As long as the data size fits into the sum of child caches, it increments the memory level and checks the sum of capacities of descendants at the level. When the data size becomes larger than the sum of capacities of memory groups or it reaches the leaf memory groups, the memory groups at the level are aggregated into single-level memory groups, applied single-level ADWS until completion of this task group.

5.4. Implementation over User-Level Threads

This section explains in detail the implementation of multi-level ADWS over user-level threading libraries such as MassiveThreads [36]. Although we only describe the implementation of multi-level ADWS, multi-level WS or other multi-level scheduling methods can be implemented similarly as well. The explanation of this section basically follows the notation explained in Section 4.3.1. Algorithm 5.2 shows the type definitions, worker-local variables, and global variables used in multi-level ADWS. It shows only the difference from Algorithm 4.1; structs not shown in Algorithm 5.2 are the same as Algorithm 4.1. Algorithm 5.3 shows the main part of the implementation of multi-level ADWS.

MemoryGroup struct (line 1–10 in Algorithm 5.2) represents the data of each memory group. Most of the members of MemoryGroup are the same as those of Worker struct in Algorithm 4.1. In multi-level ADWS,

Algorithm 5.2: Definitions in multi-level ADWS.

```
1  Struct MemoryGroup
2      parent :: Pointer to MemoryGroup
3      children :: List of pointers to MemoryGroup
4      capacity :: integer
5      mlevel :: integer
6      dtree :: Pointer to DistributionTree
7      searchRootTask :: Pointer to Task
8      localQueue :: WSQueue
9      migrationQueue :: WSQueue
10     useMigrationQueue ← false :: boolean
11 Struct DistributionRange
12     rootMG :: Pointer to MemoryGroup
13     mlevel :: integer
14     from :: double
15     to :: double
16 Worker-Local Variables
17     myMGList :: List of pointers to MemoryGroup      ▹ List of mem groups this worker belongs to
18     myMemLevel :: integer                             ▹ Current memory level of this worker
19     myMG :: Pointer to MemoryGroup                    ▹ Shorthand for myMGList[myMemLevel]
20     myTask :: Pointer to Task                         ▹ The currently running task on this worker
21 Global Variables
22     maxMemLevel :: integer                 ▹ The depth of the memory hierarchy of the machine
```

each memory group acts as a "worker" in single-level ADWS; hence members in Worker struct in Algorithm 4.1 are moved to MemoryGroup struct in Algorithm 5.2. Each worker (a real worker corresponding to a core) has a list of memory groups that it belongs to ($myMGList$ at line 17 in Algorithm 5.2). For example, worker W_4 in Fig. 5.2 has a list $[M_0^0, M_1^1, M_4^2]$.

Since memory groups act as workers at each memory level, we need to designate one of the workers to represent a memory group. In this implementation, the worker of the smallest rank under a memory group is designated to the head of the memory group. Initially, each worker's current memory level ($myMemLevel$ at line 18) is set to the level of the topmost memory group among the memory groups whose head is the worker. For example, in Fig. 5.2 the initial memory level of W_0 is 0 (because W_0 is the head of M_0^0, M_0^1, and M_0^2), that of worker W_4 is 1, and that of worker W_5 is 2. The root task is initially executed by W_0, and subsequently tasks are distributed to other workers as memory levels of workers are incremented.

In DistributionRange struct (line 11–15), members $rootMG$ and $mlevel$ are newly introduced. In multi-level ADWS, a distribution range is defined as a range of memory groups at a specific memory level. Memory groups in the domain are all memory groups under the root memory group $rootMG$ at the memory level $mlevel$. For example, in Fig. 5.2, if $rootMG$ is M_0^0 and $mlevel$ is 1, the domain of the distribution range covers M_0^1, M_1^1, M_2^1, and M_3^1, and if $rootMG$ is M_0^0 and $mlevel$ is 2, it covers $M_0^2 \dots M_{15}^2$. The latter case is used for *memory group aggregation* described in Section 5.3.3.

Algorithm 5.2 is not so different from Algorithm 4.1; one of the major differences is that $myWorker$ is replaced with $myMG$, which is the memory group of the current memory level the worker belongs to (a shorthand for $myMGList[myMemLevel]$). Another major difference comes from the management of memory

Algorithm 5.3: Algorithm of multi-level ADWS.

```
Function TaskGroup::Init($(work, size)$)
    $myTask.work \leftarrow work$
    $this.drange \leftarrow myTask.drange$; $this.dtree \Rightarrow myTask.dtree$
    $mlevel \leftarrow myMemLevel$; $cap \leftarrow myMG.capacity$
    while $mlevel < maxMemLevel - 1$ and $size \leq cap$ do
        $mlevel \leftarrow mlevel + 1$
        $cap \leftarrow$ Sum of capacities of descendants of $myMG$ at level $mlevel$
    if $mlevel \neq myMemLevel$ then
        $N_{desc} \leftarrow$ # of descendants of $myMG$ at level $mlevel$
        $myTask.drange \leftarrow$
          DistributionRange($(rootMG \Rightarrow myMG, mlevel \leftarrow mlevel, from \leftarrow 0, to \leftarrow N_{desc})$)
        $myMemLevel \leftarrow mlevel$; $myMG \Rightarrow myMGList[myMemLevel]$
    $this.isSearchRoot \leftarrow \lfloor myTask.drange.from \rfloor \neq \lfloor myTask.drange.to \rfloor$
    if $this.isSearchRoot$ then
        $dtree \Rightarrow$ new DistributionTree($(drange \leftarrow myTask.drange, parent \Rightarrow myTask.dtree)$)
        $myTask.dtree \Rightarrow dtree$; $myMG.dtree \Rightarrow dtree$
Function TaskGroup::Create($(lambda, work)$)
    $newTask \Rightarrow$ new Task($(lambda \leftarrow lambda, dtree \Rightarrow myTask.dtree)$)
    $myTask.work \leftarrow myTask.work - work$
    $(myTask.drange, newTask.drange) \leftarrow$ divide $myTask.drange$ in the ratio $myTask.work : work$
    $targetMG \Rightarrow$ the memory group of index $\lfloor newTask.drange.from \rfloor$ in $newTask.drange$
    if $targetMG = myMG$ then
        Push $myTask$ into $myMG$'s local queue and run $newTask$
    else
        if $\lfloor newTask.drange.from \rfloor \neq \lfloor newTask.drange.to \rfloor$ then
            Pass $newTask$ to $targetMG$ as a search-root task
        else
            Pass $newTask$ to $targetMG$ as a non-search-root task
    Add $newTask$ to $this.tasks$
Function TaskGroup::Wait()
    if $\lfloor myTask.drange.from \rfloor \neq \lfloor myTask.drange.to \rfloor$ then
        $myTask.dtree.active \leftarrow$ true
    foreach $task \in this.tasks$ do
        Wait for completion of $task$
    if $this.isSearchRoot$ then
        $ownerMG \Rightarrow$ the memory group of index $\lfloor myTask.drange.from \rfloor$ in $myTask.drange$
        Pass $myTask$ to $ownerMG$ as a search-root task and go to the scheduler
        $myMG.dtree \Rightarrow myTask.dtree.parent$
        delete $myTask.dtree$
    $myTask.drange \leftarrow this.drange$; $myTask.dtree \Rightarrow this.dtree$
    $myMemLevel \leftarrow myTask.drange.mlevel$; $myMG \Rightarrow myMGList[myMemLevel]$
Function OnTaskCompletion($(task)$)
    if $\lfloor task.drange.from \rfloor \neq \lfloor task.drange.to \rfloor$ then
        $task.dtree.active \leftarrow$ true
Function Scheduler()
    while true do
        $task \Rightarrow$ Try to pop a $myMG$'s local task
        if $task = NULL$ then $task \Rightarrow$ Try to steal a task from others
        if $task \neq NULL$ then Run $task$
```

CPU model	# of sockets	# of cores	frequency	L1 data cache	L2 cache	L3 cache
Xeon Platinum 8280 (Cascadelake)	2	56 (28 × 2)	2.7 GHz	32 KB/core	1 MB/core	38.5 MB/socket

Tab. 5.1.: CPU information used in the experiments.

levels. In multi-level ADWS the memory level is changed (more specifically, incremented) at the beginning of a task group and restored at the end of the task group.

Line 4–7 determines the next memory level ($mlevel$) by comparing the size of the task group (given by the programmer) with the capacity under the current memory group. If the size of the task group is small enough to fit into all child memory groups under the memory group, the memory level is further incremented (memory group aggregation). For example, supposing that the current memory level is 0, the memory level is first incremented to 1 because the root memory group has an infinite size. Furthermore, if the size of the task group is smaller than the sum of capacities of M_0^1, M_1^1, M_2^1, and M_3^1 in Fig. 5.2, the next memory level becomes 2, skipping the memory level of 1. In that case the distribution range of the current task is updated to the one with the root memory group M_0^0 and the memory level of 2 (line 9–10), which is equivalent to that of single-level ADWS. In other words, if the total data size of the computation is small enough to fit into the shared caches, multi-level ADWS falls back to single-level ADWS to achieve better performance. This implementation is a simplified version of memory group aggregation explained in Section 5.3.3; for evaluation we implemented a more complicated version that accounts for distribution ranges.

If the memory level is incremented at the beginning of a task group, the memory level is restored at the end of the task group. Since the distribution range is saved at the beginning of the task group before incrementing the memory level, the restored distribution range at the end of the task group has the memory level to be restored (line 40).

5.5. Performance Evaluation

We conducted experiments with various benchmarks using a two-socket Cascadelake machine in the Oakbridge-CX supercomputer in the University of Tokyo, which configuration is summarized in Tab. 5.1. The OS installed on the machine was CentOS Linux 7 (Core), whose kernel version was `3.10.0-957.el7.x86_64`. Hyper-threading was disabled, and turbo-boost and transparent huge pages (THP) were enabled. All programs were compiled by GCC v7.5.0 with `-O3` option, and for benchmark programs we also set `-march=native` option for enabling optimizations for architectures. We did not set `-march=native` option for compiling thread schedulers because it used SIMD instructions for scheduler code, which slowed down the CPU frequency.

For evaluation, we implemented five schedulers: single-level WS (traditional random work stealing that was the default scheduler in MassiveThreads), single-level ADWS, multi-level WS, multi-level ADWS, and SB scheduler. They are denoted as *WS (sl)*, *ADWS (sl)*, *WS (ml)*, *ADWS (ml)*, and *SB* in figures, respectively. For fair comparison, all of these schedulers were implemented in MassiveThreads [36], a lightweight user-level threading library. The implementation of SB scheduler was ported to MassiveThreads from the one published by Simhadri et al. [81], [82] that was used in the evaluation in their paper. We used the “SB-D” variant of SB schedulers [81], [82] with a little modification to avoid lock contention, because in most cases it performed better than the “SB” variant, which uses centralized task queues. The parameters for the SB scheduler used in our evaluation were $\sigma = 0.5$ and $\mu = 0.2$, which are the same configuration in their evaluation. Because it was not trivial to give hints on data sizes for task groups rather than data sizes for tasks in SB scheduler (see

```
void RRM(double* array, size_t n) {
  if (n < $CUTOFF$) {
    for (int k = 0; k < $N_b$; k++) {
      for (size_t i = 0; i < n; i++) {
        array[i] += $c$ * array[i];
      }
    }
  } else {
    for (int k = 0; k < $N_b$; k++) {
      parallel_for (size_t i = 0; i < n; i++) {
        array[i] += $c$ * array[i];
      }
    }
    task_group tg(2, n * sizeof(double));
    tg.run([&] { RRM(array      , n/2    ); }, 1);
    tg.run([&] { RRM(array + n/2, n - n/2); }, 1);
    tg.wait();
  }
}
```

Fig. 5.5.: Pseudocode of the Recursive Repeated Map (RRM) benchmark.

Section 5.3.2), we give hints on data sizes for tasks only for SB scheduler. We note that we spawned all tasks in each task group (including the last task in the task group) in all schedulers for fair comparison with SB scheduler, which requires the last task in task groups to be spawned as a separate task.

In all evaluations, as many workers as cores were created and pinned to cores by setting affinity for the underlying kernel-level threads of workers. Although we set the information of the memory hierarchy (e.g., cache capacity, parent-child relationship) by hand for evaluation, such information should be able to be automatically collected. Unless explicitly noted, we set `--interleave=all` option to `numactl` command when running experiments, meaning that physical pages are almost evenly distributed to all NUMA nodes. When measuring serial execution times, `--localalloc` option was passed to `numactl` command and the process was pinned to a core to avoid unnecessary remote memory accesses.

5.5.1. Benchmarks

We used six benchmarks for evaluating the performance of schedulers. We plotted speedup of total execution times compared to the serial execution times (except for MatMul benchmark), changing the data sizes. We repeated the computation for 11 times within each run, omitting the performance of the first run (warm-up). We also repeated the execution from the outside of the program for 3 times; the number of samples of execution times were 30 in total. For each point in figures, geometric mean with an error bar of geometric standard deviation was plotted. The explanation of each benchmark follows.

Recursive Repeated Map (RRM) Benchmark

We used a simplified version of Recursive Repeated Map (RRM) benchmark used in the evaluation of SB scheduler [81], [82]. Fig. 5.5 shows pseudocode of RRM benchmark. This benchmark takes as input an array of double-precision floating-point elements and logically divides the array into two equally-sized subarrays recursively. At each recursion level, a subarray is repeatedly applied a *map* function which just multiplies an element by a constant and adds it to the element itself (fused multiply-add; FMA). The number of repeated maps at each recursion level (N_b) is set to 3. Recursion stops when the data size becomes smaller than 32 KB. The map function (represented by `parallel_for`) is parallelized by recursively dividing the array into two equally-sized subarrays as well until the data size becomes smaller than 128 KB. Since there is little amount of computation for each memory access, it is highly memory-bound. Data locality in the consecutive map functions is expected to be exploited by deterministic scheduling. Fig. 5.6 shows speedup of RRM benchmark with different data sizes.

Quicksort Benchmark

Quicksort benchmark is based on a well-known divide-and-conquer quicksort algorithm with parallelization of recursive function calls. The partition operation is also parallelized to increase parallelism. It uses double buffering for partition; thus the total data size required is double of the input array size. A pivot is chosen as the median of the first three elements. The shape of the DAG is similar to RRM benchmark but it can be irregular, as the number of child elements depends on the pivot value. Both of the cutoff sizes for recursion and partition are set to 64 KB. Each element is a double-precision floating-point number, which is randomly generated. Fig. 5.7 shows speedup of Quicksort benchmark with different input data sizes.

KDTree Benchmark

KDTree benchmark constructs a kd-tree for three-dimensional points (double-precision), which are randomly generated. The algorithm is similar to that of quicksort. An axis to partition points (x, y, or z) is chosen in a round-robin way as recursion proceeds. The pivot value is chosen as the median of the first three values as well. It keeps creating tree nodes until the data size becomes smaller than 4 KB. Both of the cutoff sizes for task creation for recursion and partition are set to 64 KB. Compared to Quicksort benchmark, KDTree benchmark is more memory-bound because recursion stops earlier than quicksort, reducing the amount of computation per memory access. Fig. 5.8 shows speedup of KDTree benchmark with different input data sizes.

Decision Tree Benchmark

The algorithm of decision tree construction was explained in Section 5.2.3 in detail. It is also similar to Quicksort and KDTree benchmark, but it has consecutive parallel loops at each recursion level. We used HIGGS dataset downloaded from the UCI machine learning repository [90], which was published with the paper [91]. 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, and other rows were used for training, whose size is about 2 GB. The performance evaluation accounts only for the training time. As validation of our implementation of decision tree algorithm, we checked that our implementation achieved 72% accuracy for testing data, while accuracy with random prediction was 52%, which indicates that our implementation learned something from the training data. The maximum depth for a decision tree was set to 17, which achieved the best accuracy using all training data. The cutoff size for recursion was 64 KB and that for parallel loops and partition was 256 KB. Fig. 5.9 shows speedup of Decision Tree benchmark with different numbers of rows used for training.

Matrix Multiplication (MatMul) Benchmark

Matrix Multiplication (MatMul) benchmark is the same one as in Section 4.4.2. It calculates multiplication of single-precision floating-point matrices (SGEMM). The algorithm is based on cache-oblivious matrix multiplication for square-sized matrices, which divides the matrices into four submatrices recursively. The cutoff size for recursion was 64×64. The computation kernel was optimized using SIMD instructions by hand. We added padding of 128 KB to avoid cache conflict with matrices whose size is a power of two, because it showed the best performance. Because MatMul benchmark is expected to achieve nearly the peak performance of the machine, we plot FLOPS instead of speedup in the following graphs. Fig. 5.10 shows FLOPS of MatMul benchmarks with different matrix sizes. We note that the machine's peak performance was 8601.6 GFLOPS (assuming that the CPU frequency while executing AVX-512 instructions is 2.4 GHz).

Heat2D Benchmark

Heat2D benchmark is the same one as in Section 4.4.1. It is a simple 5-point stencil computation, which is parallelized by dividing a square grid into four subgrids recursively. The cutoff size for recursion was 64×64. The execution time for 50 iterations was used for performance evaluation. Since the DAG of Heat2D benchmark is a "simple" DAG where each task has at most one task group, cache reuse rate within a task is not so large as other benchmarks. Thus, for data sizes larger than the total cache size, the bandwidth of the main memory is expected to become the bottleneck. We added padding of 256 KB to avoid cache conflict with matrices whose size is a power of two, because it showed the best performance. Fig. 5.11 shows speedup of Heat2D benchmark with different matrix sizes.

5.5.2. Settings for Performance Analysis

We also conducted profiling for breaking down the performance differences among schedulers. Benchmarks with data larger than the total cache size were picked for profiling: RRM with 1 GB data; Quicksort with 1 GB data; KDTree with 1 GB data; Decision Tree with full training data; MatMul with $N = 16384$; and Heat2D with $N = 16384$. All of them correspond to the rightmost point in their graph plots.

Fig. 5.12 shows breakdown of execution times profiled by workers. Because profiling incurs overheads especially when tasks are fine-grained, the profiling was conducted separately from the performance evaluation. Thus, the overhead of schedulers might be a little larger than the normal execution without profiling. In Fig. 5.12, "Busy time" is the time consumed by benchmark programs themselves; i.e., time while workers are doing meaningful work. "Idle time" is the time while idle workers are searching for ready tasks; this metric is related to greediness of scheduling policies. "Overhead" is the time other than "Busy time" and "Idle time," which is the overhead of schedulers themselves, including context switching and task creation. Although these times were measured by each worker, they were divided by the number of workers and thus the plot shows real execution times.

Fig. 5.13 shows cache miss counts profiled by `perf` system calls in Linux. Instead of `perf` commands, we directly called `perf_event_open()` system call from the benchmark programs to avoid counting unnecessary events (e.g., operations at initialization) and to profile each iteration separately. We referenced Intel software developer's manual [92] for determining which performance counter to use. In Fig. 5.13, L1 misses and L2 misses are counted by `L1D.REPLACEMENT` and `L2_RQSTS.MISS` event, respectively. L3 misses are counted by uncore events rather than by core events, so that precise events for shared L3 caches can be obtained. The

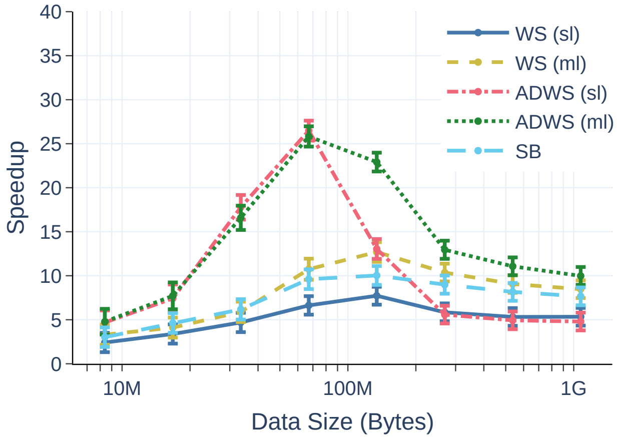


Fig. 5.6.: Speedup of Recursive Repeated Map (RRM) benchmark with different data sizes.

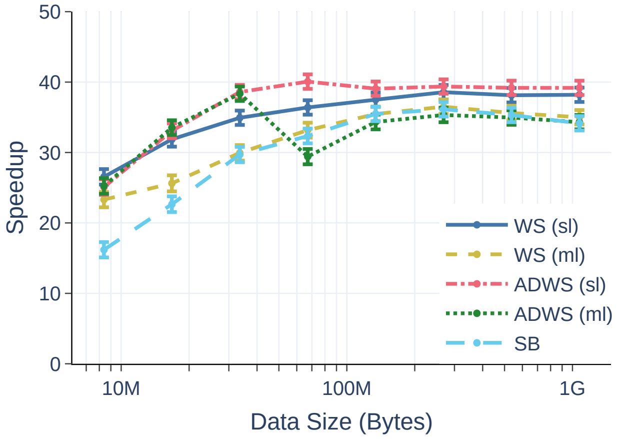


Fig. 5.7.: Speedup of Quicksort benchmark with different data sizes.

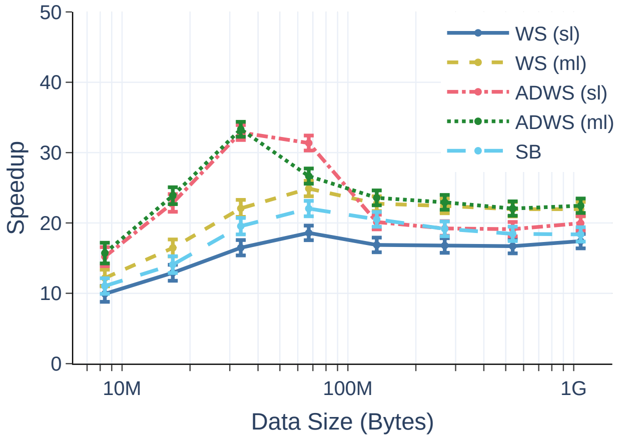


Fig. 5.8.: Speedup of KDTree benchmark with different data sizes.

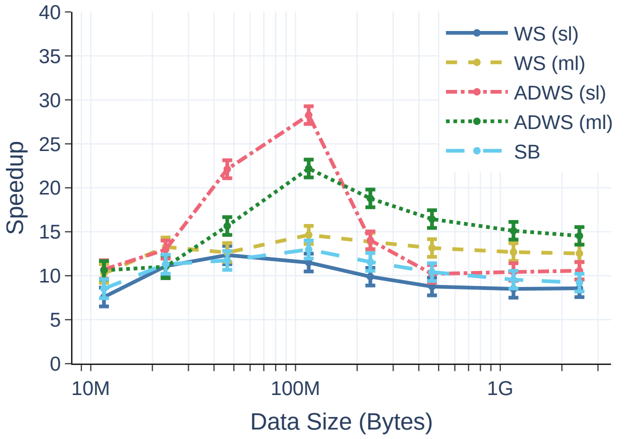


Fig. 5.9.: Speedup of Decision Tree benchmark with different data sizes.

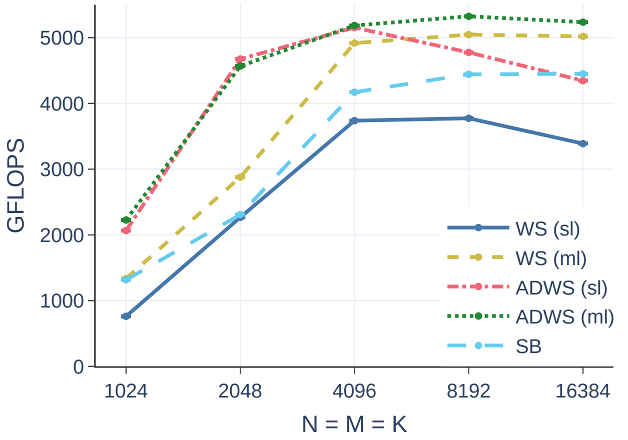


Fig. 5.10.: FLOPS of Matrix Multiplication (MatMul) benchmark with different data sizes.

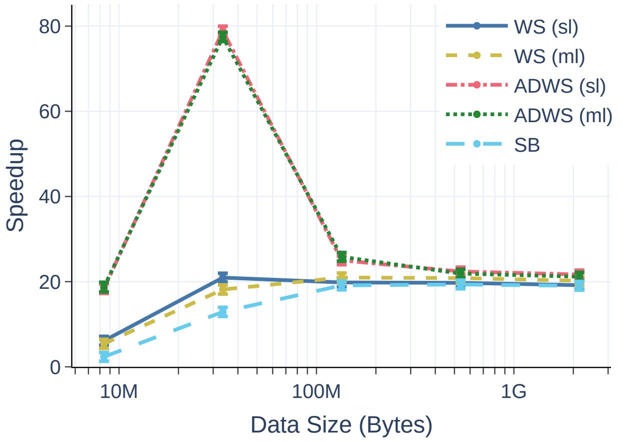


Fig. 5.11.: Speedup of Heat2D benchmark with different data sizes.

event used for counting L3 misses is `LLC_LOOKUP.ANY` (`Cn_MSR_PMON_BOX_FILTER0.state=0x1`) in uncore Caching/Home Agent (CHA) units [93]. All of these events for cache misses count the total number of cache replacements, including data reads, writes, and prefetches.

Fig. 5.14 shows breakdown of core cycles profiled using `perf` system calls together with cache misses. Since we count core cycles rather than reference cycles, the cycles may not correspond to the real time because of dynamic CPU frequency scaling, especially for serial execution (because turbo boost is enabled). “Memory stall cycles” are counted by `CYCLE_ACTIVITY.STALLS_MEM_ANY` event, which counts stall cycles due to any memory subsystems, including caches and TLBs. “Other stall cycles” are stall cycles which are not caused by memory subsystems, such as stalls due to branch misprediction, calculated by (`CYCLE_ACTIVITY.STALLS_TOTAL - CYCLE_ACTIVITY.STALLS_MEM_ANY`). “Productive cycles” are core cycles without any stalls, calculated by (`CPU_CLK_UNHALTED.THREAD - CYCLE_ACTIVITY.STALLS_TOTAL`). Note that productive cycles include cycles consumed by idle workers searching for ready tasks, which may not be “productive.”

5.5.3. Results and Discussion

This section discusses the results of performance evaluation and detailed profiling. In RRM (Fig. 5.6), KDTree (Fig. 5.8), Decision Tree (Fig. 5.9), and MatMul (Fig. 5.10) benchmarks, we can see performance improvements from single-level ADWS to multi-level ADWS with large data sizes. Notably, multi-level ADWS marks 20% performance improvement to single-level ADWS in MatMul with $N = 16384$ and 37% in Decision Tree with full training data. Looking into Fig. 5.12, we can see that multi-level ADWS largely reduces the busy time in these benchmarks, indicating that data locality is improved. The L3 cache miss count in Fig. 5.13 supports this result; L3 misses in multi-level schedulers are much smaller than that in single-level schedulers. Moreover, L3 misses in multi-level schedulers are almost the same as that in serial execution, which shows that multi-level schedulers are optimal in terms of L3 cache misses. Sometimes L3 misses in multi-level schedulers are slightly smaller than that in serial execution because parallel execution can utilize multiple L3 caches in the system.

However, in Quicksort (Fig. 5.7), the performance of multi-level schedulers is worse than that of single-level schedulers. The reason is clearly shown in Fig. 5.12; the idle time of multi-level schedulers is much larger than that of single-level schedulers. In Quicksort, multi-level ADWS slightly reduces the busy time but increase of the idle time is larger, resulting in worse overall performance. The breakdown of core cycles (Fig. 5.14) shows that Quicksort is more compute-bound than other benchmarks in which multi-level schedulers perform better because the ratio of memory stall cycles in Quicksort is smaller than them. One exception is MatMul benchmark, which does not incur large idle time because the computation pattern is rather regular. The long idle time of multi-level schedulers is due to their scheduling policy: only one level-l root task can be simultaneously scheduled in a memory group M_i^l. Especially in irregular computation, level-l root tasks can be so small that little parallelism is exposed by them, resulting in underutilization of cores under M_i^l.

SB scheduler was proposed to address this issue of underutilization of cores in irregular computations [6]; however, there is a general trade-off between idleness and data locality. The trade-off is shown in the evaluation results; execution time breakdown (Fig. 5.12) shows that idle time of SB scheduler is smaller than that of multi-level schedulers but the busy time of SB scheduler is much larger. Looking into the cache miss counts (Fig. 5.13), SB scheduler largely reduces L3 misses compared to single-level schedulers, which are oblivious to shared cache sizes; however, the L3 miss counts of SB schedulers are larger than that of multi-level schedulers. This is because SB schedulers try to schedule many tasks as long as the total data size of them is smaller than the cache size. However, at the same time, tasks are decomposed into smaller tasks for scheduling many tasks

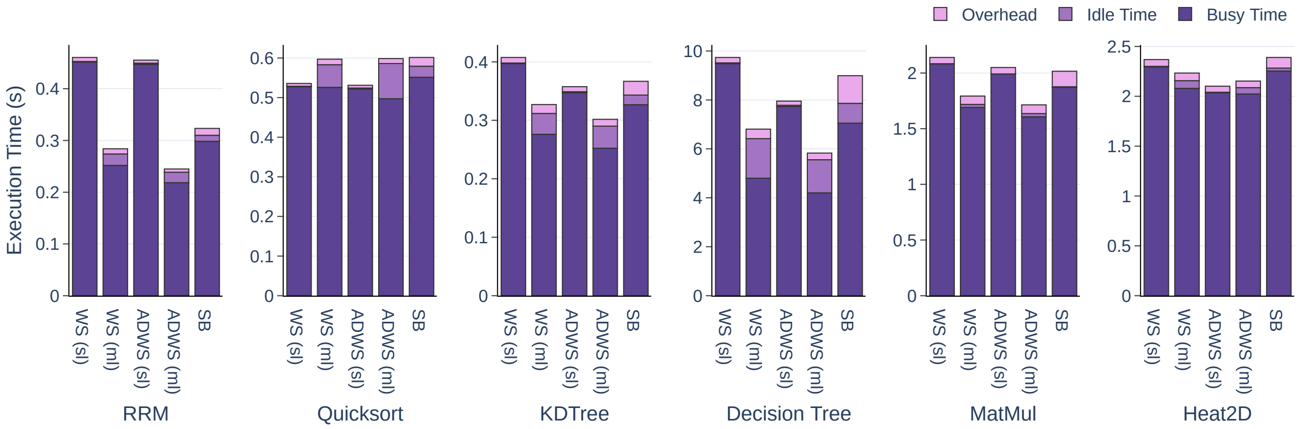


Fig. 5.12.: Breakdown of execution times profiled by the schedulers.

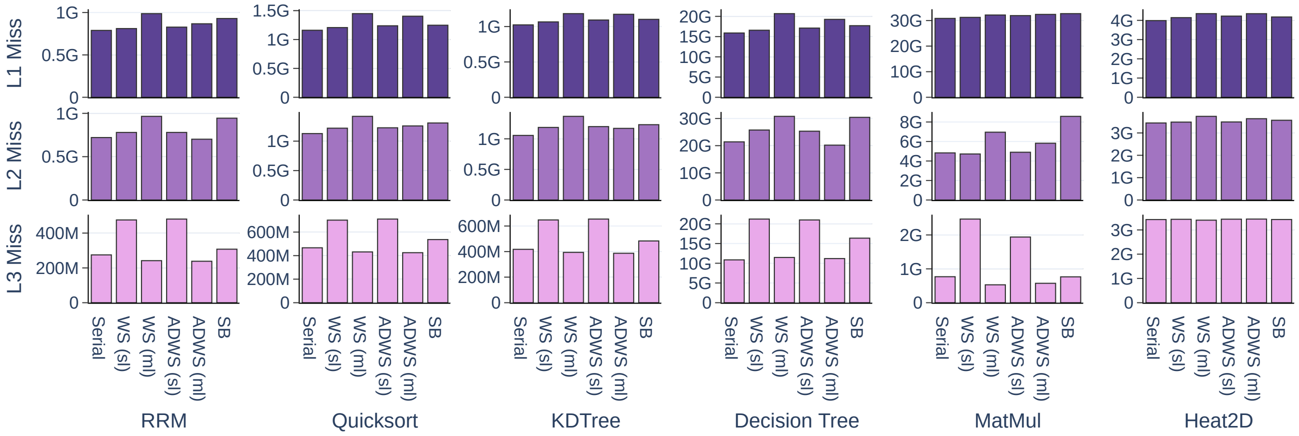


Fig. 5.13.: Cache miss counts profiled by `perf` system calls.

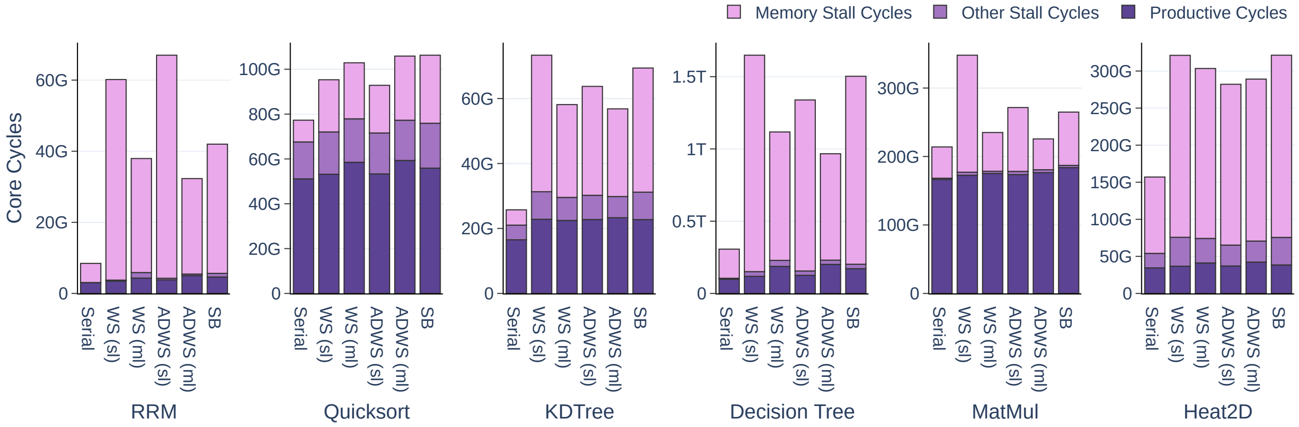


Fig. 5.14.: Breakdown of core cycles profiled by `perf` system calls.

in a shared cache at the same time, which reduces the chance of cache reuse within the task. It can be controlled by σ parameter how much tasks are decomposed [6], [81], [82], but there is still a trade-off between idleness and data locality. Another issue in the current design of SB scheduler in that it ignores data reuse across sibling tasks, which results in high L2 miss counts in practice (Fig. 5.13). Improvement for SB scheduler to respect data reuse across sibling tasks would be interesting future work, but it is out of scope of this thesis. We note that the results about SB scheduler in this thesis are a little different from the results reported in [81], [82], mainly because of the difference of the CPU architecture. In our evaluation, each socket has 28 cores, whereas in their evaluation each socket has only 8 cores. Thus, the effect of L2 cache misses should be relatively large in our settings. Besides, the overhead of SB scheduler is larger because of remaining lock contention, which may be minimized by more engineering efforts.

In RRM (Fig. 5.6), Decision Tree (Fig. 5.9), and MatMul (Fig. 5.10) benchmarks, multi-level ADWS consistently shows better performance than multi-level WS. 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 Decision Tree benchmark (see Section 5.2.3), in which multi-level ADWS marks 16% performance improvement to multi-level WS with full training data. In these benchmarks we can see that L2 cache misses in multi-level WS are larger than other schedulers (except for SB scheduler) in Fig. 5.13. In general, multi-level scheduling can increase the number of steals and thus increase the number of private cache misses. However, multi-level ADWS can reduce the number of steals by utilizing the programmer's hints on the amount of work for each task, which minimizes the increase of L2 misses. Moreover, multi-level ADWS can exploit data locality in consecutive parallel computations at each recursion level, which further reduces the L2 misses.

Heat2D benchmark (Fig. 5.11) shows different performance tendency from other benchmarks. When the data size fits into the cache, single-level and multi-level ADWS achieves nearly 80x speedup. Such super-linear speedup is possible because parallel execution can utilize multiple shared caches larger than serial execution. With data larger than the total cache size, the performance significantly drops. In heat2D, there is little data reuse within subtasks; therefore cache miss counts are almost the same among different schedulers with data larger than the total cache size (Fig. 5.13). However, the busy time in Fig. 5.12 is slightly different among schedulers. This is because simultaneous execution of close tasks in the DAG by different workers can cause large CPU stalls, because they can write to the same cache line. Such effects can hardly be seen in cache miss counts in Fig. 5.13, but memory stall cycles in Fig. 5.14 indicate that. This is because it does not cause many cache replacements. Single-level and multi-level ADWS performs slightly better than others because ADWS can schedule as far tasks from each other as possible using programmer's hints on the amount of work.

In addition, in most cases, multi-level ADWS can achieve almost the same performance as single-level ADWS with small data sizes because of memory group aggregation. Thus, our multi-level ADWS implementation can achieve higher performance than other schedulers with many benchmarks with various data sizes. One remaining issue in multi-level ADWS is that it degrades its performance for irregular and compute-bound programs because of increased idle time. As it is not a trivial issue considering the general trade-off between idleness and data locality, we leave it our future work.

5.6. Summary

This chapter pointed out the data locality issue in ADWS for shared caches and proposed multi-level scheduling, which is a generalization of existing work on shared-cache-aware schedulers. Then we introduced a shared-cache-aware variant of ADWS, called multi-level ADWS, in which ADWS is applied to each memory level. Multi-level ADWS can exploit data locality in consecutive parallel computations at each recursion level, which makes multi-level ADWS outperform other schedulers including multi-level WS. Detailed performance analysis on various benchmarks supports the benefits of multi-level ADWS, which improve data locality and execution times in many cases. However, it is also found that multi-level scheduling can impose long idle time for workers, which is especially problematic for irregular and compute-bound programs. This issue is challenging as there is a general trade-off between idleness and data locality, which has not been resolved yet. Tackling this issue is an interesting and challenging research direction in the future.

6. Related Work

6.1. Preemption for User-Level Threads

We introduced and evaluated preemption techniques for user-level threads (M:N threads) in Chapter 3, but research on preemption support for M:N threads is not new. In the following, we introduce many researches on preemptive M:N threads and some techniques similar to our approaches to support preemption for M:N threads.

6.1.1. OS Support for Preemptive User-Level Threads

In the early stages of threading support in operating systems, many researchers investigated the interface design between the kernel and user space to implement M:N threads. During this time, researchers investigated using mechanisms to notify kernel events such as blocking and preemption to M:N threads via a special KLT—they called this model *scheduler activation* [42]. Another approach was proposed by Marsh et al. as *first-class user-level threads* [94], in which the kernel notifies events to the user space by using signals. *Nanothreads* passes preemption events from the kernel to the user space via shared memory between them [95]. Solaris OS supported M:N threads; preemption was enabled by using signals [96]. All of these studies required the kernel to be aware of M:N threads. Unfortunately, modern mainstream operating systems have abandoned such capabilities. For example, Solaris OS moved to 1:1 threads from version 9 because of the simple implementation of 1:1 threads [97]. As a result, most modern M:N thread-based parallel systems lack preemption [31]–[37]. Preemption techniques for M:N threads presented in this thesis work on today's mainstream operating systems that are unaware of M:N threads.

6.1.2. Interruption-Based Preemption without OS Modification

The *signal-yield* technique that was presented in Section 3.2.1 has been previously investigated [7]–[9] and integrated into the Go language [40], [41]. In the context of real-time systems, Anantaraman et al. [7] studied signal-yield on single-core systems, whereas Mollison and Anderson [8] extended it to multicore processors. Mollison and Anderson also noted in their paper that unsafe functions should be migrated to "proxy threads" for safety. This recommendation is impractical since it requires modifying all KLT-dependent functions. Boucher et al. [9] also pointed out this problem and proposed a workaround for safely preempting the execution of external libraries. Their method isolates external shared libraries by loading multiple versions of shared libraries into different linker namespaces and dynamically switching them by modifying the global offset table (GOT). Their workaround, however, imposes additional overheads to every call to external functions. Moreover, their method cannot deal with the KLT-dependence issue when parallelism exists in external libraries (e.g., Intel MKL). Our KLT-switching is the first practical technique that tackles the KLT-dependence issue. We also note that none of the past studies mentioned timer optimizations, which is one of the contributions in this thesis.

6.1.3. Threading Techniques Similar to KLT-Switching

As explained in Section 3.2.1, KLT-switching virtualizes workers and each worker no longer corresponds to a KLT in KLT-switching. Techniques for virtualizing workers have also been studied, for example, in Concurrent Cilk [98]. These techniques, however, are not designed for implicit preemption but for explicit yield operations on run-to-completion threads or “tasks.” This approach simplifies the design somewhat, because yielding no longer needs to be in the context of a signal handler; but it also makes such previous work not applicable to implicit preemption. *User-level processes* proposed by Hori et al. [99] have a similar concept to KLT-switching to deal with the *system-call consistency* issue (i.e., a user-level process has to call system calls from the same KLT throughout its execution), which is especially important for processes. Their approach includes dynamically switching KLTs when system calls are trapped so that system calls are always called from the same KLT. Although this approach is similar to our KLT-switching technique, preemption was not concerned in their work.

6.1.4. Compiler-Based Preemption

Some researchers have proposed compiler-based techniques for preemptive M:N threads. The general idea of these approaches is that the compiler would insert explicit thread scheduling points. This approach has been extensively studied in some Java virtual machine implementations [100], [101]. Unlike timer-based preemption, however, compiler-inserted preemption points cannot be placed at arbitrary points in the code and would depend on the application code structure. Some studies [43], [44] have investigated the optimal frequency of preemption points; frequent preemption points can degrade performance but incur less latency. To achieve more precise timing of preemption at compilation time, Ghosh et al. [102] proposed static estimation for execution time of code blocks by calculating clock cycle latencies of instructions using LLVM [103]. Some work aiming at functional languages, such as Scheme, treats function entrances as scheduling points [104], [105]. It works effectively because function languages usually rely on recursion to perform iterative operations and do not have tight loops in a single function. The Glasgow Haskell Compiler [106] uses an OS timer for preemption, but rather than directly interrupting execution by signals as our approach does, it performs preemption at the explicit scheduling point right after timer expiration for safety. All of these techniques rely on compiler support, and most of them are specific to languages. Our techniques, on the other hand, can be implemented in a threading library and can therefore be easily deployed on most systems.

6.2. Locality-Aware Schedulers for Task Parallel Programs

Until now, many locality-aware schedulers for task parallel programs have been proposed. This section introduces them and compares their design with ADWS presented in Chapter 4 and Chapter 5 in this thesis. We note that schedulers listed here are not limited to ones for nested parallel computations; schedulers for more general task parallelism are also included.

6.2.1. Affinity-Based Approaches

Affinity-based approaches, in which tasks are attributed with locations where they should be executed, are often used to mitigate data locality problems in iterative parallel programs. Affinities are just scheduling hints and thus can be ignored. First, Acar et al. [28] proposed *locality-guided work stealing*, which uses a concept of “mailbox” for delivering tasks with affinities. When a task with an affinity for a worker is created, the task is pushed to both the local queue and the mailbox of the targeting worker. A worker checks its mailbox before attempting a steal. If a task with affinity is popped from the local queue, the task is removed from the mailbox, ignoring the affinity. Robison et al. [107] pointed out the problem of initial task distribution in locality-guided work stealing: workers steal inappropriate tasks before tasks with affinity for them arrive, which results in ignoring affinities of many tasks. They resolved this issue by putting a delay in the scheduler for a task tree to get sufficiently unfolded before performing work stealing.

Work pushing, which is a similar technique to the affinity-based one, was investigated in [63], [73], [108]. Drebes et al. [73], [108] proposed work pushing, in which tasks are pushed into the task queues of workers which have good data locality properties for the tasks. To decide which worker to push tasks to, they utilize data dependency information in OpenStream [109], a data-flow extension of OpenMP. Thus, their technique is not applicable to nested fork-join programs which do not have any information about data dependencies. Deters et al. [63] proposed *lazy work pushing*, which respects the work-first principle [24] in nested fork-join programs and gives a good bound on execution time and cache misses in NUMA architectures. The data locality properties of tasks are given by the programmer as hints, which is described in Section 6.2.4 in more detail.

Maglalang et al. [110] proposed *color*-based scheduling on NABBIT library [111] that targets for more general DAGs than nested fork-join computations. Colors are similar to affinities to specify which worker should preferably execute the task. They introduced an efficient runtime design for respecting colors of tasks provided by the programmer.

6.2.2. Adaptation for Iterative Parallel Programs

Some approaches take advantage of the structure of iterative applications to improve data locality [65]–[70]. *Constrained work stealing* [65] schedules tasks deterministically for iterative applications by tracing and replaying. Its previous work [74] 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. Because most of the tasks are executed in the same NUMA nodes as in the previous iteration, the data locality for NUMA nodes is improved. However, because random work stealing is used to schedule the first iteration, there is no guarantee that workers in the same memory group will execute close tasks in the computation graph.

CATS [66] and LAWS [67], [68] proposed online profiling methods to estimate the data size of each task in nested parallel computations. The estimated data size is used for improving data locality in shared caches, as explained in Section 6.2.6.

6.2.3. Combination of Initial Partitioning and Dynamic Load Balancing

Like ADWS, there are also approaches that combine the initial partitioning of tasks with dynamic load balancing. Yoo et al. [112] 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 [113] was proposed to combine the benefits of work sharing and work stealing for distributed memory machines. It first distributes to nodes tasks that are large enough to be efficiently executed within a node, and then work stealing is performed within a node. LAWS [67], [68] is similar to AHS in that they initially distribute tasks to the level of NUMA nodes deterministically to increase local accesses to NUMA nodes. However, these approaches do not account for initial partitioning to the level of cores, which limits the capability of promoting data reuse in private caches.

6.2.4. Scheduling Hints on Data Locality

As it is generally hard to improve data locality of applications without any knowledge or assumption about applications, scheduling hints on data locality provided by the programmer are often exploited for improving data locality. Affinities and colors explained in Section 6.2.1 are considered scheduling hints provided by the programmer. SLAW [62] and NUMAWS [63] require annotations for *places* where the task should be executed (e.g., NUMA nodes), which prevents the program to be oblivious to hardwares. DistWS [114] requires application-specific hints rather than hardware-specific ones, but the hints are not intuitive for the programmer because they are not transparent to how DistWS works. Compared with them, the hints in ADWS are application-specific and transparent to how ADWS works; therefore the programmer does not need to have much knowledge about hardwares and details of ADWS. It is considered to be good properties of hints in order not to harm portability and productivity.

Space-bounded schedulers [6], [79], [80] require hints on data sizes of each task for shared-cache-aware scheduling (explained in Section 6.2.6). Multi-level ADWS introduced in Chapter 5 also require this hints be provided by the programmer. This hint also respects the above mentioned good properties of locality hints.

6.2.5. Improvements of Steal Strategies

There are many studies that improve steal strategies of traditional random work stealing without relying on scheduling hints by the programmer. HotSLAW [64] extends the principle of SLAW [62], but it does not require any locality hint from the programmer. Min et al. [64] 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. Basically, they are heuristics to reduce the number of steals across different nodes in distributed memory environments.

Quintin and Wagner [84] proposed two heuristics to improve data locality in work stealing: *Probability Work Stealing (PWS)* and *Hierarchical Work Stealing (HWS)* for distributed work stealing. In PWS, a thief chooses a victim randomly but not uniformly; the probability to choose a victim is proportional to the inverse of the distance between the thief and the victim, so that close workers are preferably chosen as victims for stealing. HWS is similar to “multi-level” scheduling presented in this thesis; it splits tasks to *global* tasks, which can be stolen across nodes, and *local* tasks, which can only be stolen within a node. Only one of the workers in a node (called a leader) can steal global tasks from other nodes, and other workers in the node can steal only local tasks. HWS differs from multi-level scheduling presented in this thesis in that HWS determines if a task is global or not by considering the depth of the task rather than the data size. It works well for distributed memory assuming that the overall data fits into the main memory, but it is not aware of shared cache sizes; it can cause many shared cache misses only with the information of the task depth.

Localized work stealing was proposed in [115], 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 introduced in Section 4.2.4 in that it dynamically makes some groups preferable to steal tasks from.

All of these approaches, however, have limitation in improvement of data locality because of lack of information about applications. Especially in scheduling on shared caches, it is hard to avoid excessive shared cache misses in these approaches.

6.2.6. Schedulers Aware of Shared Cache Sizes

Section 2.3.4 explained the algorithm of Parallel depth-first (PDF) scheduler [29], [30], which is one of the earliest schedulers for shared caches. However, because of its centralized design of task management, it was known that it degrades the performance when tasks are fine-grained; therefore variants of PDF schedulers have also been proposed, such as *AsyncDF* [116]. The experimental study conducted by Chen et al. [117] supports that PDF scheduler performs better than work stealing under processors with a shared cache. However, PDF scheduler does not have good data locality property on private caches, which leaded to researches on schedulers that work optimally both for private caches and a shared cache. Narlikar [77] proposed a scheduler combining work stealing and PDF scheduler, and Blelloch et al. [78] proposed *Controlled-PDF* scheduler and gave good bound on the number of cache misses on machines with private caches and a single shared cache. In Controlled-PDF scheduler, at the shared-cache level, the DAG is executed in depth-first order until a task fits into the shared cache, and then the descendants of the task are scheduled by PDF scheduler for private caches until a subtask fits into the private cache. This is a specific case of "multi-level scheduling" presented in Section 5.3.1 for machines with private caches and a single shared cache.

For multi-socket multi-core architectures, Chen et al. [66] proposed CATS, in which triple-level work stealing was introduced. Triple-level work stealing is also a specific case of multi-level scheduling for multi-socket multi-core architectures. In CATS, data sizes for each task are estimated by online profiling as addressed in Section 6.2.2 and used for determining root tasks for each socket. They also presented LAWS [67], [68], which performs NUMA-aware initial partitioning of tasks in addition to cache-aware scheduling.

All of the above approaches assume specific configuration of hardwares. For arbitrary cache hierarchy, space-bounded (SB) scheduler [6], [79], [80] was proposed and the bound for cache complexity and execution time was analyzed. The idea of SB scheduler is to "anchor" a task to a cache, which means that the descendants of the task are executed only under the cache. SB scheduler differs from "multi-level scheduling" in that it can anchor multiple tasks as long as the total data size of tasks anchored to a cache does not exceed the cache capacity. Thus, as studied in [6], SB scheduler is expected to be more robust to irregular computations than multi-level scheduling, which is supported by our experimental results in Section 5.5. However, our evaluation also reveals that the number of cache misses is larger than that of multi-level scheduling. Simhadri et al. [81], [82] conducted experimental analysis on SB scheduler and showed that SB scheduler could reduce shared cache misses and execution times in practice. Blelloch et al. [83] proposed *space-bounded recursive-PDF scheduler*, which combines Controlled-PDF and SB scheduler to limit the space usage for programs which dynamically allocate memory during parallel computation, whereas this thesis only deal with programs that statically allocate memory before parallel computation.

7. Conclusion and Future Work

This thesis introduced efficient threading strategies for nested parallel programs. Efficiently handling nested parallelism in runtime systems is important for productivity and composability of parallel programs primarily because of the following two reasons. First, many algorithms are nested parallel by their nature. Second, when a parallel program calls parallel subroutines inside of a parallel region, nested parallelism appears. User-level threading is expected to efficiently handle nested parallelism because of its extremely lightweight threading operations and flexible scheduling policy. However, today's world was not ready for fully exploiting such nested parallelism that can appear in many contexts because of several challenges in user-level threads.

The first challenge addressed in this thesis is the deadlock issue in nested composition of parallel programs over user-level threads. Because many existing parallel programs assume that they can exclusively use all the machine resources, they sometimes use synchronization mechanisms based on busy loops for the sake of performance. However, in such settings busy-loops in each thread can cause a deadlock since today's most user-level threads are nonpreemptive. Therefore, this thesis investigated preemption techniques for user-level threads as a workaround for the deadlock issue. The evaluation shows that our preemption techniques can successfully resolve the deadlock issue while preserving lightweight threading operations. In the evaluation of Cholesky decomposition, preemptive user-level threads marked 27% performance improvement over kernel-level threads, while avoiding a deadlock which otherwise occurs over nonpreemptive user-level threads. In addition, we found that preemptive user-level threads are also beneficial for applications other than nested composition of parallel programs, such as thread packing and in situ analysis.

The second challenge is to improve data locality in scheduling nested parallel computations on machines with deep memory hierarchy. It is practically important to respect data locality in nested parallel computations for arbitrary memory hierarchy in modern computer architectures, as memory hierarchy is getting deeper and more complicated in these days. Thus, this thesis proposed a novel task scheduler for nested parallel programs, called Almost Deterministic Work Stealing (ADWS). The idea of ADWS is to deterministically schedule tasks so that the task hierarchy matches machine's memory hierarchy, by using hints on the relative amount of work for each task provided by the programmer. It also has the capability of dynamic load balancing to fix load imbalance — this is why it is "almost" deterministic. The evaluation revealed that ADWS was up to nearly six times faster than the traditional work stealing strategy in 5-point stencil computation. However, it was also found that the performance of ADWS was not good when the overall data of computation does not fit into the last level cache.

This thesis also proposed "multi-level scheduling" for improving data locality in shared caches. The concept of multi-level scheduling is orthogonal to ADWS; multi-level scheduling aims at reducing shared cache misses by explicitly taking into account the data size of each task. The design of multi-level scheduling is hierarchical and composable; scheduling policy at each cache level can be arbitrarily chosen. In the evaluation, we implemented multi-level ADWS, a variant of multi-level scheduling with ADWS applied at each level, and multi-level WS, a variant with work stealing. The detailed evaluation results showed that multi-level ADWS achieved 37% performance improvement to the original single-level ADWS in decision tree construction with 2 GB data, reducing many shared cache misses. Multi-level ADWS also achieved 16% performance improvement to multi-level WS, showing that ADWS is beneficial for data locality in consecutive parallel loops at each recursion level. These results show that multi-level ADWS performs well on many benchmarks with a wide range of data sizes.

However, the evaluation also revealed that multi-level scheduling incurred large idleness of workers, which degraded the overall performance of some benchmarks, even if data locality was improved. We can see the general trade-off between data locality and idleness of workers in the evaluation; reduction of idle times can supersede improvement of data locality depending on the characteristics of applications. Therefore, we need a unified way to tackle this trade-off in the future. One approach we can consider is to automatically choose the best scheduling method (e.g., single-level or multi-level) by analyzing the characteristics of applications (e.g., whether it is memory-bound or not). Evaluating ADWS on distributed environments would be an interesting direction of future work.

List of Publications

International Conferences

[1] S. Shiina and K. Taura, “Almost deterministic work stealing,” in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC ’19, Denver, Colorado, USA, 2019.

[2] S. Shiina, S. Iwasaki, K. Taura, and P. Balaji, “Lightweight preemptive user-level threads,” in *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP ’21, to appear, Virtual Event, Republic of Korea, 2021.

Domestic Conferences

- S. Shiina and K. Taura, “Almost deterministic work stealing,” in *The 3rd Cross-Disciplinary Workshop on Computing Systems, Infrastructures, and Programming*, ser. xSIG 2019, Yokohama, Kanagawa, May. 2019. (**Best Research Award**)

References

[3] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran, “Cache-oblivious algorithms,” in *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, FOCS ’99, New York, New York, USA, 1999, pp. 285–297.

[4] ——, “Cache-oblivious algorithms,” *ACM Transactions on Algorithms*, vol. 8, no. 1, 4:1–4:22, 2012.

[5] G. E. Blelloch, P. B. Gibbons, and H. V. Simhadri, “Low depth cache-oblivious algorithms,” in *Proceedings of the Twenty-Second Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA ’10, Thira, Santorini, Greece, 2010, pp. 189–199.

[6] G. E. Blelloch, J. T. Fineman, P. B. Gibbons, and H. V. Simhadri, “Scheduling irregular parallel computations on hierarchical caches,” in *Proceedings of the Twenty-Third Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA ’11, San Jose, California, USA, 2011, pp. 355–366.

[7] A. V. Anantaraman, A. E. haj Mahmoud, R. K. Venkatesan, Y. Zhu, and F. Mueller, “EDF-DVS scheduling on the IBM embedded PowerPC 405LP,” in *Proceedings of the IBM T.J. Watson P=ac^2 Conference*, Yorktown Heights, New York, USA, 2004, pp. 63–72.

[8] M. S. Mollison and J. H. Anderson, “Bringing theory into practice: A userspace library for multicore real-time scheduling,” in *Proceedings of the 19th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, RTAS ’13, Philadelphia, Pennsylvania, USA, 2013, pp. 283–292.

[9] S. Boucher, A. Kalia, D. G. Andersen, and M. Kaminsky, “Lightweight preemptible functions,” in *Proceedings of the 2020 USENIX Annual Technical Conference*, USENIX ATC ’20, 2020, pp. 465–477.

[10] M. Gates, J. Kurzak, A. Charara, A. YarKhan, and J. Dongarra, “SLATE: Design of a modern distributed and accelerated linear algebra library,” in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC ’19, Denver, Colorado, USA, 2019.

[11] Intel Corporation. (). “Intel® Math Kernel Library (Intel® MKL) | Intel® Software,” [Online]. Available: `https://software.intel.com/en-us/mkl`.

[12] M. Adams, J. Brown, J. Shalf, B. V. Straalen, E. Strohmaier, and S. Williams, “HPGMG 1.0: A benchmark for ranking high performance computing systems,” Tech. Rep. LBNL 6630E, 2014.

[13] R. Cochran, C. Hankendi, A. K. Coskun, and S. Reda, “Pack & cap: Adaptive DVFS and thread packing under power caps,” in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO ’11, Porto Alegre, Brazil, 2011, pp. 175–185.

[14] Sandia National Laboratories. (). “LAMMPS molecular dynamics simulator,” [Online]. Available: `http://lammps.sandia.gov`.

[15] S. Plimpton, “Fast parallel algorithms for short-range molecular dynamics,” *Journal of Computational Physics*, vol. 117, no. 1, pp. 1–19, 1995.

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

[17] OpenMP Architecture Review Board, *OpenMP Application Program Interface Version 5.0*, 2018. [Online]. Available: `https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf`.

[18] "IEEE Standard for Information Technology–Portable Operating System Interface (POSIX(R)) Base Specifications, Issue 7," *IEEE Std 1003.1-2017 (Revision of IEEE Std 1003.1-2008)*, 2018.

[19] H. Pan, B. Hindman, and K. Asanović, "Composing parallel software efficiently with Lithe," in *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, Toronto, Ontario, Canada, 2010, pp. 376–387.

[20] Z. Majo and T. R. Gross, "A library for portable and composable data locality optimizations for NUMA systems," in *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '15, San Francisco, California, USA, 2015, pp. 227–238.

[21] ——, "A library for portable and composable data locality optimizations for NUMA systems," *ACM Transactions on Parallel Computing*, vol. 3, no. 4, 20:1–20:32, 2017.

[22] 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 *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '95, Santa Barbara, California, USA, 1995, pp. 207–216.

[23] ——, "Cilk: An efficient multithreaded runtime system," *Journal of Parallel and Distributed Computing*, vol. 37, no. 1, pp. 55–69, 1996.

[24] M. Frigo, C. E. Leiserson, and K. H. Randall, "The implementation of the Cilk-5 multithreaded language," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '98, Montreal, Quebec, Canada, 1998, pp. 212–223.

[25] J. Reinders, *Intel Threading Building Blocks: Outfitting C++ for Multi-Core Processor Parallelism.* O'Reilly Media, 2007.

[26] J. Schuchart, K. Tsugane, J. Gracia, and M. Sato, "The impact of taskyield on the design of tasks communicating through MPI," in *Proceedings of the 14th International Workshop on OpenMP*, IWOMP '18, Barcelona, Spain, 2018, pp. 3–17.

[27] B. Alpern, L. Carter, and J. Ferrante, "Modeling parallel computers as memory hierarchies," in *Proceedings of Workshop on Programming Models for Massively Parallel Computers*, PMMP '93, Berlin, Germany, 1993, pp. 116–123.

[28] U. A. Acar, G. E. Blelloch, and R. D. Blumofe, "The data locality of work stealing," in *Proceedings of the Twelfth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '00, Bar Harbor, Maine, USA, 2000.

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

[30] G. E. Blelloch and P. B. Gibbons, “Effectively sharing a cache among threads,” in *Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA ’04, Barcelona, Spain, 2004, pp. 235–244.

[31] L. V. Kale and S. Krishnan, “CHARM++: A portable concurrent object oriented system based on C++,” in *Proceedings of the Eighth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, OOPSLA ’93, Washington, D.C., USA, 1993, pp. 91–108.

[32] A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas, “OmpSs: A proposal for programming heterogeneous multi-core architectures,” *Parallel Processing Letters*, vol. 21, no. 02, pp. 173–193, 2011.

[33] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, “X10: An object-oriented approach to non-uniform cluster computing,” in *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA ’05, San Diego, California, USA, 2005, pp. 519–538.

[34] B. L. Chamberlain, D. Callahan, and H. P. Zima, “Parallel programmability and the Chapel language,” *International Journal of High Performance Computing Applications*, vol. 21, no. 3, pp. 291–312, 2007.

[35] K. B. Wheeler, R. C. Murphyand, and D. Thain, “Qthreads: An API for programming with millions of lightweight threads,” in *Proceedings of the 22nd IEEE International Parallel and Distributed Processing Symposium*, IPDPS ’08, Miami, Florida, USA, 2008.

[36] J. Nakashima and K. Taura, “MassiveThreads: A thread library for high productivity languages,” *Concurrent Objects and Beyond*, vol. 8665, pp. 222–238, 2014.

[37] 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, “Argobots: A lightweight low-level threading and tasking framework,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 3, pp. 512–526, 2017.

[38] S. Iwasaki, A. Amer, K. Taura, and P. Balaji, “Lessons learned from analyzing dynamic promotion for user-level threading,” in *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, SC ’18, Dallas, Texas, USA, 2018, pp. 293–304.

[39] ——, “Analyzing the performance trade-off in implementing user-level threads,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 8, pp. 1859–1877, 2020.

[40] A. A. A. Donovan and B. W. Kernighan, *The Go Programming Language*. Addison-Wesley Professional, 2015.

[41] (2020). “Go 1.14 release notes,” [Online]. Available: `https://golang.org/doc/go1.14`.

[42] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy, “Scheduler activations: Effective kernel support for the user-level management of parallelism,” *ACM Transactions on Computer Systems*, vol. 10, no. 1, pp. 53–79, 1992.

[43] M. Bertogna, O. Xhani, M. Marinoni, F. Esposito, and G. Buttazzo, “Optimal selection of preemption points to minimize preemption overhead,” in *Proceedings of the 23rd Euromicro Conference on Real-Time Systems*, ECRTS ’11, Porto, Portugal, 2011, pp. 217–227.

[44] B. Peng, N. Fisher, and M. Bertogna, "Explicit preemption placement for real-time conditional code," in *Proceedings of the 26th Euromicro Conference on Real-Time Systems*, ECRTS '14, Madrid, Spain, 2014, pp. 177–188.

[45] S. Iwasaki, A. Amer, K. Taura, S. Seo, and P. Balaji, "BOLT: Optimizing OpenMP parallel regions with user-level threads," in *Proceedings of the 28th International Conference on Parallel Architectures and Compilation Techniques*, PACT '19, Seattle, Washington, USA, 2019, pp. 29–42.

[46] X. Tian, J. P. Hoeflinger, G. Haab, Y.-K. Chen, M. Girkar, and S. Shah, "A compiler for exploiting nested parallelism in OpenMP programs," *Parallel Computing*, vol. 31, no. 10-12, pp. 960–983, 2005.

[47] G. Marc, A. Eduard, M. Xavier, L. Jesús, N. Nacho, and O. José, "NanosCompiler: Supporting flexible multilevel parallelism exploitation in OpenMP," *Concurrency: Practice and Experience*, vol. 12, no. 12, pp. 1205–1218, 2000.

[48] Y. Tanaka, K. Taura, M. Sato, and A. Yonezawa, "Performance evaluation of OpenMP applications with nested parallelism," in *Proceedings of the Fifth International Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers*, LCR '00, Rochester, New York, USA, 2000, pp. 100–112.

[49] F. Broquedis, N. Furmento, B. Goglin, P.-A. Wacrenier, and R. Namyst, "ForestGOMP: An efficient OpenMP environment for NUMA architectures," *International Journal of Parallel Programming*, vol. 38, no. 5, pp. 418–439, 2010.

[50] A. Castelló, S. Seo, R. Mayo, P. Balaji, E. S. Quintana-Ortí, and A. J. Peña, "GLTO: On the adequacy of lightweight thread approaches for OpenMP implementations," in *Proceedings of the 46th International Conference on Parallel Processing*, ICPP '17, Bristol, UK, 2017, pp. 60–69.

[51] M. Danelutto, T. De Matteis, D. De Sensi, and M. Torquati, "Evaluating concurrency throttling and thread packing on SMT multicores," in *Proceedings of the 25th Euromicro International Conference on Parallel, Distributed and Network-Based Processing*, PDP '17, St. Petersburg, Russia, 2017, pp. 219–223.

[52] J. Park, S. Park, M. Han, and W. Baek, "POSTER: The performance impact of thread packing on synchronization-intensive applications," in *Proceedings of the 28th International Conference on Parallel Architectures and Compilation Techniques*, PACT '19, Seattle, Washington, USA, 2019, pp. 461–462.

[53] C. Iancu, S. Hofmeyr, F. Blagojević, and Y. Zheng, "Oversubscription on multicore processors," in *Proceedings of the 24th IEEE International Parallel and Distributed Processing Symposium*, IPDPS '10, Atlanta, Georgia, USA, 2010.

[54] S. Hofmeyr, C. Iancu, and F. Blagojević, "Load balancing on speed," in *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, Bangalore, India, 2010, pp. 147–158.

[55] J.-P. Lozi, B. Lepers, J. Funston, F. Gaud, V. Quéma, and A. Fedorova, "The Linux scheduler: A decade of wasted cores," in *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, London, UK, 2016.

[56] A. Hori, M. Si, B. Gerofi, M. Takagi, J. Dayal, P. Balaji, and Y. Ishikawa, "Process-in-process: Techniques for practical address-space sharing," in *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '18, Tempe, Arizona, USA, 2018, pp. 131–143.

[57] H. Carter Edwards and C. R. Trott, “Kokkos: Enabling performance portability across manycore architectures,” in *Proceeding of the 2013 Extreme Scaling Workshop*, XSW ’13, Boulder, Colorado, USA, 2013, pp. 18–24.

[58] H. C. Edwards, C. R. Trott, and D. Sunderland, “Kokkos: Enabling manycore performance portability through polymorphic memory access patterns,” *Journal of Parallel and Distributed Computing*, vol. 74, no. 12, pp. 3202–3216, 2014.

[59] E. Dirand, L. Colombet, and B. Raffin, “TINS: A task-based dynamic helper core strategy for in situ analytics,” in *Proceedings of Supercomputing Frontiers: 4th Asian Conference*, SFCA ’18, Cham, Singapore, 2018, pp. 159–178.

[60] Y.-K. Kwok and I. Ahmad, “Static scheduling algorithms for allocating directed task graphs to multiprocessors,” *ACM Computing Surveys*, vol. 31, no. 4, pp. 406–471, 1999.

[61] E. Ayguadé, N. Copty, A. Duran, J. Hoeflinger, Y. Lin, F. Massaioli, X. Teruel, P. Unnikrishnan, and G. Zhang, “The design of OpenMP tasks,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 20, no. 3, pp. 404–418, 2008.

[62] Y. Guo, J. Zhao, V. Cave, and V. Sarkar, “SLAW: A scalable locality-aware adaptive work-stealing scheduler,” in *Proceedings of the 24th IEEE International Parallel and Distributed Processing Symposium*, IPDPS ’10, Atlanta, Georgia, USA, 2010.

[63] 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 *Proceedings of the 2018 IEEE International Symposium on Workload Characterization*, IISWC ’18, 2018, pp. 59–70.

[64] S.-J. Min, C. Iancu, and K. Yelick, “Hierarchical work stealing on manycore clusters,” in *Proceedings of the Fifth Conference on Partitioned Global Address Space Programming Models*, PGAS ’11, Galveston Island, Texas, USA, 2011.

[65] J. Lifflander, S. Krishnamoorthy, and L. V. Kale, “Optimizing data locality for fork/join programs using constrained work stealing,” in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC ’14, New Orleans, Los Angeles, USA, 2014, pp. 857–868.

[66] Q. Chen, M. Guo, and Z. Huang, “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, San Servolo Island, Venice, Italy, 2012, pp. 163–172.

[67] Q. Chen, M. Guo, and H. Guan, “LAWS: Locality-aware work-stealing for multi-socket multi-core architectures,” in *Proceedings of the 28th ACM International Conference on Supercomputing*, ICS ’14, Munich, Germany, 2014, pp. 3–12.

[68] Q. Chen and M. Guo, “Locality-aware work stealing based on online profiling and auto-tuning for multisocket multicore architectures,” *ACM Transactions on Architecture and Code Optimization*, vol. 12, no. 2, 22:1–22:24, 2015.

[69] H. Zhao, Q. Chen, Y. Qiu, M. Wu, Y. Shen, J. Leng, C. Li, and M. Guo, “Bandwidth and locality aware task-stealing for manycore architectures with bandwidth-asymmetric memory,” *ACM Transactions on Architecture and Code Optimization*, vol. 15, no. 4, 55:1–55:26, 2018.

[70] J. Lifflander, S. Krishnamoorthy, and L. V. Kale, “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, Delft, The Netherlands, 2012, pp. 137–148.

[71] 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,” *Publications of the Astronomical Society of Japan*, vol. 68, no. 4, 54:1–54:22, 2016.

[72] J. Barnes and P. Hut, “A hierarchical $O(N \log N)$ force-calculation algorithm,” *Nature*, vol. 324, pp. 446–449, 1986.

[73] 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 Transactions on Architecture and Code Optimization*, vol. 11, no. 3, 30:1–30:25, 2014.

[74] J. Lifflander, S. Krishnamoorthy, and L. V. Kale, “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, Seattle, Washington, USA, 2013, pp. 507–518.

[75] M. Becker and M. Teschner, “Weakly compressible SPH for free surface flows,” in *Proceedings of the 2007 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA ’07, San Diego, California, USA, 2007, pp. 209–217.

[76] K. Taura, J. Nakashima, R. Yokota, and N. Maruyama, “A task parallel implementation of fast multipole methods,” in *Proceedings of the 2012 SC Companion: High Performance Computing, Networking Storage and Analysis – Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*, ScalA ’12, Salt Lake City, Utah, USA, 2012, pp. 617–625.

[77] G. J. Narlikar, “Scheduling threads for low space requirement and good locality,” *Theory of Computing Systems*, vol. 35, no. 2, pp. 151–187, 2002.

[78] 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 *Proceedings of the Nineteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA ’08, San Francisco, California, 2008, pp. 501–510.

[79] R. A. Chowdhury, F. Silvestri, B. Blakeley, and V. Ramachandran, “Oblivious algorithms for multicores and network of processors,” in *Proceedings of the 24th IEEE International Parallel and Distributed Processing Symposium*, IPDPS ’10, Atlanta, Georgia, USA, 2010.

[80] R. A. Chowdhury, V. Ramachandran, F. Silvestri, and B. Blakeley, “Oblivious algorithms for multicores and networks of processors,” *Journal of Parallel and Distributed Computing*, vol. 73, no. 7, pp. 911–925, 2013.

[81] H. V. Simhadri, G. E. Blelloch, J. T. Fineman, P. B. Gibbons, and A. Kyrola, “Experimental analysis of space-bounded schedulers,” in *Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA ’14, Prague, Czech Republic, 2014, pp. 30–41.

[82] ——, “Experimental analysis of space-bounded schedulers,” *ACM Transactions on Parallel Computing*, vol. 3, no. 1, 8:1–8:27, 2016.

[83] G. E. Blelloch, P. B. Gibbons, and H. V. Simhadri, "Provably efficient scheduling of dynamically allocating programs on parallel cache hierarchies," in *Proceedings of the 24th IEEE International Conference on High Performance Computing*, HiPC '17, Jaipur, India, 2017, pp. 124–133.

[84] J.-N. Quintin and F. Wagner, "Hierarchical work-stealing," in *Proceedings of the 16th European Conference on Parallel Processing*, Euro-Par '10, Ischia, Italy, 2010, pp. 217–229.

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

[86] L. Breiman, J. Friedman, C. J. Stone, and R. A. Olshen, *Classification and Regression Trees*. CRC Press, 1984.

[87] G. Ke, Q. Meng, T. Finley, T. Wang, W. Chen, W. Ma, Q. Ye, and T.-Y. Liu, "LightGBM: A highly efficient gradient boosting decision tree," in *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS '17, Long Beach, California, USA, 2017, pp. 3149–3157.

[88] K. Alsabti, S. Ranka, and V. Singh, "CLOUDS: A decision tree classifier for large datasets," in *Proceedings of the 4th International Conference on Knowledge Discovery and Data Mining*, KDD '98, New York, New York, USA, 1998, pp. 2–8.

[89] R. Jin and G. Agrawal, "Communication and memory efficient parallel decision tree construction," in *Proceedings of the 2003 SIAM International Conference on Data Mining*, SDM '03, San Francisco, California, USA, 2003, pp. 119–129.

[90] D. Dua and C. Graff, *UCI machine learning repository*, 2017. [Online]. Available: http://archive.ics.uci.edu/ml.

[91] P. Baldi, P. Sadowski, and D. Whiteson, "Searching for exotic particles in high-energy physics with deep learning," *Nature Communications*, vol. 5, no. 4308, pp. 1–9, 2014.

[92] *Intel 64 and IA-32 architectures software developer's manual – Volume 3B: System programming guide, part 2*, Intel Corporation, 2020.

[93] *Intel Xeon processor scalable memory family uncore performance monitoring*, Intel Corporation, 2017.

[94] B. D. Marsh, M. L. Scott, T. J. LeBlanc, and E. P. Markatos, "First-class user-level threads," in *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, SOSP '91, Pacific Grove, California, USA, 1991, pp. 110–121.

[95] E. D. Polychronopoulos, X. Martorell, D. S. Nikolopoulos, J. Labarta, T. S. Papatheodorou, and N. Navarro, "Kernel-level scheduling for the nano-threads programming model," in *Proceedings of the 12th International Conference on Supercomputing*, ICS '98, Melbourne, Australia, 1998, pp. 337–344.

[96] D. Stein and D. Shah, "Implementing lightweight threads," in *Proceedings of the USENIX Summer 1992 Technical Conference*, USENIX '92, San Antonio, Texas, USA, 1992, pp. 1–9.

[97] Sun Microsystems, Inc. (2002). "Multithreading in the Solaris operating environment," [Online]. Available: https://web.archive.org/web/20090226174929/http://www.sun.com/software/whitepapers/solaris9/multithread.pdf.

[98] C. S. Zakian, T. A. Zakian, A. Kulkarni, B. Chamith, and R. R. Newton, "Concurrent Cilk: Lazy promotion from tasks to threads in C/C++," in *Proceedings of the 28th International Workshop on Languages and Compilers for Parallel Computing*, LCPC '15, Raleigh, North Carolina, USA, 2015, pp. 73–90.

[99] A. Hori, B. Gerofi, and Y. Ishikawa, "An implementation of user-level processes using address space sharing," in *Proceedings of the 2020 IEEE International Parallel and Distributed Processing Symposium Workshops*, IPDPSW '20, New Orleans, LA, USA, 2020, pp. 976–984.

[100] A. Welc, A. L. Hosking, and S. Jagannathan, "Preemption-based avoidance of priority inversion for Java," in *Proceedings of the 33rd International Conference on Parallel Processing*, ICPP '04, Montreal, Quebec, Canada, 2004, pp. 529–538.

[101] A. Nilsson, T. Ekman, and K. Nilsson, "Real Java for real time - gain and pain," in *Proceedings of the 2002 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, CASES '02, Grenoble, France, 2002, pp. 304–311.

[102] S. Ghosh, M. Cuevas, S. Campanoni, and P. Dinda, "Compiler-based timing for extremely fine-grain preemptive parallelism," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '20, Atlanta, Georgia, USA, 2020.

[103] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '04, Palo Alto, California, USA, 2004, pp. 75–86.

[104] C. T. Haynes and D. P. Friedman, "Abstracting timed preemption with engines," *Computer Languages*, vol. 12, no. 2, pp. 109–121, 1987.

[105] R. K. Dybvig and R. Hieb, "Engines from continuations," *Computer Languages*, vol. 14, no. 2, pp. 109–123, 1989.

[106] P. Li, S. Marlow, S. Peyton Jones, and A. Tolmach, "Lightweight concurrency primitives for GHC," in *Proceedings of the ACM SIGPLAN Workshop on Haskell Workshop*, Haskell '07, Freiburg, Germany, 2007, pp. 107–118.

[107] A. Robison, M. Voss, and A. Kukanov, "Optimization via reflection on work stealing in TBB," in *Proceedings of the 22nd IEEE International Parallel and Distributed Processing Symposium*, IPDPS '08, Miami, Florida, USA, 2008.

[108] 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 *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*, PACT '16, Haifa, Israel, 2016, pp. 125–137.

[109] A. Pop and A. Cohen, "OpenStream: Expressiveness and data-flow compilation of openmp streaming programs," *ACM Transactions on Architecture and Code Optimization*, vol. 9, no. 4, 53:1–53:25, 2013.

[110] J. Maglalang, S. Krishnamoorthy, and K. Agrawal, "Locality-aware dynamic task graph scheduling," in *Proceedings of the 46th International Conference on Parallel Processing*, ICPP '17, Bristol, UK, 2017, pp. 70–80.

[111] K. Agrawal, C. E. Leiserson, and J. Sukha, “Executing task graphs using work-stealing,” in *Proceedings of the 24th IEEE International Parallel and Distributed Processing Symposium*, IPDPS ’10, Atlanta, Georgia, USA, 2010.

[112] R. M. Yoo, C. J. Hughes, C. Kim, Y.-K. Chen, and C. Kozyrakis, “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, Montréal, Québec, Canada, 2013, pp. 315–325.

[113] Y. Wang, Y. Zhang, Y. Su, X. Wang, X. Chen, W. Ji, and F. Shi, “An adaptive and hierarchical task scheduling scheme for multi-core clusters,” *Parallel Computing*, vol. 40, no. 10, pp. 611–627, 2014.

[114] J. Paudel, O. Tardieu, and J. N. Amaral, “On the merits of distributed work-stealing on selective locality-aware tasks,” in *Proceedings of the 42nd International Conference on Parallel Processing*, ICPP ’13, Lyon, France, 2013, pp. 100–109.

[115] W. Suksompong, C. E. Leiserson, and T. B. Schardl, “On the efficiency of localized work stealing,” *Information Processing Letters*, vol. 116, no. 2, pp. 100–106, 2016.

[116] G. J. Narlikar and G. E. Blelloch, “Space-efficient scheduling of nested parallelism,” *ACM Transactions on Programming Languages and Systems*, vol. 21, no. 1, pp. 138–173, 1999.

[117] S. Chen, P. B. Gibbons, M. Kozuch, V. Liaskovitis, A. Ailamaki, G. E. Blelloch, B. Falsafi, L. Fix, N. Hardavellas, T. C. Mowry, and C. Wilkerson, “Scheduling threads for constructive cache sharing on CMPs,” in *Proceedings of the Nineteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA ’07, San Diego, California, USA, 2007, pp. 105–115.

Appendix A.

Proof of Characteristics of Task Management in ADWS

This appendix chapter gives proof of theorems that characterize task management in ADWS. We denote a distribution range as $[i, j]$ $(i \leq j)$, where i and j are the worker ranks (integer values); worker i is the rightmost worker and worker j is the leftmost worker in the figures [1]. In this representation we omit the fractional parts of endpoints of a distribution range because fractional parts do not play an important role in proof. The owner of a distribution range $[i, j]$ 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 does not migrate tasks to itself). In the following, we denote the total number of workers in the system by P.

First, we introduce some lemmas to derive Theorem 1 that guarantees that migration of non-search-root tasks to each worker occurs sequentially.

Lemma 1. *Let i be an arbitrary number of workers except for 0 (*$i = 1, \ldots, P - 1$*). Suppose that worker j (*$0 \leq j < i$*) 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 a distribution range $[j, k]$ $(j < i \leq k)$ because it has the privilege of migrating non-search-root tasks to worker i. When worker j spawns a task, the distribution range is divided into two subranges $[j, l]$ and $[l, k]$ $(j \leq l \leq k)$ (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 without delegation. Otherwise, worker j executes the continuation with the new range $[j, l]$, and the spawned task is migrated to worker l. If $i \leq l$, 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 (*$0 \leq j < i$*) 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 $(j \leq k < i)$, 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. □

[1] 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.

Theorem 1. *Let i be an arbitrary number of workers except for 0 ($i = 1, \dots, P-1$). 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 to a worker is serialized; therefore, migration of non-search-root tasks does not require lock operations. Because of this, we can implement task queues without any locks or atomic operations for deterministic task allocation. We need locks for task queues if we enable hierarchical localized work stealing, but even if we use queues with locks, it is anticipated that 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 *distribution ranges* (see Section 4.2.4).

We have discussed the characteristic of non-search-root tasks above. Next, we discuss a characteristic of search-root tasks, deriving Theorem 2 that guarantees that only one search-root task stays at the mailbox of a worker at the same time.

Lemma 3. *Let i be an arbitrary number of workers ($i = 0, \dots, P-1$). 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 any search until T_i is completed.*

Proof. T_i must have range $[i, j]$ ($i < j$). 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 any successive search from $[k, i]$, workers can migrate only non-search-root tasks (with range $[i, i]$) to worker i. Worker k regains range $[k, j]$ only after T_i is completed. □

Theorem 2. *Let i be an arbitrary number of workers ($i = 0, \dots, P-1$). 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 migrated back to worker i only after worker i pops the previous search-root task. □

By Theorem 2, it is sufficient for each worker to have one buffer for migration of search-root tasks in the implementation. The buffer for migration of search-root tasks does not need to be protected by locks because stealing search-root tasks is not allowed.

Appendix B.

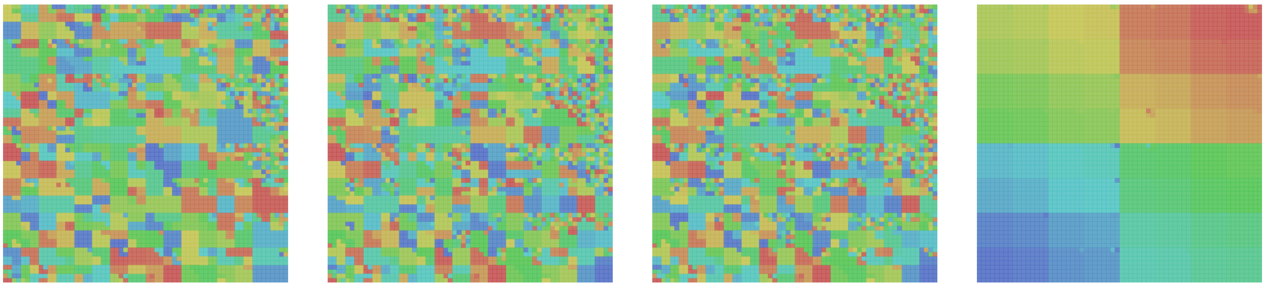
Detailed Performance Evaluation of Constrained WS

This appendix chapter gives more details about the performance evaluation of constrained work stealing (constrained WS) [65] conducted in Section 4.4.1. Although the original paper of constrained WS [65] reported good scalability for the heat benchmark, it did not scale well in our experimental settings. In the following, we discuss the performance difference by giving additional experimental results, which were measured measured using heat2D benchmark with $N = 4096$, 64 workers, and `numactl -iall` policy in the same settings as Section 4.4.1.

One of the main differences in the experimental settings between ours and theirs is the size of the matrices. Section 4.4.1 uses matrices whose size is close to the total cache size in the system, which is smaller than 32768×32768 in the original paper. Also, the cutoff size is 64×64, which is smaller than the original paper's settings (8192×64). In other words, tasks in our experimental setting are more fine-grained, so data locality among tasks close in the DAG is more likely to affect the performance. Considering NUMA-aware memory allocation, the leaf task size 64×64 is too small to finely control the physical mapping of data to tasks because of the page size, which is 4 KB in our environment. Because of this, data locality of constrained WS became worse than ADWS in our experiments.

Fig. B.1 visualizes the mapping of tasks to workers. In constrained WS, the first iteration is scheduled by random work stealing (Fig. B.1a). As *RelWS* proceeds, the DAG is decomposed into smaller parts and fine-grained tasks are randomly scattered to workers. In other words, the number of branches of *StealTree* monotonically increases as work stealing occurs in *RelWS*, leading to bad data locality. We can see the situation in Fig. B.1b and Fig. B.1c. On the other hand, Fig. B.1d indicates that in ADWS workers in the same NUMA node share tasks close in the DAG; therefore it has better locality even if tasks are fine-grained.

Reducing the number of iterations for *RelWS* will prevent the issue of over-decomposition of the DAG



(a) Constrained WS (iter: 0) (b) Constrained WS; RelWS (iter: 5) (c) Constrained WS; RelWS (iter: 20) (d) ADWS

Fig. B.1.: Visualization of task mapping of constrained WS and ADWS in heat2D benchmark. Each cell is a leaf task (64×64 size), the color of which represents the rank of the worker who executed it. The workers are colored gradually from blue to red in numerical order.

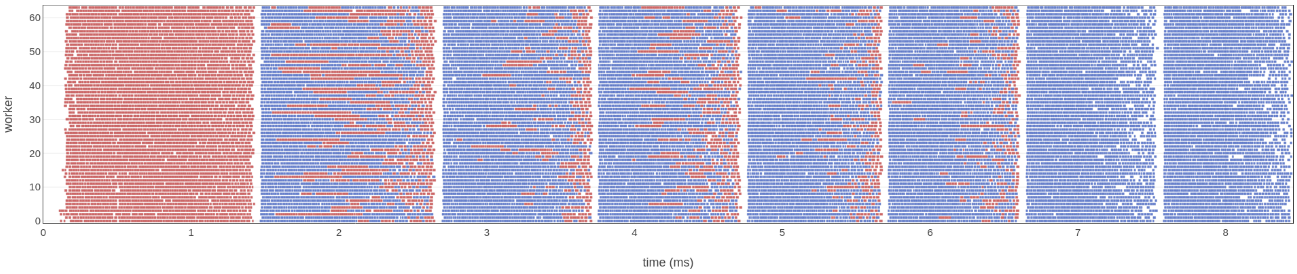


Fig. B.2.: Timeline of task scheduling by constrained WS. X axis is the time and Y axis is the worker. Each rectangle represents a leaf task of heat2D benchmark. The blue ones are tasks executed by the same worker as the previous iteration, whereas the red ones are tasks the worker did not execute at the previous iteration. The first iteration executes random work stealing, and the following five iterations execute *RelWS*. The last two iterations are *StOWS*; therefore tasks are not moved.

explained above, but there is a trade-off between data locality and idleness of workers. Too small number of iterations for *RelWS* can increase the idle time of workers. *StOWS* in constrained WS has to wait for tasks to be migrated, so it is vulnerable to the change of execution time for each task. Fig. B.2 is a timeline of how tasks were executed by workers. The blue rectangles represent leaf tasks executed by the same worker as the previous iteration, whereas the red ones are tasks the worker did not execute at the previous iteration. Since the red tasks have relatively bad data locality (because they cannot reuse cache in the previous iteration), the execution time of them should be longer than that of the blue tasks. If the next iteration was scheduled by *StOWS*, all of the red tasks became blue at the next iteration and the execution time got shorter. Then the timing when tasks are migrated changes, which causes additional idle time for workers in *StOWS*.

Fig. B.3 shows the cache miss ratio and the ratio of the red tasks in Fig. B.2 measured in heat2D benchmark. First, the cache miss ratio of constrained WS is more than double of that of ADWS, though it is quite smaller than that of random WS. We can see that the cache miss ratio of constrained WS increases as the number of iterations for *RelWS* increases (because of the over-decomposition issue). The ratio of the red tasks in Fig. B.2 is also shown in the same plot (the solid line). As *RelWS* repeats, the number of red tasks decreases and the scheduling converges. It also shows that even if we repeated *RelWS* for many iterations, at least about 5% of leaf tasks were moved across iterations. If the number of red tasks is small, the consecutive *StOWS* iterations have little idle time. However, as Fig. B.3 indicates, there is a clear trade-off between the idle time issue and the over-decomposition issue. If we increase the number of iterations for *RelWS*, the idle time decreases but cache misses increase, and vice versa.

Summarizing, in the heat2D benchmark with fine-grained settings, constrained WS underperformed ADWS because (i) constrained WS does not optimize the scheduling to be memory hierarchy-aware (i.e., it just tries to repeat the first iteration's task mapping, which is scheduled by work stealing that is oblivious to the memory hierarchy), and (ii) it adds additional idle time of workers because of variable execution times of tasks that are sensitive to data locality. Nevertheless, performance results in this thesis do not suggest that ADWS always performs better than constrained WS; it depends on the benchmark applications to use. For example, ADWS requires additional hints of work for each task, which may not be readily available. For applications where hints of work are hard to be estimated, constrained WS may perform better than ADWS since it does not require any hints. More detailed and comprehensive studies on the performance of various schedulers, including constrained WS, are needed as future work.

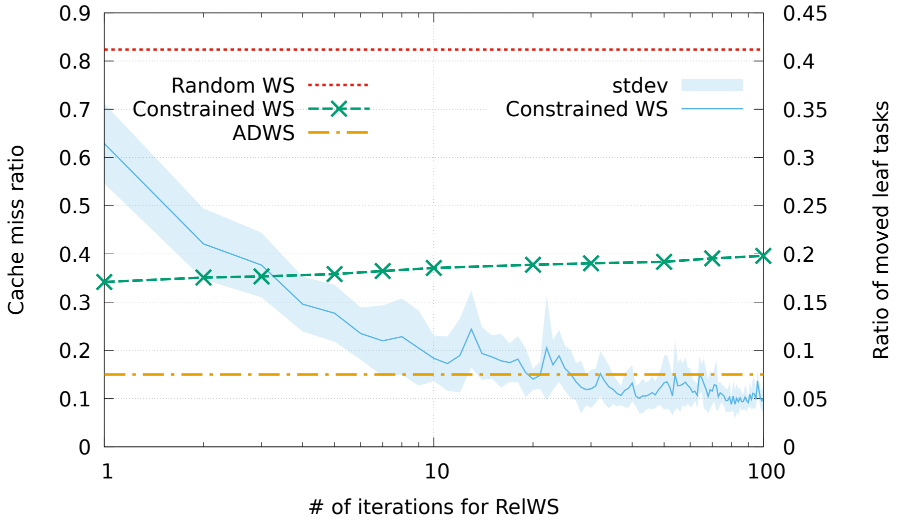


Fig. B.3.: Change of the cache miss ratio and the ratio of leaf tasks moved from the previous iteration according to the number of iterations for *RelWS*. The dotted lines show the cache miss ratio measured for 1000 iterations with a given number of iterations for *RelWS*. The solid line with the shaded region represents the ratio of leaf tasks moved from a worker to another at the iteration. The solid line is the average for ten executions and the shaded region is the standard deviation.