Distributed Continuation Stealing is More Scalable than You Might Think

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

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

***Abstract*—The need for load balancing in applications with irregular parallelism has motivated research on work stealing. An important choice in work-stealing schedulers is between *child stealing* or *continuation stealing*. In child stealing, a newly created task is made stealable by other processors, whereas in continuation stealing, the caller's continuation is made stealable by executing the newly created task first, which preserves the serial execution order. Although the benefits of continuation stealing have been demonstrated on shared memory by Cilk and other runtime systems, it is rarely employed on distributed memory, presumably because it has been thought to be difficult to implement and inefficient as it involves migration of call stacks across nodes. Akiyama and Taura recently introduced efficient RDMA-based continuation stealing, but the practicality of distributed continuation stealing is still unclear because a comparison of its performance with that of child stealing has not previously been performed.**

This paper presents the results of a comparative performance analysis of continuation stealing and child stealing on distributed memory. To clarify the full potential of continuation stealing, we first investigated various RDMA-based synchronization (task join) implementations, which had not previously been fully investigated. The results revealed that, when the task synchronization pattern was complicated, continuation stealing performed better than child stealing despite its relatively long steal latency due to stack migration. Notably, our runtime system achieved almost perfect scaling on 110,592 cores in an unbalanced tree search (UTS) benchmark. This scalability is comparable to or even better than that of state-of-the-art bag-of-tasks counterparts.

***Index Terms*—work stealing, fork-join parallelism, futures, work-first scheduling, task-based runtime systems, RDMA**

I. Introduction

Load balancing is crucial to performance in parallel computing, especially for algorithms that yield irregular parallelism. As manual load balancing is cumbersome and error-prone, runtime-level support for dynamic load balancing is substantially important. Many parallel runtime systems that efficiently support dynamic load balancing have been developed and used in production [1], [2], [3], [4], [5], [6], [7], [8].

Work stealing [9] is arguably the most widely used load balancing algorithm for irregular parallelism: each processor has a local task queue, from which other idle processors try to steal tasks by randomly selecting a "victim." An important scheduling choice in work stealing is between *child stealing* or *continuation stealing* [1] . In child stealing, a processor places a newly created child task in the local task queue to potentially be stolen by another processor and continues to run the parent task. In continuation stealing, the processor immediately executes the newly created child task and leaves the continuation of the parent task in the queue. Continuation stealing preserves the serial execution order as much as possible, because a task is executed as an ordinary function call unless its parent task is stolen. This helps to efficiently resolve task synchronization in fork-join programs ("join" primitives) and prove theoretical bounds on execution time, space, communication [9], and data locality [10], as further discussed in Section II. Because of these beneficial characteristics, many shared-memory runtime systems, including Cilk [1], [2], use continuation stealing.

On distributed memory, most of the existing work-stealing implementations are child stealing, and many child-stealing implementations have shown good scalability with the use of remote direct memory access (RDMA) [11], [12], [13]. The introduction of RDMA has eliminated the need for frequent interruptions to the victim processors, which are otherwise needed for steal attempts with two-sided (message-based) communication [14], [15]. Nevertheless, continuation stealing is still rare on distributed memory. This is presumably due to two assumptions:

1) Implementing continuation stealing as a library with native (unmodified) C/C++ compilers is difficult because it involves migration of call stacks across nodes.
2) Continuation stealing is less efficient than child stealing because a call stack is typically larger than the function pointer and its arguments needed for child stealing.

Akiyama and Taura [16], [17] challenged the first assumption by introducing an efficient stack migration scheme over RDMA that does not require special compiler support. However, the correctness of the second assumption is still unclear; there have been no reports in the literature of a performance comparison between continuation stealing and child stealing on distributed memory. The primary goal of this paper is to explore the true potential of distributed continuation stealing by comparing its performance with that of child stealing.

To determine the full potential of distributed continuation stealing, we first investigated efficient task synchronization (join) strategies over RDMA, which were not covered by previous work. In particular, the implementation by Akiyama and

[1]Child stealing is also called *help-first* or *parent-first* scheduling, and continuation stealing is also called *work-first* or *child-first* scheduling.

Taura does not allow for migration of tasks suspended at a join. Because they cannot be resumed by other processors even after they become ready for execution, available parallelism is reduced. Thus, we devised an RDMA-based technique to migrate and resume tasks suspended at a join immediately after they become ready, which is in fact a common practice in shared-memory implementations (e.g., Cilk). In addition, we devised an efficient memory management technique for dynamically allocated RDMA-accessible objects for task synchronization in continuation stealing. Our evaluation experiments showed that these join optimizations improved performance by up to 40% compared with the baseline.

Using a continuation-stealing runtime system implementing these two techniques, we conducted performance analysis of continuation stealing and child stealing. Although continuation stealing incurred longer steal latency than child stealing, the average steal latency was less than 20% higher than that of child stealing. Moreover, the experiments demonstrated that continuation stealing handled join primitives more efficiently, resulting in better overall performance than child stealing when the task synchronization pattern was complicated.

Furthermore, we used the unbalanced tree search (UTS) benchmark [18] to compare our continuation-stealing runtime implementation with existing task-parallel runtime systems (SAWS [12], Charm++ [19], [20], and X10/GLB [21], [22]). The results show that our runtime system was as scalable as or even more scalable than them. Notably, our continuation-stealing runtime system demonstrated 96.4% parallel efficiency on 110,594 cores. This is a surprising result because our UTS implementation is based on a straightforward fork-join parallelization of tree traversal, whereas other UTS implementations are based on a *bag-of-tasks* paradigm, in which task dependency cannot be described and global termination detection is needed.

The task implementation of our continuation-stealing runtime system is not only for fork-join primitives but also for more general *futures*. In other words, tasks do not have to be joined with their parent; the handler of a spawned task, called a future, can be passed to any other tasks. Thus, our runtime system can deal with a wider range of dependency patterns, which is demonstrated by the longest common subsequence (LCS) benchmark used in our evaluation. Our LCS benchmark makes heavy use of futures to represent the complicated dependency pattern (wavefront) by recursively decomposing blocks. The evaluation results show that the capability of task migration (both continuation stealing and task migration at joins) is indispensable to achieving good performance; lack of task migration led to worse performance by an order of magnitude. Our continuation-stealing runtime system with both task migration capabilities achieved the highest performance (close to the theoretical bounds) on nearly 10k cores.

II. Work Stealing

Work stealing [9] is arguably the most widely used load balancing algorithm today. We define *workers* as virtualized processors, which are typically substantiated as processes or kernel-level threads. In work stealing, each worker maintains its own task queue, from which and to which the local worker pops and pushes tasks. When the local queue is empty, the worker attempts to steal a task from another worker by selecting victims uniformly at random. Typically, the task queue is implemented as a double-ended queue (we assume that the *THE* protocol [2] is used); the worker pushes tasks to and pops tasks from the same end of the queue, whereas other workers steal tasks from the other end of the queue. Hence, local push/pop operations follow last-in first-out (LIFO) order, and steal operations follow first-in first-out (FIFO) order. Thus, the oldest task in the queue, which is expected to have the largest amount of work, is always stolen.

In the following of this section, we first discuss pros and cons about child stealing and continuation stealing in Section II-A and demonstrate by an example a shortcoming in child stealing when handling task join primitives in Section II-B. We then explain the current status of support for thread migration on distributed memory in Section II-C and finally introduce prior work on distributed continuation stealing, *uni-address threads*, in Section II-D.

A. Child Stealing and Continuation Stealing

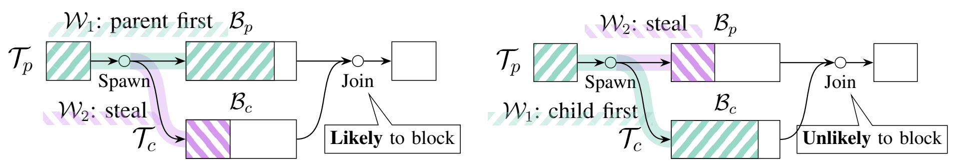
As noted above, an important choice in work stealing is between *child stealing* or *continuation stealing*, which are illustrated in Fig. 1. Worker $\mathcal{W}_1$ first runs task $\mathcal{T}_p$, which spawns child task $\mathcal{T}_c$. In child stealing, $\mathcal{W}_1$ continues executing $\mathcal{T}_p$ after spawning $\mathcal{T}_c$ and pushes $\mathcal{T}_c$ to the local task queue. In continuation stealing, $\mathcal{W}_1$ first executes $\mathcal{T}_c$ upon spawning, leaving the continuation of $\mathcal{T}_p$ subject to work stealing.

The benefit of child stealing is its simple and portable implementation [23]. Each stealable task can be simply represented as a function pointer (or a serialized closure) and its arguments because the task is stolen before execution has begun. This simplicity is the reason many shared-memory tasking runtime systems use child stealing [23], [24], [3], [25]. In contrast, the implementation of continuation stealing is more involved because stealing the continuation of an already started task is not trivial. Many efforts have been put into compiler support for continuation stealing [26], [1], [2] and user-level threading libraries [7], [27] to save and restore continuation states.

Despite its implementation difficulties, continuation stealing nevertheless has benefits. One benefit is that it preserves the serial execution order (i.e., the program order with task-related keywords removed). This helps to prove good asymptotic bounds on execution time, space, communication [9], and data locality [10]. Blumofe and Leiserson [9] proved that the execution time of continuation stealing on P processors is bounded by $T_P \leq T_1/P + O(T_\infty)$, where T_1 is the total amount of work (i.e., the serial execution time) and T_∞ is the critical path length (span) of the task graph (i.e., the ideal execution time on an infinite number of processors). This bound is analogous to the *greedy-scheduling theorem* or *Brent's theorem* [28] ($T_P \leq T_1/P + T_\infty$) for an ideal *greedy* scheduler. A scheduler is greedy if any pair of a ready task and an idle worker does not exist at any point in time, i.e., an

```
thread $\mathcal{T}_p$ = spawn([=]{
  ...
  thread $\mathcal{T}_c$ = spawn([=]{ $\mathcal{B}_c$; });
  $\mathcal{B}_p$;
  $\mathcal{T}_c$.join();
  ...
});
```

(a) Pseudocode



(b) Child stealing (c) Continuation stealing

Fig. 1. Illustration of child stealing and continuation stealing. Parent task $\mathcal{T}_p$ spawns task $\mathcal{T}_c$, computes $\mathcal{B}_p$, and joins $\mathcal{T}_c$. Child task $\mathcal{T}_c$ computes $\mathcal{B}_c$ and exits. Worker $\mathcal{W}_1$ first executes $\mathcal{T}_p$ and moves on to $\mathcal{B}_p$ or $\mathcal{T}_c$ upon spawning. $\mathcal{W}_2$ can steal either $\mathcal{T}_p$ or $\mathcal{T}_c$ left in the task queue.

idle worker immediately executes a ready task if one exists, which is an important property in general to keep processors as busy as possible.

B. Handling Task Synchronization (Join)

How to join tasks is as important as how to steal tasks in task-parallel programs, but child stealing is problematic for handling join. This is because child stealing is likely to be suspended at a join because the parent is preferably executed rather than the child. As shown in Fig. 1b, because the join for $\mathcal{T}_c$ belongs to the parent, $\mathcal{T}_p$, the parent-first order is likely to reach the join point before the child is completed. Note that reaching an unresolved join itself is not a problem. Unless $\mathcal{T}_c$ is stolen, $\mathcal{W}_1$ can immediately execute $\mathcal{T}_c$ by popping it from the local task queue at the join point [23]. Then, once $\mathcal{T}_c$ is completed, $\mathcal{W}_1$ returns to the join point and executes the continuation of $\mathcal{T}_p$. Along this path, the continuation of the join is executed immediately after it becomes runnable.

A problem arises when $\mathcal{T}_c$ is stolen by another worker ($\mathcal{W}_2$ in the figure). Supposing that $\mathcal{B}_p$ and $\mathcal{B}_c$ have the same amount of work, $\mathcal{W}_1$ is likely to complete $\mathcal{B}_p$ before $\mathcal{W}_2$ completes $\mathcal{B}_c$ because of the steal delay. Thus, $\mathcal{W}_1$ is likely to reach the join point before $\mathcal{T}_c$ is completed. $\mathcal{W}_1$ thus initiates work stealing by suspending $\mathcal{T}_p$ at the join. We call such an unresolved join resulting from a steal event an *outstanding join*. An outstanding join may not be resumable even after $\mathcal{T}_c$ is completed by $\mathcal{W}_2$ if $\mathcal{W}_1$ is busy running another task and migration of suspended tasks is impossible. In child stealing, it is typically impossible to migrate an already started task to another worker, because of its simple task representation. Thus, child stealing can yield many outstanding joins, most of which may not be immediately resumed, which can severely limit available parallelism.

Conversely, continuation stealing is unlikely to yield an outstanding join. Moreover, even if an outstanding join appears, it can be immediately resumed because continuation stealing should already support dynamic task migration.

C. Suspending and Migrating Threads

Even aside from work stealing, the generic suspension capability for tasks is important, as evidenced by the extensive research on the suspension capability of user-level threads on shared memory [6], [7], [8], [29], [27], [30]. In this paper, we use the terms “tasks” and “threads” interchangeably. The suspension capability is also useful for efficiently implementing a generic *yield* operation and other synchronization mechanisms (e.g., locks, barriers, and condition variables).

Many child-stealing runtime systems that operate on distributed memory, including Scioto [31], [11], SAWS [12], and X10/GLB [21], [22], are based on the bag-of-tasks (BoT) paradigm and their tasks cannot be suspended. Several distributed child-stealing runtime systems, including Satin [32], KAAPI [33], HotSLAW [13], and Grappa [34], support suspension at thread synchronization (e.g., at joins), but their thread implementations are *tied* tasks, which means that a task cannot be migrated to another worker once it starts to run. Thus, they are also subject to the performance problem caused by outstanding joins. Compared with the many user-level thread implementations on shared memory, thread suspension on distributed memory has not been well investigated.

D. Uni-Address Threads

A naive approach to achieving thread migration across nodes, without special compiler support, is to copy a thread stack to the same virtual address on the target node. The virtual address of the stack has to be preserved because the stack may contain pointers to local stack variables. In the *iso-address* scheme [35], which is used in Charm++ [19], [20] and Adaptive MPI [36], a globally unique virtual address is assigned to each stack in order to prevent virtual address overlapping. However, this can consume a huge amount of virtual address space, which is problematic in RDMA because memory for thread stacks would need to be pinned in advance.

To reduce address space consumption, the *uni-address* scheme was devised by Akiyama and Taura [16], [17]. The idea is to place thread stacks at the same virtual address *only while threads are running*. When a thread is suspended, its stack is *evacuated* to an arbitrary virtual address to make room for another thread to run. When the suspended thread is resumed, its stack is brought back to the virtual address to which it was first allocated. Therefore, the virtual addresses of thread stacks are no longer globally unique, reducing overall virtual address consumption. Note that this does not mean that threads at the same virtual address cannot be executed at the same time as each worker has its own address space. We call a memory region for running threads a *uni-address region* and one for suspended threads an *evacuation region*. Both regions are pinned to physical memory accessible by RDMA. Workers need to have the same virtual address for the uni-address region, but not for the evacuation region.

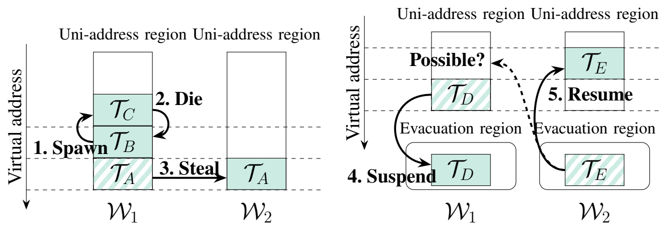


Fig. 2. Stack management in uni-address threads.

One key point of the uni-address scheme is that it is optimized for continuation stealing. To avoid frequent copying of thread stacks between the uni-address region and the evacuation region, a new stack is placed on top of the current thread stack so that the virtual addresses of thread stacks with a parent-child relationship do not overlap. Fig. 2 illustrates the flow of continuation stealing with uni-address threads.

1. Spawn: When $\mathcal{T}_B$ creates a new thread $\mathcal{T}_C$, $\mathcal{T}_C$'s stack is placed immediately above $\mathcal{T}_B$'s stack. The virtual address and size of $\mathcal{T}_B$'s stack are recorded in the task queue to be potentially stolen.

2. Die: When $\mathcal{T}_C$ is completed, the execution of $\mathcal{T}_B$ is resumed in the same way as an ordinary subroutine returns, unless $\mathcal{T}_B$ has already been stolen by another worker.

3. Steal: When $\mathcal{W}_2$ steals $\mathcal{T}_A$ from $\mathcal{W}_1$, the call stack of $\mathcal{T}_A$ at $\mathcal{W}_1$ is copied to the same virtual address at $\mathcal{W}_2$, preserving pointers to local stack variables.

4. Suspend: When $\mathcal{T}_D$ is blocked due to an unresolved join, for example, $\mathcal{T}_D$'s call stack is temporarily moved to any virtual address in the evacuation region.

5. Resume: When $\mathcal{T}_E$ is resumed, $\mathcal{T}_E$'s stack is moved from the evacuation region to the virtual address to which it was previously assigned.

The original uni-address thread implementation has shortcomings at joins; e.g., thread migration at joins (indicated by the dashed arrow in Fig. 2) is not allowed.

III. Joining Threads over RDMA

As briefly mentioned in Section II-D, efficient join implementations on distributed memory have not been well investigated. This section first examines different joining strategies in Section III-A and then proposes an efficient memory management technique in Section III-B.

A. Joining Strategy

In general, joining strategies for work stealing can be classified as either *stalling* or *greedy* [37]. In the stalling strategy, threads are not migrated to other workers across a join, whereas in the greedy strategy, threads can be migrated to other workers so that they are resumed immediately after becoming ready. On shared memory, for example, Intel TBB [3] uses the stalling strategy and Cilk uses the greedy strategy.

We introduce several notations to explain our algorithms. The *location* of a variable specifies where the variable is; it typically consists of the worker ID (rank) of the owner, the virtual address, and the size. Type $Loc(T)$ specifies the location of a variable of type T. The "**get** $v \leftarrow L$" statement copies the remote variable at location L to local variable v, and the "**put** $L \leftarrow v$" statement copies local variable v to the remote variable at L. The "**fetch_and_add** (L, v)" statement atomically adds v to the remote variable at L and returns the original value. The "**Suspend** *context* **do**" block first saves the current execution state to *context* and then executes the statements in the block on a separate context (stack). The saved context can be resumed by executing the "**resume** *context*" statement, which discards the current execution context.

```
Struct ThreadEntry
    retval :: return value of the joined thread function
    flag :: completion status of the joined thread (default: 0)
Function Die(E :: Loc(ThreadEntry), retval)
    put E.retval ← retval
    put E.flag ← 1
    nextThread ← PopFromThreadQueue()
    if nextThread is found then
        resume nextThread.context
    else
        resume scheduler.context
Function Join(E :: Loc(ThreadEntry))
    get f ← E.flag
    while f = 0 do // The joined thread has not been completed.
        suspend context do
            PushToWaitQueue(context)
            resume scheduler.context
        get f ← E.flag
    get retval ← E.retval
    FreeRemote(E)
    return retval
```

Fig. 3. Implementation of *stalling* join over RDMA.

1) Stalling Join: Fig. 3 shows the stalling join implementation in the original uni-address threads [16]. In the following, without loss of generality, we consider only two threads: a *joining* thread and a *joined* thread. The joining thread waits for completion of the joined thread by the Join function, and the joined thread calls the Die function when completed. In continuation stealing, the joining thread does not know where the joined thread is, and vice versa, as threads can be dynamically migrated. Thus, we allocate a *thread entry* to the memory where the joined thread was originally spawned, so that the joined thread and joining thread can communicate via this RDMA-accessible thread entry. A thread entry's location is used as a thread handler, and both the Die and Join function receive it as an argument. The type ThreadEntry has two fields: *retval*, the return value of the thread, and *flag*, which indicates the exit status of the joined thread.

When the joined thread is completed, it first puts the return value (line 5) and then sets the flag in the thread entry (line 6). The joining thread waits for the flag to be set in a loop (lines 14–18), after which it gets the return value of the joined thread (line 19). Each time the flag check fails, it switches to the scheduler context to initiate work stealing; at the same time, its current context is saved and pushed into the *wait queue*. A wait queue is a per-worker FIFO queue in which

```
22 Struct ThreadEntry
23     $retval$ :: return value of the joined thread function
24     $flag$ :: atomic flag for greedy join (default: 0)
25     $ctxloc$ :: location of the saved context of the joining thread
26 Function Die($E$ :: $Loc$(ThreadEntry), $retval$)
27     put $E.retval \leftarrow retval$
28     $nextThread \leftarrow$ PopFromThreadQueue()
29     if $nextThread$ is found then // The parent has not been stolen.
30         put $E.flag \leftarrow 1$
31         resume $nextThread.context$
32     else // The parent has already been stolen.
33         $f \leftarrow$ fetch_and_add $(E.flag, 1)$
34         if $f = 0$ then // The joined thread won the race.
35             resume $scheduler.context$
36         else // The joined thread lost the race.
37             get $C \leftarrow E.ctxloc$
38             get $context \leftarrow C$
39             FreeRemote($C$)
40             resume $context$
41 Function Join($E$ :: $Loc$(ThreadEntry))
42     get $f \leftarrow E.flag$
43     if $f = 0$ then // The joined thread seems to be running.
44         suspend $context$ do
45             put $E.ctxloc \leftarrow$ LocationOf($context$)
46             $f \leftarrow$ fetch_and_add $(E.flag, 1)$
47             if $f = 0$ then // The joining thread won the race.
48                 resume $scheduler.context$
49             else // The joining thread lost the race.
50                 resume $context$
51     get $retval \leftarrow E.retval$
52     FreeRemote($E$)
53     return $retval$
```

Fig. 4. Implementation of *greedy* join over RDMA.

suspended threads are stored. Each time a steal attempt fails, the scheduler pops a thread from the wait queue and resumes its execution in a round-robin fashion. A suspended thread cannot be executed until it is popped from the wait queue, even if it becomes runnable, which makes this scheduler nongreedy.

2) Greedy Join: In the greedy join implementation, the continuation of a join is executed by the worker who runs the joined thread or the joining thread, whichever reaches the synchronization point later. Therefore, the continuation of a join is executed immediately after it becomes runnable. While Cilk uses a mutex for implementing greedy join, the use of atomic operations on shared memory has been investigated [38], [39]. Borrowing this idea, we consider an efficient greedy join implementation using RDMA atomic operations.

Fig. 4 shows our greedy join implementation. The basic idea is that the two threads race on an atomic variable, and the worker who runs the loser (who executes the atomic operation later) runs the continuation of the join. However, calling an RDMA atomic operation at every join and die can be too costly. To avoid frequent atomic operations, we devised a technique based on the *work-first principle* [2]. The idea is for the joined thread to try to pop the parent thread from the queue (line 28) before racing with the joining thread. Because we assume continuation stealing and the oldest-first stealing strategy of the queue, the popped thread is guaranteed to always be the parent of the joined (completed) thread [9]. If the parent thread is successfully popped, the corresponding join is guaranteed to occur only after the joined thread has died. Therefore, we can simply set the flag without atomic operations (lines 29–31), which is in turn read by the same worker at line 43. Along this fast path, the Join function can be completed without suspending its execution.

If the parent thread has already been stolen when the joined thread is completed, the execution moves to the slow path (lines 32–40). Because the return value has been written at line 27, an atomic fetch-and-add is executed to race with the joining thread (line 33). If the joined thread wins the race, it simply returns to the scheduler to initiate work stealing (lines 34–35). Otherwise, it resumes the continuation of the join (lines 36–40) because the joining thread has already been suspended at the join. The joining thread performs the atomic fetch-and-add (line 46) after saving its current context and putting its location in the thread entry (line 45) so that the joined thread can obtain the context immediately after the race (lines 37–38). On rare occasions, the joining thread loses the race even after it sees that the flag is unset (line 43) and suspends its execution, in which case it resumes the continuation of the join by itself (line 50).

Note that this algorithm is not limited to fork-join but is also applicable to *futures* [40], [41]; the joining thread does not need to be the parent of the joined thread. One limitation in this implementation is that a future can have only one consumer, but this limitation will be slightly relaxed in Section V-D.

B. Freeing Remote Objects

In continuation stealing, thread entries and suspended threads can be freed remotely by any worker; we call such RDMA-accessible objects *remote objects*. The FreeRemote function in Fig. 3 and Fig. 4 receives a remote object's location and frees it (possibly) remotely. In the original implementation by Akiyama and Taura, each worker has its own lock-protected incoming queue to receive locations of remotely freed objects. When freeing an object remotely, it acquires the lock of the target queue, increments the counter, inserts the object location into the buffer, and releases the lock; this operation involves four round trips. When a worker collects remotely freed objects in its local queue, it acquires the lock, frees all received objects, sets the counter to zero, and releases the lock.

The key to improved performance is to move the overheads of remote workers to the local worker because the cost of local operations is much lower than that of remote operations. Thus, we introduce an optimization called *local collection*. In local collection, all remote objects are managed in a doubly linked list by the local worker. When a remote object is newly allocated, it is added to the list. When a remote object is freed locally, it is immediately removed from the list and freed. When it is freed remotely, its free bit is set by a remote put operation in a nonblocking manner. When the total size of the allocated remote objects exceeds a limit, the worker iterates over the list and frees remote objects with a free bit set. This strategy eliminates locks and involves only one communication

TABLE I
EXPERIMENTAL ENVIRONMENT.

	ITO-A	WISTERIA-O
Processor	Intel Xeon Gold 6154	Fujitsu A64FX
Architecture	Skylake-SP	ARMv8.2-A + SVE
Frequency	3.0 GHz (Turbo 3.7 GHz)	2.2 GHz
# of cores	36 (18 × 2 sockets)	48
Memory	DDR4 (192 GiB)	HBM2 (32 GiB)
Interconnect	InfiniBand EDR 4x (100 Gbps)	Tofu Interconnect-D
Compiler	GCC 11.2.0	Fujitsu compiler 4.5.0
Compile opts	`-O3 -march=native`	`-O3 -Nclang`
MPI	Open MPI 5.0.x	Fujitsu MPI 4.0.1
OS	RHEL 7.3	RHEL 8.3

```
PFor(int n) {
  for (int k = 0; k < K; k++) {
    parallel_for (int i = 0; i < n; i++)
      compute(M); // run for duration of M
  }
}
```

```
RecPFor(int n) {
  if (n == 1) {
    compute(M);
  } else {
    PFor(n);
    thread th = spawn([=] { RecPFor(n / 2); });
    RecPFor(n / 2);
    th.join();
  }
}
```

Fig. 5. Pseudocode of PFor and RecPFor benchmarks.

for freeing a remote object (which can even be nonblocking), which greatly reduces the cost of remote workers.

IV. EVALUATION METHODOLOGY

Our primary interest here is performance comparison between different joining strategies (stalling vs. greedy join) and stealing strategies (child vs. continuation stealing). To analyze their performance, we implemented child stealing in our runtime library by mimicking the prevalent distributed child-stealing implementations (Section IV-B), in addition to continuation stealing with both stalling and greedy join. We use synthetic benchmarks (Section IV-C) to explore the detailed performance characteristics and more realistic applications (the UTS and LCS benchmarks) for evaluation.

A. Experimental Settings

Our implementation is based on a C++ library *MassiveThreads/DM* (an implementation of uni-address threads) originally developed by Akiyama and Taura [16], [17]. To achieve better portability, we modified it to use MPI-3 RMA [42] as a backend communication library, as high-quality implementations for MPI-3 RMA are available today. In addition, we fixed a few trivial performance issues in MassiveThreads/DM and used it as the baseline for evaluation.

The experimental environment is summarized in TABLE I. ITO-A is ITO supercomputer (subsystem A) at Kyushu University [43], which is comprised of Intel Skylake processors and InfiniBand Interconnect, and WISTERIA-O is Wisteria/BDEC-01 supercomputer (Odyssey subsystem) at the University of Tokyo [44], comprised of Fujitsu A64FX processors and Tofu interconnect-D developed for Fugaku supercomputer [45]. For ITO-A, we built a nightly version (5.0.x) of Open MPI (Git commit hash: `85fec3a74fee`) with UCX [46] 1.11.0 and the `osc_ucx_acc_single_intrinsic` option; it has better one-sided communication performance than other MPI implementations [47]. When allocating nodes in WISTERIA-O, we specified a 3D mesh topology as close to a cube as possible, because it showed the best performance for random work stealing. To obtain stable performance results, we executed warm-up runs until the performance results stabilized. Each point in the following plots represents an average of 100 runs with an error bar representing the 95% confidence interval unless explicitly noted.

B. Implementations of Child Stealing

Assuming that thread implementations in child stealing are tied tasks as discussed in Section II, we implemented only stalling join for child stealing. We further classify stalling join implementations into two types depending on the thread suspension capability, *run-to-completion (RtC) threads* and *fully fledged (Full) threads*, following the taxonomy of Iwasaki et al. [29], [27].

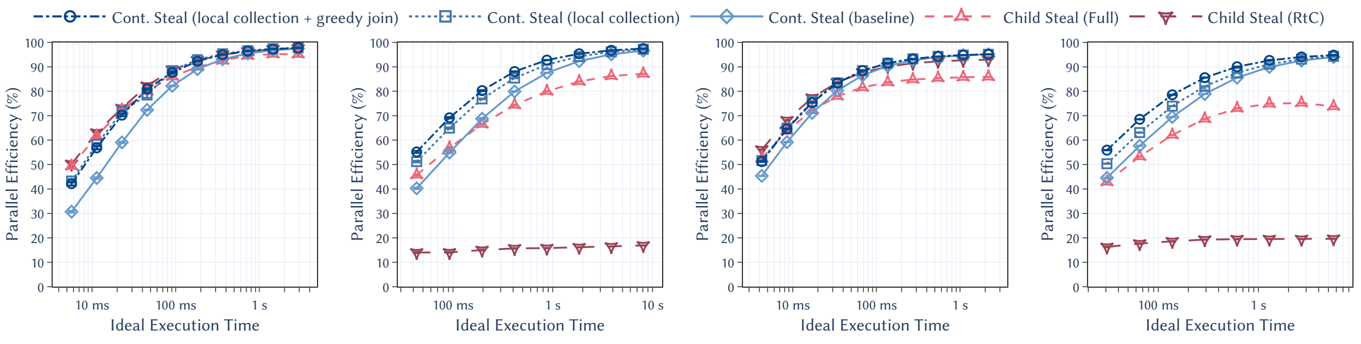
RtC threads are simply realized by ordinary function calls. When a RtC thread encounters an unresolved join, it calls the scheduler function directly on top of its stack to find another runnable task. If a runnable task is found, the task is executed on top of the stack as in ordinary function calls. Meanwhile, the unresolved join is "buried" and cannot be resumed until all the tasks running on top of it are completed. This "buried join" problem [25] can substantially reduce available parallelism.

Full threads avoid the buried join problem because each thread has its own stack. When a task begins, it is assigned a new separate stack (32 KB in our experiments). When encountering an unresolved join, it context-switches to another thread by suspending itself (using assembly instructions). Suspended threads are managed in the wait queue (Section III-A).

C. Synthetic Benchmarks

Fig. 5 shows the pseudocode of the synthetic benchmarks we used for performance analysis: *PFor* and *RecPFor*. In the PFor benchmark, a simple parallel loop is repeated multiple times. Each parallel loop is implemented as a recursive binary fork-join pattern (as in `cilk_for` in Cilk Plus [48]). In the RecPFor benchmark, parallel tasks are recursively created as a binary tree, and, at each recursion, parallel loops are repeatedly executed. This computation pattern appears in many algorithms (e.g., quicksort and decision tree construction [49]) in which an input array is first manipulated by parallel loops and then divided into two sub-arrays, and the same procedure is applied to each sub-array recursively. The RecPFor benchmark mimics this behavior.

These benchmarks are parameterized by the number of consecutive parallel loops in the `PFor()` function (K), the execution time of each leaf task (`compute()` function)



(a) PFor on ITO-A (b) RecPFor on ITO-A (c) PFor on WISTERIA-O (d) RecPFor on WISTERIA-O

Fig. 6. Performance comparison between different joining strategies and steal strategies with synthetic benchmarks (PFor and RecPFor) on ITO-A with 576 cores (16 nodes) and on WISTERIA-O with 1728 cores (36 nodes). X-axis represents ideal execution time calculated by T_1/P (see Section IV-C); y-axis represents parallel efficiency (ratio of ideal execution time to measured execution time).

(M), and the problem size given to the root task (N). The `compute()` function executes a fixed number of SIMD fused multiply-add (FMA) operations. The number of FMA operations was configured to run for a specified duration (M). The total work of the PFor benchmark is $T_1 = KMN$, and that of the RecPFor benchmark is $T_1 = KMN \log_2 N + MN$. For evaluation, we fixed $K = 5$ and $M = 10\mu$s for both benchmarks and varied N to change the problem size.

V. Performance Analysis and Evaluation

A. *Performance Analysis of Joining Strategies*

To evaluate the join techniques introduced in Section III, we implemented a version with only local collection enabled and a version with both local collection and greedy join enabled. The baseline is the original implementation of MassiveThreads/DM with a few trivial performance issues fixed.

The parallel efficiencies of the synthetic benchmarks with various problem sizes (parameter N) are plotted in Fig. 6. The ideal execution time is simply T_1/P, where T_1 is the theoretical work discussed in Section IV-C, and parallel efficiency is the ratio of the ideal execution time to the actual (measured) execution time. In PFor on ITO-A, local collection achieved up to 40% improvement over the baseline. Greedy join did not contribute to the performance improvement, whereas in RecPFor it did. This is because RecPFor has a more complicated synchronization pattern than PFor; the PFor benchmark has little work after a join at each recursion, while the RecPFor benchmark has substantial work after each join as parallel loops are repeated at each recursion. As a result, in RecPFor on ITO-A, local collection improved performance by 27% over the baseline, and greedy join improved performance by an additional 8%. In total, our join techniques achieved up to 37% performance improvement over the baseline in RecPFor on ITO-A. We observed similar results for WISTERIA-O.

B. *Child Stealing vs. Continuation Stealing*

Fig. 6 also shows the performance of child stealing with Full and RtC threads. In PFor, the performance difference between child and continuation stealing is small, whereas in RecPFor, continuation stealing consistently outperformed child stealing. Notably, continuation stealing was at most $1.3\times$ faster than child stealing (Full) and at most $4.8\times$ faster than child stealing (RtC) in RecPFor on WISTERIA-O. Note that Full threads incur larger overheads on WISTERIA-O because of their relatively large context switching costs and slower processor speed.

TABLE II shows statistics for outstanding joins and steal events, which were profiled using the largest problem sizes in Fig. 6. The numbers of outstanding joins and successful steals show that, in child stealing, a majority of successful steals resulted in outstanding joins. The numbers for child stealing (RtC) are exceptionally large because of the buried join problem explained in Section IV-B. Continuation stealing incurred by an order of magnitude fewer outstanding joins than successful steals. This supports our assertion in Section II-B that continuation stealing causes fewer outstanding joins.

Here, we introduce a metric, *outstanding join time*, which is the duration since an outstanding join's continuation became resumable (i.e., both the joining and joined thread reach the synchronization point) and until it is actually resumed. This metric well characterizes the join implementations as all three implementations based on stalling join have long average outstanding join times, whereas continuation stealing (greedy) has the shortest outstanding join time. Note that a longer outstanding join time does not necessarily mean that the scheduling is bad; for example, as long as all of the workers are busy executing tasks, a long outstanding join time is not a problem. However, it becomes problematic when workers are idle, because idle workers should execute ready tasks (outstanding joins) but do not execute them, which is attributable to the scheduler [50].

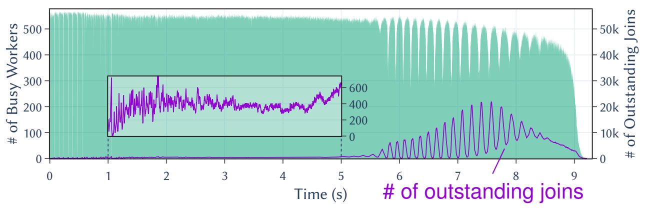
To confirm that outstanding joins are related to the low efficiency of child stealing, we plotted time series data for RecPFor in Fig. 7. The filled (green) area shows how many workers were busy executing tasks, and the line (purple) plot represents the number of outstanding joins that were runnable at that time. With continuation stealing, almost all workers were busy all the time, and there were almost no outstanding joins. In contrast, with child stealing (Full), the

TABLE II
STATISTICS OF JOIN AND STEAL EVENTS IN CHILD STEALING AND CONTINUATION STEALING ON ITO-A WITH 576 CORES (16 NODES) AND ON WISTERIA-O WITH 1728 CORES (36 NODES). THE PROBLEM SIZES CORRESPOND TO THE LARGEST SIZES IN FIG. 6.

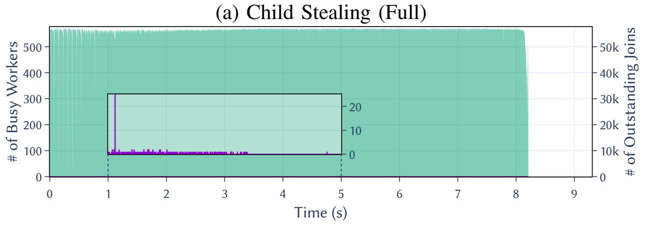
System	Bench	Steal Strategy	Execution Time	# of Outstanding Joins	Avg. Outstanding Join Time	# of Steals (Successful)	Avg. Steal Latency (Successful)	# of Steals (Failed)	Avg. Stolen Task Size	Avg. Task Copy Time
ITO-A	PFor	Cont. Steal (greedy)	2.98 s	8672	14.1 μs	72831	28.8 μs	1108412	1789 bytes	5.85 μs
		Cont. Steal (stalling)	2.98 s	6832	13965.4 μs	74091	28.9 μs	1089578	1687 bytes	6.02 μs
		Child Steal (Full)	3.07 s	45920	1016.3 μs	76583	27.7 μs	4936210	56 bytes	3.76 μs
		Child Steal (RtC)	3.01 s	61005	17096.3 μs	75777	27.1 μs	2661418	56 bytes	3.83 μs
	RecPFor	Cont. Steal (greedy)	8.30 s	56876	15.0 μs	474991	31.6 μs	2853760	1845 bytes	5.72 μs
		Cont. Steal (stalling)	8.33 s	72546	8128.6 μs	807097	30.4 μs	2561299	1790 bytes	5.68 μs
		Child Steal (Full)	9.31 s	3208417	8237.9 μs	6038858	29.3 μs	16656796	55 bytes	4.18 μs
		Child Steal (RtC)	48.64 s	40797378	297.6 μs	45092301	59.5 μs	1471927098	55 bytes	8.53 μs
WISTERIA-O	PFor	Cont. Steal (greedy)	2.30 s	25425	6.6 μs	206857	20.7 μs	1602730	1289 bytes	3.47 μs
		Cont. Steal (stalling)	2.30 s	19093	7006.5 μs	210601	20.9 μs	1583784	1286 bytes	3.50 μs
		Child Steal (Full)	2.57 s	156116	165.2 μs	221887	19.8 μs	1341056	56 bytes	2.91 μs
		Child Steal (RtC)	2.37 s	175845	8887.9 μs	218420	18.4 μs	1571870	56 bytes	2.78 μs
	RecPFor	Cont. Steal (greedy)	5.94 s	229154	6.5 μs	1790018	20.4 μs	3186116	1139 bytes	3.37 μs
		Cont. Steal (stalling)	5.97 s	279035	4852.7 μs	3086034	20.6 μs	4349765	1156 bytes	3.42 μs
		Child Steal (Full)	7.69 s	8602558	6309.9 μs	15704498	19.9 μs	62509110	55 bytes	2.90 μs
		Child Steal (RtC)	27.71 s	205358008	88.9 μs	223562173	19.2 μs	8164463070	56 bytes	2.98 μs

number of busy workers has many "valleys" in the latter half of the execution. Simultaneously, there were a large number of outstanding joins, which means that runnable tasks were not being executed despite some workers being idle because of a nongreedy schedule.

Finally, we analyze steal latency. TABLE II shows that continuation stealing had two orders of magnitude larger "stolen task sizes" than child stealing because of stack migration. Therefore, continuation stealing incurs a longer "task copy time," but the times are of the same order of magnitude. Note that child stealing (RtC) in RecpPFor on ITO-A incurred much longer latency because of the greater number of steal attempts, whereas latency on WISTERIA-O was less affected. Overall, we found that the increased average steal latency (successful) with continuation stealing was less than 20% compared with that for child stealing for all cases in Fig. 6. Summarizing, despite a small increase in steal latency, continuation stealing can outperform child stealing by efficiently resolving joins.



(a) Child Stealing (Full)



(b) Continuation Stealing (greedy join)

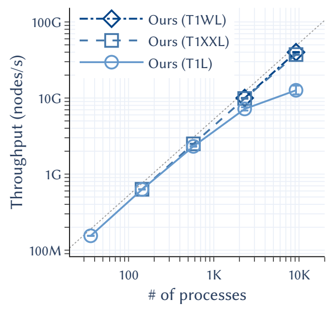
Fig. 7. Time series of scheduler activities in RecPFor benchmark ($N = 2^{22}$) on ITO-A with 576 cores (16 nodes). Filled (green) area represents number of busy workers (left y-axis); line (purple) plot represents number of ready-to-execute outstanding joins (right y-axis). Numbers of outstanding joins between 1 s and 5 s are zoomed in and plotted as an overlay on the main chart.

C. Unbalanced Tree Search (UTS)

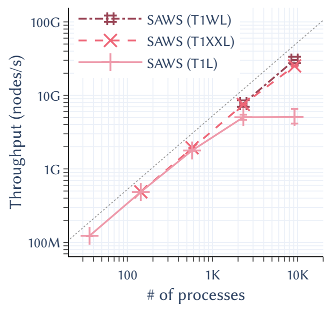
The UTS benchmark [18] has been used for measuring the load balancing capability of parallel runtime systems in many studies [11], [12], [13], [14], [51], [52], [53], [21], [22], [16], [17]. Most existing implementations on distributed memory are based on a BoT, in which task dependency cannot be described. We found that the scalability of our runtime system was comparable to or even better than that of existing state-of-the-art UTS implementations.

The task of UTS is to count the total number of nodes in an unbalanced tree, which is generated on the fly using a random hash (SHA-1). The identical tree is generated deterministically if the same seed and parameters are given. A natural way to count the number of nodes would be depth-first tree traversal, and the recursive fork-join constructs in MassiveThreads/DM can straightforwardly parallelize the tree traversal.

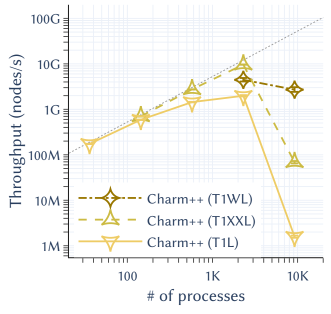
We compared the performance of MassiveThreads/DM with three existing distributed work-stealing implementations: SAWS [12], Charm++ [19], [20], and X10/GLB [21], [22]. SAWS is a state-of-the-art RDMA-based work-stealing library (see Section VI). We used the author-provided implementation of SAWS and UTS [54] with the OpenSHMEM implementation build together with Open MPI in our environment (TABLE I). We compiled Charm++ v7.0.0 with the MPI backend and the ParSSSE [53] implementation of UTS included in the official repository, with work stealing enabled (`-module workstealing` option). We built X10 [55] with the MPI backend and the GLB-UTS benchmark [56] using the compiler with the C++ backend (`x10c++`). We modified all of their UTS implementations to run multiple times to execute



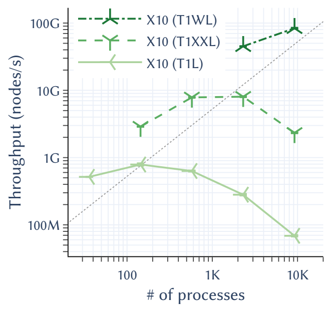
(a) MassiveThreads/DM (ours)



(b) SAWS



(c) Charm++



(d) X10/GLB

Fig. 8. Throughput of UTS benchmark on ITO-A (up to 9212 cores).

the same number of warm-up runs. The implementations were based on a BoT[2], which relies on global termination detection and collective communication to gather the node counts at the end of execution.

For evaluation, we used three sizes of the geometric tree (T1L < T1XXL < T1WL). The primary performance metric was throughput, which was calculated in terms of the number of nodes visited per second (nodes/s). The execution time of the serial depth-first search with T1L was 19.4 s (5.27 Mnodes/s) on ITO-A and 65.9 s (1.55 Mnodes/s) on WISTERIA-O. Note that MassiveThreads/DM adds serial overheads of only 18% on ITO-A and 27% on WISTERIA-O to the serial depth-first execution.

Fig. 8 shows the scalability of the four UTS implementations on ITO-A. The dotted straight line represents the ideal throughput calculated using the throughput of a serial depth-first search. Although the throughput of X10/GLB exceeded the ideal because the UTS implementation of X10/GLB is highly optimized from the original, we do not delve deeper into its serial performance because scalability is our primary interest. We can see that Charm++ did not scale to large core counts and that X10/GLB failed to exhibit strong scaling for small trees. We consider that these performance problems were due to the message-based (two-sided) work-stealing implementation in Charm++ and X10/GLB. In contrast, the RDMA-based (one-sided) implementations, MassiveThreads/DM and SAWS, exhibited good scalability even for relatively small trees. Note that the UTS implementation of SAWS was based on a BoT, which is a more restrictive form of parallelism than the nested fork-join in MassiveThreads/DM. These results indicate that continuation stealing (thread migration) and task dependencies (join) are not performance-limiting factors in implementing distributed task-parallel runtime systems.

Fig. 9 shows the scalability of MassiveThreads/DM on WISTERIA-O, for which the trend is similar to that on ITO-A. We did not run other runtime systems on WISTERIA-O because of porting issues. Notably, for T1WL on 110,592 cores, it achieved 96.4% parallel efficiency calculated with a single-core execution time.

[2]Although the UTS implementation of Charm++ is based on a BoT, Charm++ itself has more synchronization capability than a BoT.

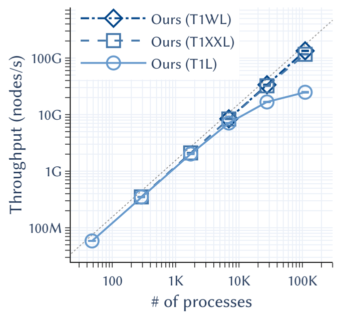


Fig. 9. Throughput of UTS with MassiveThreads/DM on WISTERIA-O.

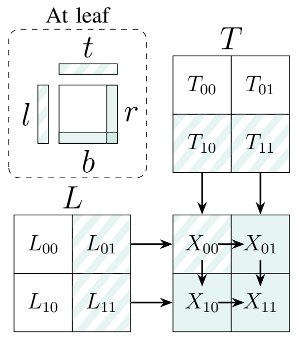


Fig. 10. Task dependency of LCS.

D. Longest Common Subsequence (LCS)

To examine scheduler performance when *futures* were intensively used, we developed the LCS benchmark. The LCS problem for two sequences $A = \langle a_1, ..., a_n \rangle$ and $B = \langle b_1, ..., b_n \rangle$ is defined by the following recurrence:

$$X(i,j) = \begin{cases} 0 & (i = 0 \lor j = 0) \\ X(i-1, j-1) & (i,j > 0 \land a_i = b_j) \\ \max\{X(i, j-1), X(i-1, j)\} & (i,j > 0 \land a_i \neq b_j) \end{cases}$$

$X(n,n)$ is the length of the LCS for A and B. Although a naive algorithm would require $O(n^2)$ space for computing the LCS, linear space algorithms with $O(n)$ space are known [57], [58]. In this evaluation, following the problem setting in [59], [60], we solved only for the length of an LCS. LCS computation can be naturally parallelized by divide-and-conquer and has good cache locality, but a naive divide-and-conquer approach lengthens the critical path from $O(n)$ to $O(n^{\log_2 3})$, which severely degrades parallelism [57], [58]. This is because strict fork-join programs add artificial dependencies to the original wavefront dependency pattern. Although our approach is also based on recursive decomposition of 2D space similar to the naive divide-and-conquer approach, the use of *futures* resolves the critical path length problem.

Fig. 10 illustrates the recursive 2D decomposition and task dependencies between the blocks; the notations correspond to those used in the algorithm in Fig. 11, which is based on the sequential algorithm by Chowdhury and Ramachandran [58]. Futures are dynamically created at each recursion and form

```
54 Function LCS($i, j, T, L, n$)
55     if $n \le C$ then
56         $(t, \_) \leftarrow T.join()$,   $(\_, l) \leftarrow L.join()$
57         $(b, r) \leftarrow$ LCS_SEQ($i, j, t, l, n$)
58         return $(b, r)$
59     else
60         $(\_, T_{10}, T_{11}) \leftarrow T.join()$,   $(L_{01}, \_, L_{11}) \leftarrow L.join()$
61         $X_{00} \leftarrow$ spawn LCS($i$        , $j$        , $T_{10}$ , $L_{01}$ , $n/2$)
62         $X_{01} \leftarrow$ spawn LCS($i$        , $j + n/2$, $T_{11}$ , $X_{00}$ , $n/2$)
63         $X_{10} \leftarrow$ spawn LCS($i + n/2$, $j$        , $X_{00}$ , $L_{11}$ , $n/2$)
64         $X_{11} \leftarrow$ spawn LCS($i + n/2$, $j + n/2$, $X_{01}$ , $X_{10}$ , $n/2$)
65         $X_{00}.join()$
66         return $(X_{01}, X_{10}, X_{11})$
```

Fig. 11. Algorithm for LCS based on recursive decomposition and futures.

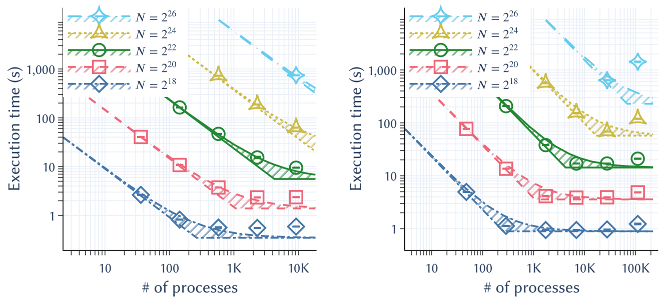
TABLE III
EXECUTION TIMES OF LCS ON ITO-A WITH 576 CORES (16 NODES).

Size	Cont. Steal (greedy)	Cont. Steal (stalling)	Child Stealing (Full)
2^{18}	0.569 s	3.44 s	93.1 s
2^{22}	45.9 s	433 s	2.11×10^4 s

a tree-like structure in which each node is a future. Each intermediate node has three child futures (X_{01}, X_{10}, X_{11}), and each leaf has the values at the boundaries (b, r) to be passed to the successive computations. The LCS function receives two futures, T and L, which are geometrically at the top and left of the current block. At an intermediate level, we first join them (line 60) and the returned child futures (T_{10}, T_{11}, L_{01}, L_{11}) are then passed to the child computations (lines 61–64), following the dependency pattern illustrated in Fig. 10. X_{00} is joined at line 65 to avoid creating an excessive number of futures. When problem size n is sufficiently small ($n < C$), the algorithm sequentially computes the values at the output boundaries (b and r) using the values at the input boundaries (t and l), which are acquired by joining the futures (T and L).

We experimentally implemented futures that allow for multiple consumers in our runtime system. A limitation is that we must set a fixed number of consumers when spawning a future, but it suffices for LCS because the number of consumers is known in advance. When a future is completed and it finds multiple futures waiting for its completion, it immediately resumes one of them and pushes the others into the local task queue. For evaluation, we set $C = 512$ and used random input sequences of 1-byte characters.

TABLE III compares the performance of the LCS benchmark with different scheduling policies on ITO-A with 16 nodes. Continuation stealing with greedy join achieved the best performance; it was an order of magnitude faster than continuation stealing with stalling join and two orders of magnitude faster than child stealing. Stalling join is much slower than greedy join because the lack of thread migration at joins led to a larger load imbalance. In particular, execution delay of futures near the root was the bottleneck. In child stealing, we found that almost all tasks were executed by the main worker because tied tasks were never migrated[3].



(a) ITO-A (b) WISTERIA-O

Fig. 12. Execution times of LCS with continuation stealing (greedy join). The filled area enclosed by lines represent $\max(T_1/P, T_\infty) \le T_P \le T_1/P + T_\infty$ (lower bound and upper bound of the ideal, greedy-scheduling theorem) for each problem size.

Next, we validate the performance of continuation stealing (greedy join) by comparing it against the theoretical ideal performance. Fig. 12 shows the experimental results. The filled area for each N is enclosed by the lines representing the lower bound $T_P \ge \max(T_1/P, T_\infty)$ and the upper bound of the greedy-scheduling theorem $T_P \le T_1/P + T_\infty$ (see Section II-A). Note that the greedy-scheduling theorem assumes an ideal greedy scheduler with no run-time overhead; the performance of real-world schedulers can be out of this bound. Suppose that the execution time for each leaf block is T_c, the work and span can be calculated as $T_1 = (N/C)^2 T_c$ and $T_\infty = (2N/C - 1)T_c$. The value of T_c was 0.340 ms for ITO-A and 0.872 ms for WISTERIA-O, which was measured by serial execution for $N = 2^{16}$. The results show that most of the parallel execution times were within these bounds, which suggests that almost no tasks were unnecessarily blocked by the scheduler of continuation stealing (greedy join). Despite a few exceptions, continuation stealing successfully scaled to 9,126 cores on ITO-A and 27,648 cores on WISTERIA-O. We could not see any performance improvements on WISTERIA-O with 110,592 cores, which needs further performance investigation. More sophisticated scheduling policies for futures [61] might help to further improve performance, but this is beyond the scope of this paper.

VI. Related Work

Child stealing on distributed memory has been intensively studied. Dinan et al. [11] implemented RDMA-based work stealing in the Scioto infrastructure [31] with an efficient implementation of the steal-half queue [62]. Structured Atomic Work Stealing (SAWS) [12] further improved the queue implementation by utilizing RDMA atomic operations. The global load balancing (GLB) library [21], [22] was developed for the X10 language using lifeline-based work stealing [14]. The above systems are all based on a BoT, in which task dependencies cannot be described.

[3]Load balancing did not work in child stealing. Most tasks were popped from the queue and suspended immediately before being stolen, because steal operations are much slower than local pop and suspend operations.

Satin [32] and HotSLAW [13] support fork-join parallelism with RtC threads, so they are subject to the "buried join" problem [25] explained in Section IV-B. Grappa [34] also uses RtC threads, but it mitigates the buried join problem by creating many user-level *workers* (rather than threads) as Full threads to oversubscribe cores. This approach is essentially similar to that of our child stealing implementation with Full threads tied to each core. KAAPI [33] uses an approach similar to that of Grappa by using kernel-level threads. These implementations are tied-tasks; i.e., once a task begins, it cannot be migrated to other nodes.

Compared with child stealing, continuation stealing are rarely implemented on distributed memory. Distributed Cilk [63] and Tascell [64] support continuation stealing but they require special compiler support. Charm++ [19], [20] and Adaptive MPI [36] support thread migration on distributed memory based on the iso-address scheme [35], but they are not designed for RDMA. MassiveThreads/DM (uni-address threads) [16], [17] is the only library that supports RDMA-based continuation stealing (as explained in Section II-D).

Many researchers have investigated topology-aware work stealing to reduce the number of remote work-stealing requests [15], [51], [52], [65], [66], [67], but they basically assumed two-sided communication. Although these approaches could be used in conjugation with RDMA-based work stealing, their benefits have not been well studied in the context of RDMA, which is our future interest.

Performance comparison between child and continuation stealing on shared memory has also been conducted [68], [69], [27], and the tradeoffs between them have been reported. They found that, roughly speaking, child stealing is preferable for shallow parallel loops in which a parent task creates many child tasks, whereas continuation stealing is preferable for deeply nested parallelism, which is consistent with our results.

VII. Conclusion

We have demonstrated that distributed continuation stealing can scale to as many as 110,592 cores. In-depth analysis revealed that, even though continuation stealing incurred a slightly longer steal latency (less than 20% overhead), it achieved better overall performance than child stealing in several benchmarks by resolving task join primitives more efficiently. Particularly when the synchronization pattern is complicated (e.g., futures in LCS), the performance penalty due to the lack of thread migration was substantial.

To focus on the scheduler performance, we did not include applications that use global memory in this paper; data are only exchanged via arguments or return values of tasks. Nevertheless, global heaps are essential for writing more practical applications. Efficient support for global heaps, such as Partitioned Global Address Space (PGAS) or Distributed Shared Memory (DSM), remains for future work.

Our distributed continuation-stealing runtime system is implemented as a C++ library that can be compiled with ordinary C++ compilers and MPI-3 RMA. The source code and scripts for experiments in this paper are publicly available at https://github.com/s417-lama/cluster22-contsteal-artifact.

Acknowledgment

This work was supported by JSPS KAKENHI Grant Number 21J22305. This paper is based on results obtained from a project, JPNP16007, commissioned by the New Energy and Industrial Technology Development Organization (NEDO). The computation was carried out using the computer resource offered under the category of General Projects by Research Institute for Information Technology, Kyushu University, and the FUJITSU Supercomputer PRIMEHPC FX1000 (Wisteria/BDEC-01 Odyssey) at the Information Technology Center, The University of Tokyo.

References

[1] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An efficient multithreaded runtime system," *Journal of Parallel and Distributed Computing*, vol. 37, no. 1, pp. 55–69, Aug. 1996.

[2] 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*, ser. PLDI '98, 1998, pp. 212–223.

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

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

[5] 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, Mar. 2011.

[6] K. B. Wheeler, R. C. Murphy, 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*, ser. IPDPS '08, 2008.

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

[8] 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, Oct. 2017.

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

[10] 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*, ser. SPAA '00, July 2000, pp. 1–12.

[11] J. Dinan, D. B. Larkins, P. Sadayappan, S. Krishnamoorthy, and J. Nieplocha, "Scalable work stealing," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, ser. SC '09, 2009, pp. 53:1–53:11.

[12] H. Cartier, J. Dinan, and D. B. Larkins, "Optimizing work stealing communication with structured atomic operations," in *Proceedings of the 50th International Conference on Parallel Processing*, ser. ICPP '21, 2021, pp. 36:1–36:10.

[13] 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*, ser. PGAS '11, 2011, pp. 1–10.

[14] V. A. Saraswat, P. Kambadur, S. Kodali, D. Grove, and S. Krishnamoorthy, "Lifeline-based global load balancing," in *Proceedings of the 16th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '11, 2011, pp. 201–212.

[15] 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*, ser. ICPP '13, 2013, pp. 100–109.

[16] S. Akiyama and K. Taura, "Uni-address threads: Scalable thread management for RDMA-based work stealing," in *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '15, 2015, pp. 15–26.

[17] ——, "Scalable work stealing of native threads on an x86-64 InfiniBand cluster," *Journal of Information Processing*, vol. 24, no. 3, pp. 583–596, May 2016.

[18] S. Olivier, J. Huan, J. Liu, J. Prins, J. Dinan, P. Sadayappan, and C.-W. Tseng, "UTS: An unbalanced tree search benchmark," in *Proceedings of the 19th International Conference on Languages and Compilers for Parallel Computing*, ser. LCPC '06, 2006, pp. 235–250.

[19] L. Kale, B. Acun, S. Bak, A. Becker, M. Bhandarkar, N. Bhat, A. Bhatele, E. Bohm, C. Bordage, R. Brunner, R. Buch, S. Chakravorty, K. Chandrasekar, J. Choi, M. Denardo, J. DeSouza, M. Diener, H. Dokania, I. Dooley, W. Fenton, J. Galvez, F. Gioachin, A. Gupta, G. Gupta, M. Gupta, A. Gursoy, V. Harsh, F. Hu, C. Huang, N. Jagathesan, N. Jain, P. Jetley, P. Jindal, R. Kanakagiri, G. Koenig, S. Krishnan, S. Kumar, D. Kunzman, M. Lang, A. Langer, O. Lawlor, C. Wai Lee, J. Lifflander, K. Mahesh, C. Mendes, H. Menon, C. Mei, E. Meneses, E. Mikida, P. Miller, R. Mokos, V. Narayanan, X. Ni, K. Nomura, S. Paranjpye, P. Ramachandran, B. Ramkumar, E. Ramos, M. Robson, N. Saboo, V. Saletore, O. Sarood, K. Senthil, N. Shah, W. Shu, A. B. Sinha, Y. Sun, Z. Sura, E. Totoni, K. Varadarajan, R. Venkataraman, J. Wang, L. Wesolowski, S. White, T. Wilmarth, J. Wright, J. Yelon, and G. Zheng, "The Charm++ Parallel Programming System," Aug 2019. [Online]. Available: https://charm.cs.illinois.edu

[20] L. V. Kale and G. Zheng, "Chapter 1: The Charm++ Programming Model," in *Parallel Science and Engineering Applications: The Charm++ Approach*, 1st ed., L. V. Kale and A. Bhatele, Eds. Boca Raton, FL, USA: CRC Press, Inc., 2013, ch. 1, pp. 1–16.

[21] W. Zhang, O. Tardieu, D. Grove, B. Herta, T. Kamada, V. Saraswat, and M. Takeuchi, "GLB: Lifeline-based global load balancing library in X10," in *Proceedings of the First Workshop on Parallel Programming for Analytics Applications*, ser. PPAA '14, 2014, pp. 31–40.

[22] O. Tardieu, B. Herta, D. Cunningham, D. Grove, P. Kambadur, V. Saraswat, A. Shinnar, M. Takeuchi, and M. Vaziri, "X10 and APGAS at petascale," in *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '14, pp. 53–66.

[23] D. B. Wagner and B. G. Calder, "Leapfrogging: A portable technique for implementing efficient futures," in *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPOPP '93, 1993, pp. 208–217.

[24] D. Lea, "A Java fork/join framework," in *Proceedings of the ACM 2000 conference on Java Grande*, ser. JAVA '00, 2000, pp. 36–43.

[25] K.-F. Faxén, "Efficient work stealing for fine grained parallelism," in *Proceedings of the 39th International Conference on Parallel Processing*, ser. ICPP '10, 2010, pp. 313–322.

[26] E. Mohr, D. A. Kranz, and R. H. Halstead, "Lazy task creation: A technique for increasing the granularity of parallel programs," in *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, ser. LFP '90, 1990, pp. 185–197.

[27] S. Iwasaki, A. Amer, K. Taura, and P. Balaji, "Analyzing the performance trade-off in implementing user-level threads," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 8, pp. 1859–1877, Feb 2020.

[28] R. P. Brent, "The parallel evaluation of general arithmetic expressions," *Journal of the ACM*, vol. 21, no. 2, pp. 201–206, Apr. 1974.

[29] 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*, ser. SC '18, 2018, pp. 23:1–23:12.

[30] 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*, ser. PPoPP '21, 2021.

[31] J. Dinan, S. Krishnamoorthy, D. B. Larkins, J. Nieplocha, and P. Sadayappan, "Scioto: A framework for global-view task parallelism," in *Proceedings of the 37th International Conference on Parallel Processing*, ser. ICPP '08, 2008, pp. 586–593.

[32] R. V. Van Nieuwpoort, G. Wrzesińska, C. J. H. Jacobs, and H. E. Bal, "Satin: A high-level and efficient grid programming model," *ACM Transactions on Programming Languages and Systems*, vol. 32, no. 3, pp. 9:1–9:39, Mar. 2010.

[33] T. Gautier, X. Besseron, and L. Pigeon, "KAAPI: A thread scheduling runtime system for data flow computations on cluster of multi-processors," in *Proceedings of the 2007 International Workshop on Parallel Symbolic Computation*, ser. PASCO '07, 2007, pp. 15–23.

[34] J. Nelson, B. Holt, B. Myers, P. Briggs, L. Ceze, S. Kahan, and M. Oskin, "Latency-tolerant software distributed shared memory," in *Proceedings of the 2015 USENIX Annual Technical Conference*, ser. USENIX ATC '15, 2015, pp. 291–305.

[35] G. Antoniu, L. Bougé, and R. Namyst, "An efficient and transparent thread migration scheme in the PM2 runtime system," in *Proceedings of the 11th IPPS/SPDP'99 Workshops Held in Conjunction with the 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing*, ser. RTSPP '99, 1999, pp. 496–510.

[36] C. Huang, O. Lawlor, and L. V. Kalé, "Adaptive MPI," in *Proceedings of the 16th International Conference on Languages and Compilers for Parallel Computing*, ser. LCPC '03, 2003, pp. 306–322.

[37] A. Robison, "A primer on scheduling fork-join parallelism with work stealing," ISO/IEC JTC1/SC22/WG21 - The C++ Standards Committee, Tech. Rep. N3872, 2014.

[38] 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, March 1999.

[39] F. Schmaus, N. Pfeiffer, W. Schröder-Preikschat, T. Hönig, and J. Nolte, "Nowa: A wait-free continuation-stealing concurrency platform," in *Proceedings of the 36th IEEE International Parallel and Distributed Processing Symposium*, ser. IPDPS '21, 2021.

[40] G. E. Blelloch and M. Reid-Miller, "Pipelining with futures," *Theory of Computing Systems*, vol. 32, no. 3, pp. 213–239, June 1999.

[41] D. Spoonhower, G. E. Blelloch, P. B. Gibbons, and R. Harper, "Beyond nested parallelism: Tight bounds on work-stealing overheads for parallel futures," in *Proceedings of the Twenty-First Annual Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '09, 2009, pp. 91–100.

[42] T. Hoefler, J. Dinan, R. Thakur, B. Barrett, P. Balaji, W. Gropp, and K. Underwood, "Remote memory access programming in MPI-3," *ACM Transactions on Parallel Computing*, vol. 2, no. 2, pp. 1–26, July 2015.

[43] K. University, "Introduction of ITO," 2017. [Online]. Available: https://www.cc.kyushu-u.ac.jp/scp/eng/system/ITO/01_intro.html

[44] T. U. of Tokyo, "Introduction to the Wisteria/BDEC-01," 2021. [Online]. Available: https://www.cc.u-tokyo.ac.jp/en/supercomputer/wisteria/system.php

[45] M. Sato, Y. Ishikawa, H. Tomita, Y. Kodama, T. Odajima, M. Tsuji, H. Yashiro, M. Aoki, N. Shida, I. Miyoshi, K. Hirai, A. Furuya, A. Asato, K. Morita, and T. Shimizu, "Co-design for A64FX manycore processor and "Fugaku"," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '20, 2020, pp. 47:1–47:15.

[46] P. Shamis, M. G. Venkata, M. G. Lopez, M. B. Baker, O. Hernandez, Y. Itigin, M. Dubman, G. Shainer, R. L. Graham, L. Liss, Y. Shahar, S. Potluri, D. Rossetti, D. Becker, D. Poole, C. Lamb, S. Kumar, C. Stunkel, G. Bosilca, and A. Bouteiller, "UCX: An open source framework for HPC network APIs and beyond," in *Proceedings of the 2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*, ser. HOTI '15, 2015, pp. 40–43.

[47] J. Schuchart, A. Bouteiller, and G. Bosilca, "Using MPI-3 RMA for active messages," in *Proceedings of the 2019 IEEE/ACM Workshop on Exascale MPI*, ser. ExaMPI '19, 2019, pp. 47–56.

[48] C. E. Leiserson, "The Cilk++ concurrency platform," *The Journal of Supercomputing*, vol. 51, no. 3, pp. 244–257, Mar. 2010.

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

[50] A. Huynh and K. Taura, "Delay Spotter: A tool for spotting scheduler-caused delays in task parallel runtime systems," in *Proceedings of the 2017 IEEE International Conference on Cluster Computing*, ser. CLUSTER '17, 2017, pp. 114–125.

[51] S. Perarnau and M. Sato, "Victim selection and distributed work stealing performance: A case study," in *Proceedings of the 28th IEEE International Parallel and Distributed Processing Symposium*, ser. IPDPS '14, 2014, pp. 659–668.

[52] H. Parikh, V. Deodhar, A. Gavrilovska, and S. Pande, “Distributed work stealing at scale via matchmaking,” in *Proceedings of the 2021 IEEE International Conference on Cluster Computing*, ser. CLUSTER ’21, 2021, pp. 250–260.

[53] Y. Sun, G. Zheng, P. Jetley, and L. V. Kalé, “ParSSSE: An adaptive parallel state space search engine,” *Parallel Processing Letters*, vol. 21, no. 03, pp. 319–338, 2011.

[54] D. B. Larkins, “saws (GitHub repository),” Oct. 2021, commit hash: `cdd36782c971`. [Online]. Available: https://github.com/brianlarkins/saws

[55] IBM Corporation, “X10 (GitHub repository),” Feb. 2020, commit hash: `5412ae0a0db1`. [Online]. Available: https://github.com/x10-lang/x10

[56] ——, “X10 benchmarks (GitHub repository),” Feb. 2020, commit hash: `2d6a59614ad9`. [Online]. Available: https://github.com/x10-lang/x10-benchmarks

[57] D. S. Hirschberg, “A linear space algorithm for computing maximal common subsequences,” *Commun. ACM*, vol. 18, no. 6, pp. 341–343, June 1975.

[58] R. A. Chowdhury and V. Ramachandran, “Cache-oblivious dynamic programming,” in *Proceedings of the Seventeenth Annual ACM-SIAM Symposium on Discrete Algorithm*, ser. SODA ’06, 2006, pp. 591–600.

[59] Y. Tang, R. You, H. Kan, J. J. Tithi, P. Ganapathi, and R. A. Chowdhury, “Cache-oblivious wavefront: Improving parallelism of recursive dynamic programming algorithms without losing cache-efficiency,” in *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP ’15, 2015, pp. 205–214.

[60] R. Chowdhury, P. Ganapathi, Y. Tang, and J. J. Tithi, “Provably efficient scheduling of cache-oblivious wavefront algorithms,” in *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA ’17, 2017, pp. 339–350.

[61] K. Singer, Y. Xu, and I.-T. A. Lee, “Proactive work stealing for futures,” in *Proceedings of the 24th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP ’19, 2019, pp. 257–271.

[62] D. Hendler and N. Shavit, “Non-blocking steal-half work queues,” in *Proceedings of the Twenty-First Annual Symposium on Principles of Distributed Computing*, ser. PODC ’02, 2002, pp. 280–289.

[63] K. H. Randall, “Cilk: Efficient multithreaded computing,” Ph.D. dissertation, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, May 1998.

[64] T. Hiraishi, M. Yasugi, S. Umatani, and T. Yuasa, “Backtracking-based load balancing,” in *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP ’09, 2009, pp. 55–64.

[65] T.-T. Vu and B. Derbel, “Link-heterogeneous work stealing,” in *Proceedings of the 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, ser. CCGrid ’14, 2014, pp. 354–363.

[66] R. Nakashima, H. Yoritaka, M. Yasugi, T. Hiraishi, and S. Umatani, “Extending a work-stealing framework with priorities and weights,” in *Proceedings of the 2019 IEEE/ACM 9th Workshop on Irregular Applications: Architectures and Algorithms*, ser. IA3 ’19, 2019, pp. 9–16.

[67] J.-N. Quintin and F. Wagner, “Hierarchical work-stealing,” in *Proceedings of the 16th European Conference on Parallel Processing*, ser. Euro-Par ’10, 2010, pp. 217–229.

[68] Y. Guo, R. Barik, R. Raman, and V. Sarkar, “Work-first and help-first scheduling policies for async-finish task parallelism,” in *Proceedings of the 23rd IEEE International Symposium on Parallel and Distributed Processing Symposium*, ser. IPDPS ’09, 2009.

[69] 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 Symposium on Parallel and Distributed Processing Symposium*, ser. IPDPS ’10, 2010.