Electronic Preprint for Journal of Information Processing Vol.24 No.3

Regular Paper

Scalable Work Stealing of Native Threads on an x86-64 Infiniband Cluster

Shigeki Akiyama[1,a) Kenjiro Taura[1,b)

Received: July 31, 2015, Accepted: December 8, 2015

Abstract: Task parallelism on large-scale distributed memory environments is still a challenging problem. The focuses of our work are flexibility of task model and scalability of inter-node load balancing. General task models provide functionalities for suspending and resuming tasks at any program point, and such a model enables us flexible task scheduling to achieve higher processor utilization, locality-aware task placement, etc. To realize such a task model, we have to employ a thread—an execution context containing register values and stack frames—as a representation of a task, and implement thread migration for inter-node load balancing. However, an existing thread migration scheme, *iso-address*, has a scalability limitation: it requires virtual memory proportional to the number of processors in each node. In large-scale distributed memory environments, this results in a huge virtual memory usage beyond the virtual address space limit of current 64 bit CPUs. Furthermore, this huge virtual memory consumption makes it impossible to implement one-sided work stealing with Remote Direct Memory Access (RDMA) operations. One-sided work stealing is a popular approach to achieving high efficiency of load balancing; therefore this also limits scalability of distributed memory task parallelism. In prior work, we propose *uni-address*, a new thread migration scheme which significantly reduces virtual memory usage for thread stacks and enables RDMA-based work stealing, and implements a lightweight multithread library supporting RDMA-based work stealing on top of Fujitsu FX10 system. In this paper, we port the library to an x86-64 Infiniband cluster with GASNet communication library. We develop one-sided and non one-sided implementations of inter-node work stealing, and evaluate the performance and efficiency of the work stealing implementations.

Keywords: task parallelism, lightweight multithreading, thread migration, inter-node work stealing, remote direct memory access, Infiniband

1. Introduction

Dynamic, hierarchical, and fine-grain parallelism are increasingly believed to play a key role in programming extreme scale systems, to achieve load balancing, to hide latency, to combat against performance variability, and to enhance programmability. Systems supporting such parallelism, which we collectively call *task-parallel* systems, have been widely adopted in shared memory machines [1], [2], [3], [4], [5], [6]. On large-scale distributed memory environments, research efforts are under way but we are yet to see a widely used implementation, as there are many intricate issues associated with the lack of shared memory and the scale of such machines. They include how to implement dynamic task migration without shared memory, what happens on pointers upon migration, how to scale dynamic load balancing to extremely large systems, etc.

Previous research efforts take a variety of forms; some systems support task parallelism on distributed machines, but do not support global load balancing [7], [8]; many implement a restrictive "bag of tasks" or "atomic tasks" model as a target (see Section 2). There are a few systems general enough to express fork-join parallelism, but to the best of our knowledge, all assume tasks are tied to a specific processor, which may lower processor utilization. In addition, most systems supporting fork-join parallelism are built with a significant source or bytecode processing [9], [10], which renders them difficult to reuse across multiple languages. The situation contrasts with shared memory machines, where we have task-parallel *libraries* [3], [4], [5], which can be used from most C/C++ programs compiled with ordinary C/C++ compilers.

The main goal of the present work is to narrow this gap, by implementing a library satisfying the following.

- It supports general lightweight threading primitives (creating a thread and joining a thread) on large-scale distributed memory environments.
- In particular, it supports general migration of native threads across nodes, written in ordinary C/C++ programs.
- It does not require a special source code processing or a new code generator; the user program can be compiled with ordinary C/C++ compilers.

The main issue is how to migrate native threads, whose stack may contain ambiguous pointers. In our prior work [11], we proposed a new implementation scheme, *uni-address*, which overcomes a scalability limitation of a previously proposed *iso-address* [12] scheme. We implemented a multithread library, *uni-address threads*, providing inter-node work stealing on Fujitsu FX10 system [13] and tested its scalability up to 3,840 cores. The

[1] The University of Tokyo, Bunkyo, Tokyo 110–0003, Japan
[a) akiyama@eidos.ic.i.u-tokyo.ac.jp
[b) tau@eidos.ic.i.u-tokyo.ac.jp

prior work presented the following contributions:

- We proposed a new native thread migration scheme, called *uni-address*, which significantly reduces the usage of virtual address space.
- Based on this technique, we designed and implemented a work stealing scheduler with *one-sided* task stealing, which can take advantage of Remote Direct Memory Access (RDMA).

The goal of this work is to evaluate the efficiency of the uni-address scheme on an x86-64 Infiniband cluster and investigate the performance impact of one-sided work stealing by RDMA features. In particular, existing works [1], [14], [15], [16] mentioned that one-sided work stealing achieves better scalability of load balancing; however, it has not been demonstrated especially with a focus on the "one-sided" feature on large-scale distributed memory environments.

Novel contributions of this paper are as follows:

- We port uni-address threads to an x86-64 Infiniband cluster. The implementation is done on top of a state-of-the-art communication library, GASNet [17]. In order to implement RDMA-based work stealing, we have to manipulate a task queue on a remote processor with RDMA READ/WRITE/fetch-and-add operations. Infiniband network provides all of the operations, but GASNet does not provide RDMA fetch-and-add, and only provides RDMA READ/WRITE and Active Messages [18] (AM). Therefore, we have to emulate RDMA fetch-and-add in software. We develop two remote fetch-and-add implementation—RDMA-emulated implementation and AM-based implementation. The implementation details are described in Section 6.
- We evaluate the virtual memory usage and the performance of uni-address threads on 1,800 processing cores of an x86-64 Infiniband cluster in three benchmark programs: Binary Task Creation, Unbalanced Tree Search, and NQueens. We confirmed all the benchmarks works with less than 136 KB virtual memory for work stealing of native threads in each processor, and achieved more than 98% parallel efficiency of load balancing on 1,800 processing cores, relative to the results on 225 processing cores.
- We compare the performance of the RDMA-emulated and the AM-based implementation in order to evaluate the performance impact of RDMA-based work stealing. In the comparison, we confirmed that message handling arising from the use of active messaging degrades the scalability of work stealing.

The rest of this paper is organized as follows. Section 2 discusses related work. In Section 3, we describe the task model assumed in this paper. Section 4 presents the iso-address thread migration scheme and its scalability limitations. In Section 5, we describe the uni-address scheme and present the implementation of RDMA-based work stealing and inter-task synchronization on top of it. In Section 6, we describe the communication layer for implementing uni-address threads on an Infiniband cluster. Section 7 shows an experimental evaluation of the uni-address scheme, and Section 8 concludes this paper.

2. Related Work

To position the present work in context, this section gives a taxonomy of task-parallel systems on distributed memory environments. By task-parallel systems, we broadly mean systems that support creation of tasks at runtime and their dynamic load balancing. The implementation strategies and complexities are heavily affected by synchronization patterns supported by the system.

2.1 Bag of Tasks

Some systems such as Scioto [15], [19] and X10/GLB [20] only support independent "bag of tasks"; tasks neither synchronize nor communicate with other tasks. Note that X10 supports async-finish primitives, but native X10 tasks do not migrate across nodes (places); X10/GLB is a system built on top of X10 for global load-balancing. The bag of tasks is particularly simple to implement; it suffices to represent a task with a data structure (e.g., a function pointer + arguments to the function) and exchange the task structure among nodes to achieve load balancing. The bag of tasks is clearly very restrictive and cannot express many important divide-and-conquer algorithms naturally.

2.2 Atomic Tasks

Some other systems support dependencies (synchronizations) among tasks but assume a task is "atomic," in the sense that a task never blocks and always runs until completion once it gets started [9], [21]. We say such systems support "atomic tasks" model in the rest of the discussion.

Atomic tasks admit an implementation strategy similar to that for the bag of tasks, with the only difference being that it has to keep track of the status (ready to execute or not) of each task. From the programmability standpoint, this model forces a cumbersome programming style in which a logically sequential flow of computation needs to be "split" at each synchronization point and data used by the continuation of a synchronization must be manually packaged as a data structure. Arguably, atomic tasks are not for human programmers and can only be useful as a compiler target.

2.3 Fork-join and More General Models

Then there are systems that support a natural expression of fork-join parallelism or more general synchronization patterns. Examples are abundant on shared memory environments (Cilk [1], OpenMP tasks [2], TBB [3], MassiveThreads [4], Qthreads [5], Java fork-join [6]), but scarce on distributed memory environments; notable exceptions are Satin [10], Tascell [22], HotSLAW [23], and Grappa [24]. A task can create any number of child tasks and then call a "wait" function that waits for its outstanding children to finish. The calling task suspends until its children finish and then continues; the programmer does not have to package variables used after the synchronization. **Figure 1** contrasts the Fibonacci function in atomic tasks and fork-join.

Implementation of fork-join is more involved than atomic tasks, as it is now the system's responsibility to package the variables used by the continuation of a synchronization. Load bal-

```
thread Fib(cont int k, int n) {
  if (n < 2) {
    send_argument(k, n);
  } else {
    cont int x, y;
    spawn_next Sum(k, ?x, ?y);
    spawn Fib(x, n - 1);
    spawn Fib(y, n - 2);
  }
}
thread Sum(cont int k, int x, int y) {
  send_argument(k, x + y);
}
```

```
long fib(long n) {
  if (n < 2) {
    return n;
  } else {
    long r0, r1;
    r0 = spawn fib(n - 1);
    r1 = spawn fib(n - 2);
    sync;
    return r0 + r1;
  }
}
```

Fig. 1 Fibonacci in atomic tasks model (left [9]); and in fork-join model (right [1]).

ancing entails passing the representation of the migrating task's continuation between workers. In procedural programming languages such as C, a task's continuation is essentially its stack of activation frames.

2.4 Implementing Fork-join with Tied Tasks

To avoid the complication that stems from such "continuation passing" between workers, many of the systems mentioned above, including Satin, HotSLAW, and Grappa, avoid migrating tasks already started; when a task is created, a task is put in a task pool; only before it gets started can it be stolen by other workers. In other words, once a task gets started by a worker, it is "tied" to the worker. This scheme allows an implementation strategy similar to atomic tasks, as a yet-to-be-started task can be simply represented by a function pointer + its arguments, similarly to the atomic tasks. On the other hand, it can lose some opportunities for load migration and thus potentially lower processor utilization.

Despite its potential performance problem, this scheme seems popular as it can be readily implemented by ordinary procedure calls and returns [3], [6], [10], [23]; when a worker encounters a synchronization point, it repeats executing a task in its local task pool or stealing one from others, until all tasks it waits for finish. Either way it is just an indirect procedure call. The technique, which seems first described in Ref. [25], is particularly attractive when implemented in high-level languages, e.g., Java, that do not support non-local jumps.

2.5 "Genuine" Task Migration

This paper focuses on an efficient implementation scheme supporting "genuine" task migration, in the sense that a task can migrate even after it is started. Specifically, we implement a work stealing scheduling algorithm (child-first execution order upon task creation + FIFO stealing) first proposed by Mohr et al. in Ref. [26] and adopted in Cilk [1] and other systems [4], [5], which are possible only when a task's continuation can migrate at each task creation and each synchronization point. This particular scheduling policy is important both in theory and in practice. In theory, an established time bound of the work stealing scheduler [27] applies only when any task, started or not, can be stolen by any idle worker. A bound on extra cache misses [28] applies only when each worker preserves the serial order of execution except when a task steal happens. In practice, the work stealing scheduler is important because it tends to migrate coarse-grain tasks and its execution order tends to minimally deviate from the sequential execution, making it easy to reason about tasking overhead and data locality.

In shared memory environments, migrating a task in the middle of its execution can be done simply by passing the address of the stack, as both workers share the same address space [1], [4]. In distributed memory environments, it entails copying the stack frames of the task. Since address spaces are not shared by workers, pointers from/to the stack further complicate the issue. A scheme proposed in the literature, iso-address, as well as our proposed scheme, uni-address, are further elaborated in Section 4 and 5. There are two systems using iso-address thread migration—Adaptive MPI [29] and Charm++ [30]. Adaptive MPI uses iso-address to migrate MPI processes for dynamic load balancing on distributed memory environments, and Charm++ uses iso-address to support migratable threads as threaded entry methods for concurrent objects.

2.6 Distributed-memory Implementation of Work Stealing

When implementing work stealing in distributed memory environments, we have to manipulate a task queue on a remote node and steal a task from an entry in the queue. In commodity clusters, work stealing is implemented on top of a underlying message passing mechanism such as TCP/IP sockets and MPI [10], [31]. On the other hand, in HPC clusters built with dedicated interconnects such as Infiniband, we can implement one-sided work stealing with Remote Direct Memory Access (RDMA) features provided by such a interconnect.

To the best of our knowledge, Olivier et al. [14] and Dinan et al. [19] are the first implementations of work stealing on top of RDMA features. In Olivier et al. [14], mutual exclusion of a task queue is performed with the shared memory abstraction of Unified Parallel C (i.e., it implements a shared memory mutual exclusion algorithm), but the work stealing implementation is not one-sided; the work stealing implementation requires the victim's assistance to reduce the locking cost at local task queue operations. On the other hand, Dinan et al. [19] implements one-sided work stealing with RDMA READ/WRITE and locks provided by ARMCI [32]. Here, ARMCI locks implement a simplified version of the bakery algorithm [33] with a communication server approach; i.e., a lock operation sends a lock request to a communication thread corresponding to the ARMCI process which owns the lock object, and then the communication thread locally performs a lock operation. Therefore, the ARMCI lock operation is one-sided; it does not interrupt the remote processor.

In large-scale distributed memory environments, scalability of work stealing heavily relies on the mutual exclusion of a task queue. Dinan et al. [15] compared mutual exclusion implementations based on ARMCI locks and spinlocks for work stealing. Here, the spinlock is implemented with the ARMCI atomic swap operation. As a result, this work achieves good scalability with spinlocks.

Prior work [11] implements mutual execution of work stealing in a similar approach to Ref. [15]. The implementation uses the THE protocol [1], a state-of-the-art mutual exclusion protocol for work stealing on shared memory environments, with one-sided remote memory access operations such as RDMA READ-/WRITE and remote fetch-and-add. Because the implementation is built on a Fujitsu FX10 system and the system does not provide RDMA fetch-and-add, the implementation emulates RDMA fetch-and-add in a communication server approach; the implementation assigns a processor within a node to handle remote fetch-and-add requests.

Our work ports the work stealing implementation from FX10 system to an Infiniband cluster with both one-sided approach (RDMA-emulated) and non one-sided approach (AM-based). Furthermore, our work demonstrates the performance impact of one-sided work stealing in large-scale distributed memory environments by comparing the load balancing performance of our two work stealing implementations.

3. Task-parallel Model

In this section, we describe a task-parallel model that we assume in this paper. In our model, a *task* is a unit of parallelism. A program starts with a main task, and there is no parallelism at this program point. In order to utilize the parallelism of computational resources, a programmer has to spawn new tasks. Our model provides fork-join primitives for creating a task and waiting for the completion of a task, shown in **Fig. 2**.

Each task has its own call stack. This stack memory is managed according to underlying C calling convention so that a task can use C programming language features such as local variable accesses and function calls.

In our task model, tasks are automatically load-balanced. The runtime system automatically detects load imbalance, and then migrates tasks among processors across shared memory nodes. Therefore, programmers can write programs in a processor-oblivious manner; they do not have to be concerned about processor and node boundaries.

In order to support automatic load balancing among computational nodes, tasks should be isolated. A call stack is task-local and unable to be shared among tasks, and a task must not access the call stack of another task by passing C pointers. In order to share data among tasks, programmers can use task arguments and global memory features provided existing global address space frameworks such as partitioned global address space systems and distributed shared memory systems.

Our task model permits the runtime system to migrate a task between processors at *migration points*. A migration point is defined as a program point where a task may switch to another task. They include points where a task creates a new task and points where a task waits for the completion of another task.

```
template <class T, class F, class... Args>
task<T> spawn(F f, Args... args);

template <class T>
void join(task<T> t, T *result);
```

Fig. 2 Fork-join task API.

4. Iso-Address

This section reviews *iso-address*, which inspires our work most. As noted in Section 2, migrating a task already started involves copying the currently active stack frames of the task—representation of variables used in the rest of the task. Simply copying the stack frames does not complete the job, however, as there are pointers from/to stack, which might need to be "fixed" when a stack moves across address spaces and changes its address.

One way to solve this problem is to implement a compiler that leaves enough information about stack frame and data layout, so that the runtime system knows which slots of a stack frame or which fields of a structure might contain pointers that need to be fixed. This approach is good for type-safe languages but is very difficult if not impossible to apply to languages with ambiguous pointers such as C and C++.

Iso-address [12] is a scheme that makes fixing pointers unnecessary, by ensuring stacks are copied into exactly the same address in the new address space. Intra-stack pointers (pointers from within the migrating stack to inside it) just continue to be valid after migration. Pointers to heap objects outside the migration stack (heap objects) are also copied to the same address in the new address space; they are allocated by a special memory allocation routine `pm2_isomalloc` so that the system knows where they are. In Ref. [12], it is assumed that pointers to such heap objects are not passed between threads and there are no inter-stack pointers.

The main advantage of the iso-address scheme is that it just works with languages with ambiguous pointers and their ordinary compilers unaware of migration. Also, as the simple bitwise copy suffices to copy a stack, migration is efficient.

Bringing this technique to a large-scale environment has several problems, however.

(1) Iso-address requires the address of each live stack to be *globally* unique in the entire system, and *each* node to reserve these addresses. This results in consuming a huge *virtual* address space.
In parallel divide-and-conquer algorithms, typical use cases of task-parallel systems, the number of simultaneously live tasks is roughly the maximum depth of the task tree × the number of hardware concurrency (workers) [27]; thus, with the concurrency of the largest machines already surpassing three million [34], and expected to only increase, allocating a few hundred kilobytes per stack has a risk of running out a *virtual* address space.
As a point of reference, assume we have 4 million (2^{22}) hardware concurrency, the depth of the task tree is ten thousand

or 2^{13}, a number that happens in an unbalanced tree search benchmark, and the size of a stack for each task is a modest 16 KB (2^{14})[*1]; the total virtual address space that needs to be reserved for tasks is $2^{22+13+14} = 2^{49}$, which surpasses the virtual address space size the current x86-64 processors support (2^{48}).

(2) While allocating a virtual address does not immediately translate into consuming a physical memory, address usage in iso-address may still result in significant growth in physical memory usage. Operating systems generally allocate a physical page for a logical page when it is touched for the first time. Microscopically, when a node steals a task and hosts the incoming stack to its designated address, it may be the first touch on this page by that node.

More quantitatively, the growth is determined by how many nodes, on average, will ever touch each logical page in the reserved area. When a particular page is reused by r distinct tasks in the lifetime of the application and each task migrates m times on average, tasks allocated on that page experience mr migrations in total; thus, roughly $(1 + mr)$ physical pages will be committed for that page in the entire system. Note that we expect r to be small ($\ll 1$), yet for long running applications, whose r is proportionally large, the overall growth may be significant.

Also, note that it will also increase the number of page faults due to on-demand paging. Note that the scheme relies on on-demand paging in an essential way; it is obviously not possible to populate (pre-fault) the pages. In a SPARC64IXfx processor, a page fault takes 21 K cycles on average, so this may degrade work stealing performance considerably.

(3) Iso-address has another issue that it practically prohibits us taking advantage of the now common hardware support of Remote Direct Memory Access (RDMA), or one-sided communication, for copying a stack upon migration. With RDMA, a node can trigger a data transfer without involving the host CPU on the target node. One-sided task stealing is a common practice in shared memory machines [1], and has been proved important in distributed memory environments [15].

The problem is that, RDMA generally requires the region accessed by a remote node to be pinned to the physical memory, but we obviously do not have the luxury of pinning the entire region reserved for stacks. We might consider a more sophisticated strategy that pins (only) the migrating stack on demand, but it would hinder the original benefit of RDMA—stealing a task without involving the victim.

(4) Less imminent but potentially an important issue is that, the luxury use of virtual address space may conflict with other techniques relying on sparsely populating a huge linear address range.

[*1] This estimation (16 KB) practically assumes each task has its dedicated linear stack, so as to be compatible with ordinary C compilers. Alternatively, the ordinary procedure calls may obtain frames from a general free list shared by many tasks (heap frames, split stack, cactus stack, etc.), in which case the initial stack size per each task can be made minimum (just a single frame, in an extreme case). One might expect task stacks not to grow to their limits at the same time, in which case the maximum virtual address range that must be reserved can be reduced accordingly. Yet, as we want to impose a minimum allocation size (e.g., 4 KB) to keep the overhead of frame allocation low, the overall conclusion is similar.

5. Uni-address Scheme

5.1 The Basic Idea

In order to address the problem of iso-address, which is reserving a huge virtual address space for stacks, we propose a uni-address scheme and RDMA-based work stealing on top of it. In order to simplify the exposition, we first describe its basic idea without performance considerations.

Recall that the iso-address scheme maintains the validity of intra-stack pointers by copying a stack into the same address upon migration. The key idea behind the uni-address scheme is that, in order to maintain the validity of intra-stack pointers, all we need to guarantee is to map the stack on the designated address *when the task is actually running.* Stacks of not running tasks can be put at *an arbitrary address*; we put them into a reserved, RDMA-accessible region to make them available for task stealing.

A crucial assumption is that there are no pointers pointing to a stack from outside (i.e., there are neither inter-stack pointers nor heap-to-stack pointers). Were there such pointers, it is unsafe to relocate stacks even when the task is not running. Iso-address also makes the assumption.

For stack-to-heap pointers, we are separately working on a global address space library supporting explicit global references and assume objects potentially referenced by multiple threads are always referenced by a global pointer. To dereference a global pointer, a function must be called, which can trigger data transfer if necessary. We currently do not support thread-private heaps that can be referenced by ordinary C pointers, but it is possible to add a mechanism similar to `pm2_isomalloc`. Further details about the memory model of our system are beyond the scope of the paper and will be addressed in a separate paper.

To summarize, in its simplest and crudest form, the uni-address scheme works as follows.

(1) It creates a separate address space for each worker (a hardware concurrency such as a CPU core and a hardware thread).

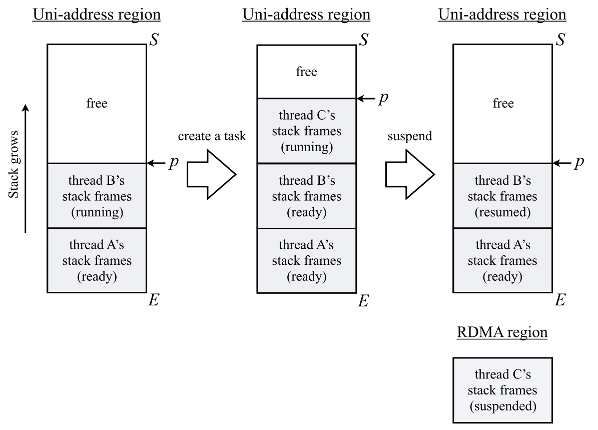
(2) It reserves a region of virtual addresses accommodating a *single* stack. This region is *the* stack for running a task, always used to run a task. We call this region *the uni-address region.*

(3) It reserves a region of virtual addresses accommodating stacks for not running tasks and pins them to the physical memory. Their addresses do not matter, as long as they can be reached from other nodes by RDMA. We call the region *RDMA region.*

(4) Whenever a task switches, the previously running task is swapped out from the uni-address region to the RDMA region and the next task is brought into the uni-address region.

Unlike iso-address, which *never* changes the stack addresses even if the task is not running, we do not have to reserve a sparsely used huge virtual address range; we only have to reserve a region large enough to accommodate the number of tasks *simultaneously live in a single address space.*

Note that we have a separate address space for each worker so



Uni-address region
S
free
p
thread B's stack frames (running)
thread A's stack frames (ready)
E
Stack grows
create a task
Uni-address region
S
free
p
thread C's stack frames (running)
thread B's stack frames (ready)
thread A's stack frames (ready)
E
suspend
Uni-address region
S
free
p
thread B's stack frames (resumed)
thread A's stack frames (ready)
E
RDMA region
thread C's stack frames (suspended)

Fig. 3 Multiple threads on uni-address region.

that all workers can allocate *the* uni-address region at the same virtual address. In practice, it means we need to create a process per core. This is to guarantee that, at any moment of execution, any ready task can be run by any idle worker. In order to reduce the number of processes, we might alternatively have multiple workers and uni-address regions in each address space. In this case, a task allocated to a particular uni-address region can migrate to uni-address regions of the same address (of a different address space); in unlucky cases, there may be many unfilled regions and many ready yet not running tasks, due to their unmatching addresses. This may lower processor utilization. Further elaborating and quantifying the impact of this approach is our future work. The present paper explores only the basic, process-per-core approach.

5.2 An Optimized Scheme

This crude scheme just mentioned is simple but obviously inefficient, as it incurs two stack copies upon every context switch. Especially in the child-first work stealing scheduler, which immediately switches to the new child upon every task creation, it will be very inefficient.

To address this issue, we developed a better stack management technique.

The key observation is that, we do not have to allocate all stacks on the same address. The real requirement is each task, when executed, always occupies the same address as the address allocated to it upon creation. At least conceptually, a new stack can be allocated at any address in the uni-address region, as long as we ensure that the area the new stack may grow into is empty. More specifically, our memory management works as follows (**Fig. 3**):

(1) Assume the address range of the uni-address region is $[S, E)$.

(2) Each address space manages a pointer p in the uni-address region (i.e., $S \leq p < E$) pointing to the next free address, much like the stack pointer of sequential languages. Assuming a stack grows downwards, we have addresses $\in [p, E)$ are used, and addresses $\in [S, p)$ are free.

(3) When a new task is created, its stack is allocated just below p, much like allocating a new frame from a linear stack, and the task immediately starts. We maintain an invariant that the running task occupies the lowest addresses of the used region.

```
1  void do_create_thread(context *ctx, thread_func_t f, void
            *arg) {
2      // push the parent thread
3      taskq_entry entry;
4      entry.frame_base = ctx->rsp;
5      entry.frame_size = (uint8_t *)ctx->parent_ctx +
           sizeof(context) - ctx->rsp;
6      entry.ctx = ctx;
7      TASK_QUEUE_PUSH(entry);
8
9      // start a child thread
10     current_worker()->parent = ctx;
11     f(arg);
12
13     // pop the parent thread
14     bool ok = TASK_QUEUE_POP(&entry);
15     if (!ok) go_to_scheduler();
16
17 }
18 void create_thread(thread_func_t f, void *arg) {
19     context *parent = current_worker()->parent;
20     save_context_and_call(parent, do_create_thread,
21                           f, arg);
22     current_worker()->parent = parent;
23 }
```

Fig. 4 Optimized implementation of task creation function.

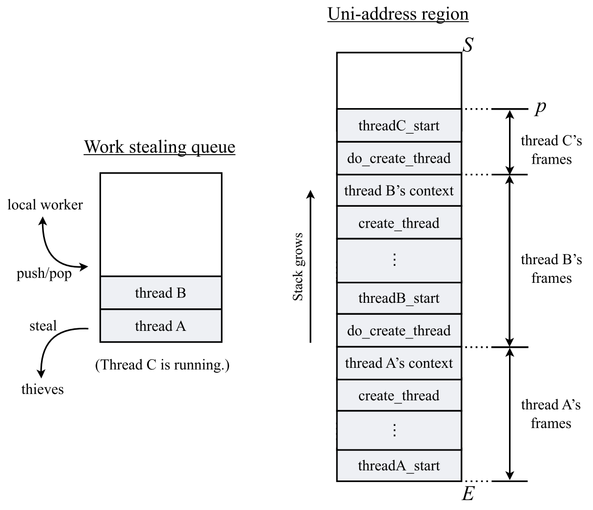


Fig. 5 A work stealing queue and the corresponding uni-address region.

(4) When a task is suspended, its stack is copied out from the uni-address region into any free address in the RDMA region, and the task just above it is resumed if there is one. This way, we maintain the above invariant.

(5) Only when the uni-address region becomes empty, does the process steal work from another node. Thus, the uni-address region of this process can accommodate any task.

In this scheme, a task creation is very efficient, as it is much like an ordinary procedure call, except that we need to save registers before task creation so that the caller can be stolen.

Figure 4 shows the implementation of task creation based on this idea, and **Fig. 5** illustrates a work stealing queue and the corresponding uni-address region. The task creation function `create_thread` saves the context of the running thread and call `do_create_thread` function (Line 19–20) by `save_context_and_call` function shown in Appendix A.1. Then, `do_create_thread` function first pushes an entry to the work stealing queue (Line 3–7). The entry contains information for resuming the parent thread when the thread is stolen. Next, `do_create_thread` function executes a given thread start function (Line 11). After the function call, it pops an entry from the work stealing queue. If it succeeds, the parent thread has not been stolen and resumes the thread after removing the saved context on the stack. Otherwise, the parent thread has been stolen, so the control goes to the scheduler code to execute waiting threads or perform work stealing. Executing a child thread does not evict the parent thread from the uni-address region. The overhead of task creation consists of only save and restoration of the parent thread and manipulations of the work stealing queue.

5.3 RDMA-based Work Stealing

Under random work stealing, a processor selects a victim processor and steals a task from the victim's task queue when the processor becomes idle. To steal a task with RDMA operations, the following memory regions are pinned to physical memory: the uni-address region, RDMA region, and work stealing queues. Under the requirement, we now explain the implementation of RDMA-based work stealing.

The implementation of a task queue is one of the most important parts in work stealing. A task queue is accessed from a local worker upon a task creation and a local exit from a task, and accessed from remote workers upon a work stealing. Therefore, a naive locking scheme for mutual exclusion does not scale well especially on large-scale distributed memory environments [15]. To address this issue, we implement THE protocol [1] with RDMA READ, WRITE, and fetch-and-add. THE protocol is used in several task-parallel systems on shared memory machines, such as Cilk [1] and MassiveThreads [4], and because it eliminates locking from local accesses to a task queue, it reduces the tasking overhead and improves the scalability of work stealing.

Figure 6 shows the pseudo-code of our work stealing implementation. A thief first selects a victim process and calculates the remote address of the task queue of the victim process. Next, the thief tries to lock the task queue with RDMA fetch-and-add operation, and if it fails, the steal process aborts. If the locking succeeds, the thief tries to steal an entry from the task queue. If the task queue is empty, the steal process aborts. Otherwise, the thief starts migrating the task in the stolen task queue entry. In task migration, we first calculate the remote address of the stack region for RDMA operations, and then perform an RDMA READ operation from the remote address to the uni-address region without changing the address of the thread stack. At this point, the context of the thread becomes valid; the saved register values and the contents of the stack become readable. Next, the thief releases the lock of the task queue and resumes the thread loaded to the uni-address stack.

```
void resume_remote_context(saved_context_t *sctx,
          taskq_entry e) {
    WAIT_QUEUE_PUSH(sctx);
    void *remote_base = get_remote_base(e.stack_base,
          victim);
    RDMA_GET(e.stack_base, remote_base, e.stack_size,
          victim);
    taskq_unlock(q, victim);
    resume_context(e.ctx);
}
bool steal() {
    int victim = select_victim_randomly();
    taskq *q = get_remote_taskq(victim);
    if (!try_lock(q, victim))
        return;

    taskq_entry e;
    if (!taskq_steal(q, victim, &e)) {
        taskq_unlock(q, victim);
        return false;
    }
    suspend(resume_remote_context, e.ctx);
    return true;
}
```

Fig. 6 Implementation for RDMA-based work stealing.

```
1  void resume_saved_context_1(saved_context *next_sctx) {
2      // restore stack frames
3      memcpy(next_sctx->stack_top, sctx->stack_buf,
             next_sctx->stack_size);
4      // restore the execution state of the next thread
5      resume_context(next_sctx->ctx);
6  }
7  void resume_saved_context(saved_context *sctx,
             saved_context *next_sctx) {
8      WAIT_QUEUE_PUSH(sctx);
9
10     /* after moving SP to unused area by the suspending
11        thread and resuming thread, call the function.
12        This is implemented in assembly. */
13     CALL_WITH_SAFE_SP(resume_saved_context_1,
14                       next_sctx);
15 }
16 void join(task<T> t, T *result) {
17     T value;
18     while (!try_join(t, result)) {
19         // first try to switch to a ready thread
20         bool ok = TASK_QUEUE_POP(&entry);
21         if (ok) {
22             suspend(resume_context, entry->ctx);
23         } else {
24             // start work stealing
25             ok = steal();
26             if (!ok) {
27                 // execute a waiting thread if the steal
               fails
28                 saved_context_t *sctx = WAIT_QUEUE_POP();
29                 suspend(resume_saved_context, sctx);
30             }
31         }
32     }
33 }
```

Fig. 7 Implementation of join function.

5.4 Inter-task Synchronization

In this section, we describe an implementation of inter-task synchronization in the optimized uni-address scheme, and we take join operation, an operation to wait for the exit of a thread, as an example. **Figure 7** shows the implementation. The join function checks whether the target thread has terminated or not with `try_join` function. If it has, the function returns with the result of the thread. Otherwise, the function suspends the running thread with `suspend` function, pushes the suspended thread to a wait queue, and switches to another thread. We have three kinds of threads as a target of a context switching: a ready thread in the work stealing queue, a suspended thread in the wait queue, and a thread stolen by work stealing. As mentioned in Section 5.2, the uni-address region has to be empty when a worker steals work from another worker. Hence, the join function first tries to resume a ready thread on the work stealing queue (Line 20). Next, it tries to steal a thread from another worker and resume it if the steal succeeds (Line 25). Otherwise, it tries to resume a suspended thread in the wait queue (Line 28–29).

```
1  typedef struct {
2      void *ip, *sp;
3      context_t *ctx;
4      uint8_t *stack_top;
5      size_t stack_size;
6      void *stack_buf;
7  } saved_context_t;
8  typedef void (*suspend_func_t)(saved_context_t *sctx,
             void *arg);
9
10 void do_suspend(context_t *ctx, suspend_func_t f,
11                 void *arg) {
12     // calculate the stack range of the thread
13     uint8_t *parent_sp = current_worker()->parent->rsp +
               sizeof(context_t);
14     uint8_t *stack_top = ctx->rsp;
15     size_t stack_size = parent_sp - stack_top;
16
17     // pack the suspending thread
18     saved_context_t *sctx = pinned_malloc(sizeof(
               saved_context_t));
19     sctx->ip = ctx->rip; sctx->sp = ctx->rsp;
20     sctx->ctx = ctx; sctx->stack_top = stack_top;
21     sctx->stack_size = stack_size;
22     sctx->stack_buf = pinned_malloc(stack_size);
23     memcpy(sctx->stack_buf, stack_top, stack_size);
24
25     // execute a thread start function
26     current_worker()->parent = ctx;
27     f(sctx, arg);
28     // not reached
29 }
30 void suspend(suspend_func_t f, void *arg) {
31     context_t *parent = current_worker()->parent;
32     save_context_and_call(prev_ctx, fp, do_suspend, arg);
33     // here, this thread is resumed
34     current_worker()->parent = parent;
35 }
```

Fig. 8 Implementation of suspend function.

Figure 8 shows the suspend function; it saves the context of the running thread (Line 32), swaps out the stack frames of the thread from the uni-address region to the RDMA region (Line 12–23), and calls a given function resuming a next thread (Line 27).

In such an implementation of join operation, a swap-out of a thread in the uni-address region occurs only when the target thread is executing on another worker due to work stealing. In typical cases where the parent thread is not stolen, the join function only confirms termination of a target thread by `try_join` function.

6. Implementation on Infiniband Clusters

This section describes the implementation of the communication layer of uni-address threads for Infiniband clusters. Uni-address threads requires RDMA READ/WRITE/fetch-and-add operations to manipulate a task queue on a remote processor with the THE protocol. In order to utilize RDMA operations, we use the GASNet communication library for ease of implementation. GASNet is a state-of-the-art communication library supporting Infiniband platforms.

6.1 Communication Primitives in GASNet

GASNet provides two communication mechanisms—Active Messages [18] and Remote Memory Access. Active Messages is an asynchronous remote proceduce call mechanism; i.e., a node (similar to MPI's process) sends a message with a message handler which is executed at the target node when the messages arrives at the node. Unlike TCP/IP sockets and MPI, Active Messages does not need to call a *recv* function to receive a message; instead, Active Messages requires a receiver to call a *polling* function which receives incoming messages and executes their message handlers. **Figure 9** shows a part of Active Messages API in GASNet. `gasnet_AMRequestShortM` sends an active message consisting of a message handler `handler` and its arguments `arg0`, ..., `argM-1` to the specified node `node`. Receivers need to call the `gasnet_AMPoll` function periodically to handle incoming messages, and the function internally executes a message handler when an incomming message arrives.

```
int gasnet_AMRequestShortM(
        gasnet_node_t node,
        gasnet_handler_t handler,
        gasnet_handlerarg_t arg0,
        ...,
        gasnet_handler_arg_t argM-1);
void gasnet_AMPoll();
```

Fig. 9 An example of Active Messages API in GASNet.

```
void gasnet_get(void *dest, gasnet_node_t node, void *src
          , size_t nbytes);
void gasnet_put(gasnet_node_t node, void *dest, void *src
          , size_t nbytes);
```

Fig. 10 An example of remote memory access API in GASNet.

GASNet also provides a communication mechanism called Remote Memory Access (RMA) which performs reads and writes to memory at remote nodes. In Infiniband clusters, GASNet implements RMA with Remote Direct Memory Access (RDMA) operations which is *one-sided*; i.e., an RDMA operation can access remote memory *without involving the remote processor*. Therefore, compared to Active Messages, the use of RMA can avoid a message handling cost on the remote processor and reduce communication latency. **Figure 10** shows a part of Remote Memory Access API in GASNet. GASNet RMA does not provide atomic operations such as compare-and-swap and fetch-and-add. Therefore, in order to manipulate data structures on remote nodes in a one-sided manner, we have to implement such kinds of operations with Active Messages.

6.2 Remote Fetch-and-Add Implementations

A goal of this paper is to investigate and demostrate the performance impact of one-sided work stealing with RDMA operations. To achive this goal, we chose two strategies implementing software-based remote fetch-and-add operations—*RDMA-emulated* implementation and *AM-based* implementation. RDMA-emulated fetch-and-add is implemented to emulate truly one-sided remote fetch-and-add operation for one-sided work stealing, and AM-based fetch-and-add is implemented to emulate non one-sided work stealing to compare performance with one-sided work stealing.

Both strategies implement remote fetch-and-add with Active Messages as follows:

(1) A processor sends a fetch-and-add AM request to a target processor.

(2) The target processor handles the request on the AM polling function, and performs a local fetch-and-add operation.

(3) The target processor sends an AM reply message (acknowl-

edge) to the initiated processor.

The difference between the two implementation strategies is when and where to handle fetch-and-add AM requests. In RDMA-emulated fetch-and-add, a worker does not call the AM polling function during computation; instead, we assign a physical processor, called a *communication-assisting processor*, for each worker to handle AM requests. A communication-assisting processor continuously calls the AM polling function which internally handles remote fetch-and-add requests. The uni-address threads implementation on top of RDMA-emulated fetch-and-add requires additional processors for communication, but many supercomputer interconnects support hardware-based remote atomic operations (e.g., Infiniband Verbs, Fujitsu Tofu interconnect 2, Cray Aries); therefore, this problem is not so critical on such environments.

In AM-based fetch-and-add, a worker handles remote fetch-and-add requests when the AM polling function is explicitly called by programmers or a thread is created (i.e., a thread creation internally calls the AM polling function). The uni-address threads implementation on top of AM-based fetch-and-add incurs additional overhead to thread creation because the AM-based fetch-and-add calls the AM polling function. This overhead includes message handling for fetch-and-add requests, and this may increase the threading overhead.

To summarize, RDMA-emulated implementation focuses on the emulation of hardware RDMA fetch-and-add; i.e., the fetch-and-add operation does not interrupt the execution of a remote procesor, but it requires additional processors. On the other hand, AM-based implementation does not require additional processors but the remote fetch-and-add interrupts the execution of a remote processor; i.e, it is not one-sided. In the case of implementing inter-node work stealing, AM-based remote fetch-and-add introduces additional overhead to thread operations because they have to call the AM polling function internally to handle remote fetch-and-add requests. This overhead contains message handler executions which occurs N times such that N is the number of work stealing. In general, the number of work stealing increases in superlinear manner to the number of processors (shown in Section 7); therefore, AM-based remote fetch-and-add degrades scalability of work stealing.

In order to mitigate the overhead of software-based implementation of remote fetch-and-add, we implemented *empty check* before locking a task queue on a remote processor with remote fetch-and-add. Empty check confirms the size of a task queue with an RDMA READ operation before locking the task queue. Owing to this confirmation, we can avoid performance issues on software-based remote fetch-and-add in the case that a task queue is empty, and reduce the performance impact from the software-based fetch-and-add implementation.

7. Experimental Evaluation

This section shows the experimental evaluation on the efficiency and the performance of uni-address threads. We conducted all experiments on the TSUBAME2.5 supercomputer, which is an x86-64 Infiniband cluster, at Tokyo Institute of Technology. **Table 1** shows the system configuration of the TSUBAME2.5 supercomputer. The TSUBAME2.5 supercomputer has GPGPUs but we do not use them in the following experiments. We used up to 300 nodes for the experiments, and we used up to 1,800 cores for computation and 1,800 cores for communication assistance in RDMA-emulated implementation of work stealing. Confidence intervals in the following figures are calculated with a 95% confidence level.

Table 1 Experimental setup.

TSUBAME2.5 supercomputer	
CPU	Intel Xeon X5670 2.93 GHz 6 cores × 2 sockets
Memory	54 GB/node
Interconnect	Infiniband QDR
OS	SUSE Linux Enterprise Server 11 SP3 (GNU/Linux 3.0.76-0.11)
Compiler	GCC 4.7.2 (option -O3)
Library	GASNet 1.24.2 (ibv-conduit), MPICH2 3.1

For comparison, we used an existing lightweight multithread library, MassiveThreads [4], which is one of the most lightweight multithread libraries to the best of our knowledge. Differently from uni-address threads, MassiveThreads only provides intra-node load balancing, and it implements a child-first work stealing scheduler just like uni-address threads. In our experiments, we used MassiveThreads 0.95.

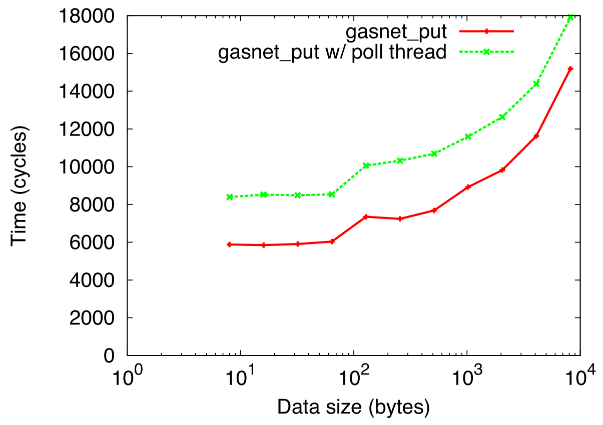
Figure 11 shows the communication latencies of GASNet remote memory access operations on TSUBAME2.5. Figure 11 (a) and Fig. 11 (b) presents the execution time of `gasnet_put` and `gasnet_get` which corresponds to RDMA WRITE and READ. "Poll thread" in Fig. 11 means remote memory access latencies when there is an AM polling thread which is an OS thread continuously calling GASNet AM polling function. The polling thread is placed on a different physical CPU core from a compute core. These experiments with a polling thread are for investigating the performance difference derived from the communication performance with or without the presence of a polling thread when comparing the performance of RDMA-emulated and AM-based work stealing. In Fig. 11, `gasnet_put` and `gasnet_get` takes 5.9 K and 8.2 K cycles, respectively, in 8 bytes data transfer.

On the other hand, in the presence of the polling thread, `gasnet_put` and `gasnet_get` takes 8.4 K and 11.6 K cycles, respectively, in 8 bytes data transfer. We can see the remote memory access latencies in the polling thread configuration is larger than the usual cases; we speculate this stems from the overhead of mutual exclusion on shared resources among `gasnet_put/get` and the polling function.

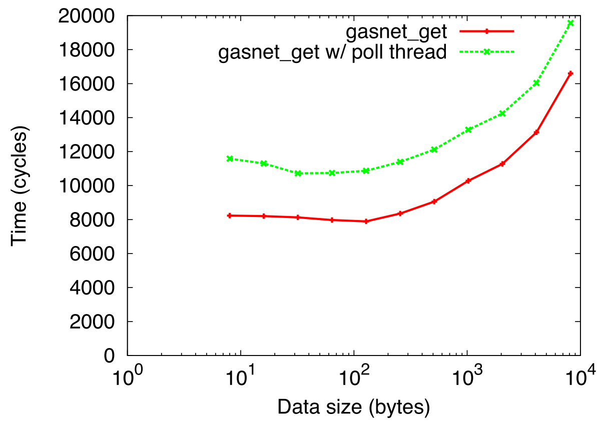
7.1 Benchmark Programs

To evaluate the scalability of work stealing in our library, we chose three benchmarks—Binary Task Creation (BTC) benchmark, Unbalanced Tree Search (UTS) benchmark, and NQueens solver:

BTC Binary Task Creation benchmark generates tasks recursively. Each task spawns two child tasks and waits for their completions. This benchmark has a parameter *depth*, which means the depth of a generated task tree. Load balancing of this benchmark is relatively easy for work stealing schedulers because generated task trees are balanced. The experiments performed with $depth = 18$ unless the parameter is



(a) `gasnet_put`



(b) `gasnet_get`

Fig. 11 GASNet remote memory access latency w/ and w/o polling thread on TSUBAME2.5 supercomputer.

Table 2 Thread creation overhead.

	Time (cycles)
Uni-address threads (RDMA-emulated)	184 cycles
Uni-address threads (AM-based)	353 cycles
MassiveThreads	122 cycles

explicitly described.

UTS Unbalanced Tree Search benchmark [35] is a benchmark to evaluate performance of dynamic load balancing algorithms and implementations. The UTS benchmark traverses an unpredictable, tree-based state space generated by a probability distribution. The detailed description of parameters of the UTS benchmark are in Ref. [35]. In our experiments, we chose a tree whose nodes have 0–4 child nodes based on a geometric distribution and performed experiments with tree cutoff depth = 18. The command-line arguments is "`-t 1 -r 0 -b 4 -a 3 -d 18`".

NQueens NQueens benchmark is a benchmark to calculate the number of possible ways to place N queens on a $N \times N$ chess board. The program used in our experiments is based on the one in BOTS Benchmark [36]. The experiments performed with $N = 18$ unless the parameter is explicitly described.

Because ordinary work stealing schedulers do not work well with parallel loops that appear in UTS and NQueens, we modified them to an efficient divide-and-conquer traversal over loops in which each task generates zero or two subtasks. Such an optimization is common in work stealing schedulers; in fact, Intel Cilk Plus [37] performs such an optimization for its `cilk_for` statement.

7.2 Thread Creation Overhead

We measured the overhead of a task creation in uni-address threads on Intel Xeon X5670 processor. For comparison, we also measured the overhead of task creation in MassiveThreads.

Table 2 shows the results. The task creation overhead of uni-address threads is 184 cycles and 353 cycles on average with RDMA-emulated implementation and AM-based implementation, respectively. Here, we can see that uni-address threads achieved a comparable performance to MassiveThreads, which is one of the most lightweight multithread libraries to the best

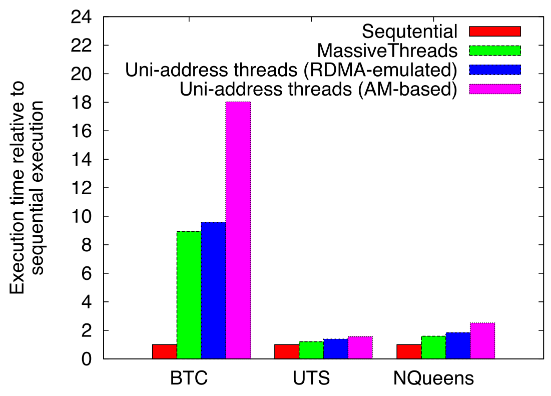


Fig. 12 Threading overhead with benchmarks.

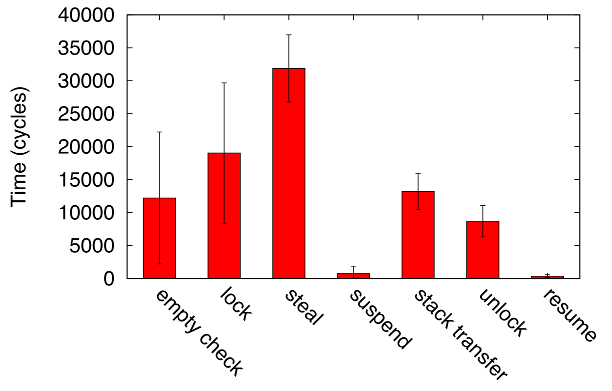
of our knowledge. The performance difference between RDMA-emulated and AM-based implementation is derived from an AM polling function call at thread creation.

Figure 12 shows the relative execution time to sequential execution of the three benchmarks. This experiment is performed with the parameter *depth* = 28 in the BTC benchmark, *depth* = 11 in UTS benchmark, and $N = 13$ in NQueens benchmark. As a result, the RDMA-emulated implementation is comparable to MassiveThreads; it takes at most 1.16x more execution time than MassiveThreads in the three benchmarks. In BTC benchmark, we can see the threading overhead is larger than the other benchmarks. It is because its task granularity is very small, i.e., a task only performs two child task creations or nothing.

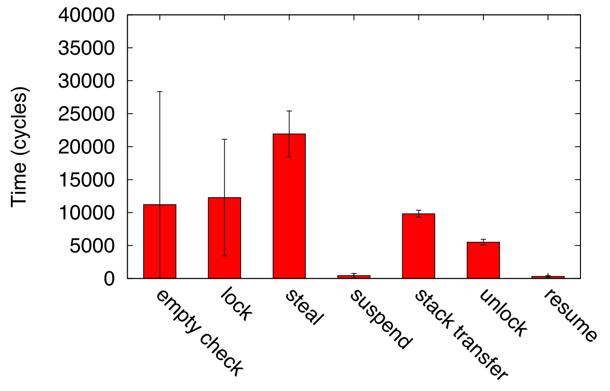
7.3 Work Stealing Time Analysis

We measured the execution time of work stealing in uni-address threads on TSUBAME2.5 supercomputer. In this experiment, two workers steal a single thread from each other and measure the execution time and its breakdown of a steal operation. The size of the stolen stack frame is 664 bytes.

Figure 13 shows the execution time of inter-node work stealing in RDMA-emulated and AM-based implementation of uni-address threads, and **Table 3** shows the operations constituting RDMA-based work stealing. A RDMA-emulated and AM-based work stealing takes 86 K and 61 K cycles in total, respectively, and suspend and resume operations, which are the main sources



(a) RDMA-emulated work stealing



(b) AM-based work stealing

Fig. 13 Breakdown of work stealing time.

Table 3 Operations consisting of RDMA-based work stealing.

Operation	Description
empty check	A operation to check whether a remote task queue is empty or not. It consists of an RDMA READ operation.
lock	A lock operation for a remote task queue. It consists of a remote fetch-and-add operation.
steal	An operation to steal an entry from a remote task queue. It consists of two RDMA READ and an RDMA WRITE operations.
suspend	An operation to suspend a running thread.
stack transfer	An operation to transfer stack frames. It consists of an RDMA READ operation.
unlock	A unlock operation for a remote task queue. It consists of an RDMA WRITE operation.
resume	An operation to resume a stolen thread.

Table 4 The number of generated tasks or nodes in three benchmarks. *Time* is average execution time on 1,800 cores. *Stack usage* means maximum usage of the uni-address region.

Benchmark	Parameters	Total tasks or nodes	Stack usage	Fetch-and-add	Empty check	Time
Binary Task Creation	*depth* = 41	4,398 billion tasks	36,144 bytes	RDMA-emulated	Enabled	113.0 sec
				RDMA-emulated	Disabled	113.4 sec
				AM-based	Enabled	325.1 sec
				AM-based	Disabled	416.3 sec
Unbalanced Tree Search	*depth* = 18	439 billion nodes	132,224 bytes	RDMA-emulated	Enabled	108.2 sec
				RDMA-emulated	Disabled	108.6 sec
				AM-based	Enabled	141.0 sec
				AM-based	Disabled	150.0 sec
NQueens	*N* = 18	59 billion nodes	79,360 bytes	RDMA-emulated	Enabled	194.0 sec
				RDMA-emulated	Disabled	195.3 sec
				AM-based	Eanbled	272.7 sec
				AM-based	Disabled	-

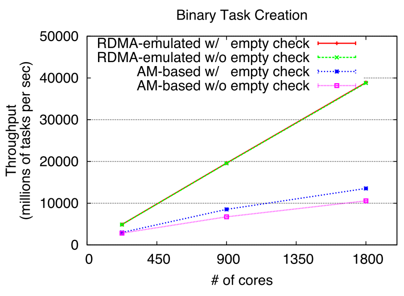
of the overhead of uni-address scheme, take 1.1 K and 0.8 K cycles, respectively. This overhead does not exceed 1.2% of the total work stealing time, and the other execution time is mostly spent by RDMA operations for task queue manipulations. The difference of the execution time between RDMA-emulated and AM-based work stealing stems from the difference of remote memory access latencies between the cases that a polling thread exists or not.

7.4 Load Balancing Scalability

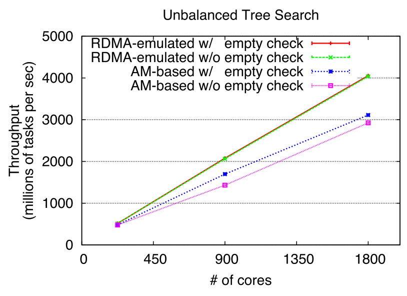
In this section, we evaluate stack memory usage in the uni-address region and the parallel performance of uni-address threads with the three benchmark programs. **Table 4** shows the basic information of the benchmarks—total number of generated nodes, stack memory usage in the uni-address region, and execution time on 1,800 compute cores. Note that all benchmarks worked with less than 136 KB virtual memory for thread stacks.

We performed strong scaling experiments of inter-node work stealing with the three benchmark programs—Binary Task Creation, Unbalanced Tree Search, and NQueens. In all the benchmarks, the parallel performance is reported as the total throughput of processed tasks or nodes per second. The number of processors used is from 225 to 1,800 cores. We evaluated the load balancing performance of our work stealing implementation with four configurations—RDMA-emulated work stealing with-/without empty check and AM-based work stealing with/without empty check.

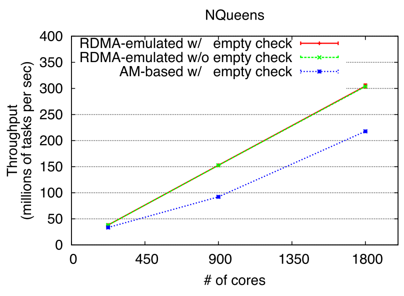
Figure 14 shows the load balancing performance of our work stealing implementation. RDMA-emulated work stealing achieved a good load balancing performance on 1,800 cores, 99%, 98%, and 99% efficiency relative to 225 cores results in BTC, UTS, and NQueens, respectively, regardless of whether empty check is performed or not. As the number of processing cores increases, the load balancing performance of AM-based work stealing begins to degrade, and the experiment on 1,800 cores reports 47% and 77% efficiency relative to 225 cores without empty check in BTC and UTS benchmark, respectively. In RDMA-emulated work stealing, empty check improves load bal-



(a) Binary Task Creation

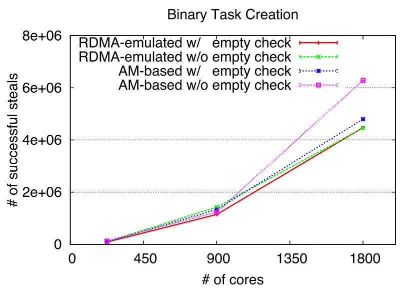


(b) Unbalanced Tree Search

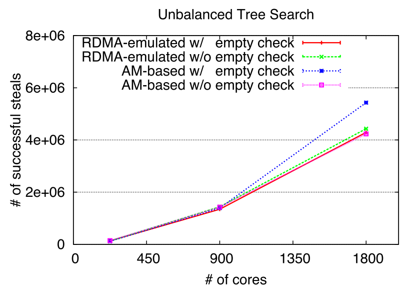


(c) NQueens

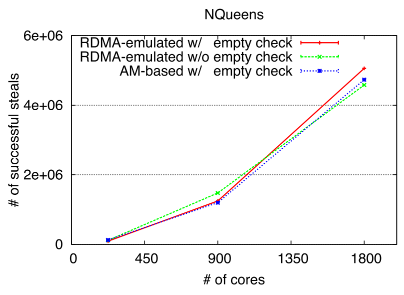
Fig. 14 Parallel performance of three benchmarks.



(a) Binary Task Creation

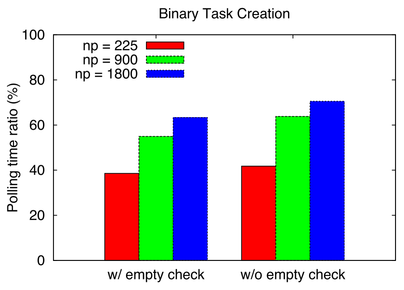


(b) Unbalanced Tree Search

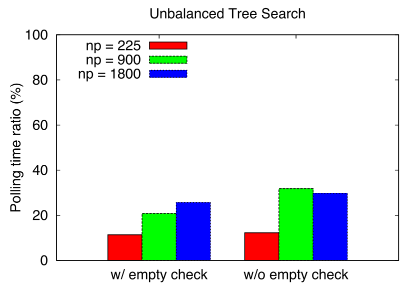


(c) NQueens

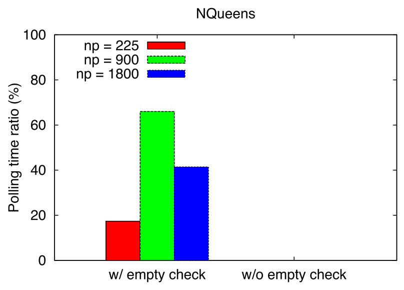
Fig. 15 The number of successful steals in three benchmarks.



(a) Binary Task Creation



(b) Unbalanced Tree Search



(c) NQueens

Fig. 16 The ratio of message handling time at task creation to total execution time.

ancing performance a little.

In order to dig into the performance results, we measured the number of successful steals and the ratio of message handling time to total execution time. **Figure 15** shows the number of successful steals in the experiment shown in Fig. 14. The number of successful steals is an index of how fast we can find tasks from remote processors. The results indicate that the number of successful steals have different trends from the load balancing performance, and that the speed of task discovery in RDMA-emulated and AM-based work stealing is not so different. Therefore, idleness of processors is not a crucial factor to the load balancing performance of our work stealing implementation in this experimental setup.

In AM-based work stealing, there is another important factor to degrade load balancing performance. Because a remote fetch-and-add in the remote task queue manipulation is handled by a victim worker, AM-based work stealing prevents task execution in victim workers and introduces an additional overhead to thread creation which internally calls an AM polling function. **Figure 16** shows the ratio of message handling time at task creation to total execution time. The results show that the message handling time of BTC, UTS, and NQueens occupies 63.3%, 25.7%, and 41.4% of the total execution time, respectively, in AM-based work stealing with empty check. In AM-based work stealing without empty check, the message handling time of BTC and UTS occupies 70% and 29.8% of the total execution time. This ratio confirms the performance degradation in BTC and UTS benchmarks with error rates of approximately 8%. For the results of NQueens, the error rate when using 225 cores is approximately 6%; the rate when using 900 and 1,800 cores varies 22% to 78% for unknown reasons.

8. Conclusion

In this paper, we presented uni-address, a scalable thread management technique for RDMA-based work stealing. The uni-address scheme solves scalability problems in applying an existing thread migration scheme, iso-address, to large-scale distributed memory environments. Iso-address consumes a huge amount of virtual address space proportional to the number of processing cores in each node, and therefore thread migration cannot be implemented with RDMA operations, which are important for scalable work stealing. Uni-address significantly reduces virtual memory usage for thread migration and enables RDMA-based work stealing.

We implemented uni-address threads, a lightweight multi-thread library supporting inter-node work stealing with uni-address scheme. The library is implemented in C++ and a few assembly codes, and therefore it can easily be integrated with existing application codes, libraries, and programming languages. We implemented the library on top of the GASNet communication library with two implementation strategies—RDMA-emulated and Active Messages (AM) based implementation to investigate the efficiency of one-sided work stealing with RDMA features.

We performed experiments to evaluate the performance and the efficiency of uni-address threads with microbenchmarks and three benchmarks: Binary Task Creation, Unbalanced Tree Search, and NQueens solver. Microbenchmark results indicate that the task creation takes 184 cycles on a x86-64 processor and the context switching takes about 1 K cycles. On the three benchmarks, uni-address threads worked with less than 136 KB virtual memory for thread migration in each processor and achieved more than 98% parallel efficiency with the RDMA-emulated implementation on 1,800 processing cores of the TSUBAME2.5 supercomputer. We also investigated the performance differences between RDMA-emulated and AM-based work stealing, and showed performance impacts from active message handling on inter-node work stealing.

Acknowledgments This work is partially supported by JST, CREST through its research project: "Highly Productive, High Performance Application Frameworks for Post Petascale Computing." The calculations were carried out on the TSUBAME2.5 supercomputer in the Tokyo Institute of Technology.

References

[1] Frigo, M., Leiserson, C.E. and Randall, K.H.: The implementation of the Cilk-5 multithreaded language, *Proc. ACM SIGPLAN 1998 conference on Programming language design and implementation*, *PLDI '98*, pp.212–223 (1998).
[2] Board, A.R.: OpenMP Application Program Interface version 3.0, Technical report (2008).
[3] Intel Corporation: *Intel® Threading Building Blocks reference manual* (2009).
[4] Nakashima, J. and Taura, K.: MassiveThreads: A Thread Library for High Productivity Languages, Agha, G., Igarashi, A., Kobayashi, N., Masuhara, H., Matsuoka, S., Shibayama, E. and Taura, K. (eds.), *Concurrent Objects and Beyond*, Lecture Notes in Computer Science, Vol.8665, pp.222–238 (2014).
[5] Wheeler, K., Murphy, R. and Thain, D.: Qthreads: An API for programming with millions of lightweight threads, *2008 IEEE International Symposium on Parallel and Distributed Processing*, *IPDPS '08*, pp.1–8 (2008).
[6] Lea, D.: A Java fork/join framework, *Proc. ACM 2000 conference on Java Grande*, *JAVA '00*, pp.36–43 (2000).
[7] Charles, P., Grothoff, C., Saraswat, V., Donawa, C., Kielstra, A., Ebcioglu, K., von Praun, C. and Sarkar, V.: X10: An object-oriented approach to non-uniform cluster computing, *Proc. 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, *OOPSLA '05*, pp.519–538 (2005).
[8] Callahan, D., Bradford, L. and Chamberlain, H.P.Z.: The Cascade High Productivity Language, *International Workshop on High-Level Programming Models and Supportive Environments*, pp.52–60 (2004).
[9] Blumofe, R.D. and Lisiecki, P.A.: Adaptive and Reliable Parallel Computing on Networks of Workstations, *Proc. Annual Conference on USENIX Annual Technical Conference*, *ATEC '97*, p.10 (1997).
[10] Van Nieuwpoort, R.V., Wrzesińska, G., Jacobs, C.J.H. and Bal, H.E.: Satin: A high-level and efficient grid programming model, *ACM Trans. Program. Lang. Syst.*, Vol.32, pp.9:1–9:39 (2010).
[11] Akiyama, S. and Taura, K.: Uni-Address Threads: Scalable Thread Management for RDMA-Based Work Stealing, *Proc. 24th International Symposium on High-Performance Parallel and Distributed Computing*, *HPDC '15*, pp.15–26 (2015).
[12] Antoniu, G., Bougé, L. and Namyst, R.: An Efficient and Transparent Thread Migration Scheme in the PM2 Runtime System, *Proc. 11 IPPS/SPDP'99 Workshops Held in Conjunction with the 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing*, pp.496–510 (1999).
[13] Fujitsu Co., Ltd.: FUJITSU Supercomputer PRIMEHPC FX10, available from ⟨http://www.fujitsu.com/global/products/computing/servers/supercomputer/primehpc-fx10/⟩ (accessed 2015-01-20).
[14] Olivier, S. and Prins, J.: Scalable Dynamic Load Balancing Using UPC, *37th International Conference on Parallel Processing, 2008, ICPP '08*, pp.123–131 (2008).
[15] Dinan, J., Larkins, D.B., Sadayappan, P., Krishnamoorthy, S. and Nieplocha, J.: Scalable work stealing, *Proc. Conference on High Performance Computing Networking, Storage and Analysis*, *SC '09*, pp.53:1–53:11 (2009).
[16] Saraswat, V.A., Kambadur, P., Kodali, S., Grove, D. and Krishnamoorthy, S.: Lifeline-based global load balancing, *Proc. 16th ACM symposium on Principles and practice of parallel programming*, *PPoPP '11*, pp.201–212 (2011).
[17] Bonachea, D.: GASNet Specification, v1.1, Technical report, Berkeley, CA, USA (2002).
[18] von Eicken, T., Culler, D.E., Goldstein, S.C. and Schauser, K.E.: Active Messages: A Mechanism for Integrated Communication and Computation, *Proc. 19th Annual International Symposium on Computer Architecture*, *ISCA '92*, pp.256–266, ACM (1992).
[19] Dinan, J., Krishnamoorthy, S., Larkins, D.B., Nieplocha, J. and Sadayappan, P.: Scioto: A Framework for Global-View Task Parallelism, *Proc. 2008 37th International Conference on Parallel Processing*, *ICPP '08*, Washington, DC, USA, pp.586–593, IEEE Computer Society (2008).
[20] Zhang, W., Tardieu, O., Grove, D., Herta, B., Kamada, T., Saraswat, V.A. and Takeuchi, M.: GLB: Lifeline-based Global Load Balancing library in X10, available from ⟨http://arxiv.org/abs/1312.5691⟩ (accessed 2013).
[21] YarKhan, A.: Dynamic Task Execution on Shared and Distributed Memory Architectures, PhD Thesis, University of Tennessee (2012).
[22] Hiraishi, T., Yasugi, M., Umatani, S. and Yuasa, T.: Backtracking-based load balancing, *Proc. 14th ACM SIGPLAN symposium on Principles and practice of parallel programming, PPoPP '09*, pp.55–64 (2009).
[23] Min, S.-J., Iancu, C. and Yelick, K.: Hierarchical work stealing on manycore clusters, *5th Conference on Partitioned Global Address Space Programming Models, PGAS '11* (2011).
[24] Nelson, J., Holt, B., Myers, B., Briggs, P., Ceze, L., Kahan, S. and Oskin, M.: Grappa: A Latency-Tolerant Runtime for Large-Scale Irregular Applications, *International Workshop on Rack-Scale Computing* (*WRSC w/EuroSys*) (2014).
[25] Wagner, D.B. and Calder, B.G.: Leapfrogging: A Portable Technique for Implementing Efficient Futures, *Proc. 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '93*, pp.208–217 (1993).
[26] Mohr, E., Kranz, D.A. and Halstead, Jr., R.H.: Lazy Task Creation: A Technique for Increasing the Granularity of Parallel Programs, *IEEE Trans. Parallel Distrib. Syst.*, Vol.2, pp.264–280 (1991).
[27] Blumofe, R.D. and Leiserson, C.E.: Scheduling Multithreaded Computations by Work Stealing, *J. ACM*, Vol.46, No.5, pp.720–748 (online), DOI: 10.1145/324133.324234 (1999).
[28] Acar, U.A., Blelloch, G.E. and Blumofe, R.D.: The data locality of work stealing, *Proc. 12th annual ACM symposium on Parallel algorithms and architectures, SPAA '00*, pp.1–12 (2000).
[29] Huang, C., Zheng, G., Kalé, L. and Kumar, S.: Performance Evaluation of Adaptive MPI, *Proc. 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '06*, pp.12–21 (2006).
[30] Kale, L. and Lifflander, J.: Controlling Concurrency and Expressing Synchronization in Charm++ Programs, Agha, G., Igarashi, A., Kobayashi, N., Masuhara, H., Matsuoka, S., Shibayama, E. and Taura, K. (eds.), *Concurrent Objects and Beyond*, Lecture Notes in Computer Science, Vol.8665, pp.196–221 (2014).
[31] Dinan, J., Olivier, S., Sabin, G., Prins, J., Sadayappan, P. and Tseng, C.-W.: Dynamic Load Balancing of Unbalanced Computations Using Message Passing, *IEEE International Parallel and Distributed Processing Symposium, 2007, IPDPS 2007*, pp.1–8 (2007).
[32] Nieplocha, J., Tipparaju, V., Krishnan, M. and Panda, D.K.: High Performance Remote Memory Access Communication: The Armci Approach, *Int. J. High Perform. Comput. Appl.*, Vol.20, pp.233–253 (2006).
[33] Lamport, L.: A New Solution of Dijkstra's Concurrent Programming

Problem, *Commun. ACM*, Vol.17, No.8, pp.453–455 (1974).
[34] TOP500.org: TOP500 Supercomputer Site, available from ⟨http://www.top500.org⟩ (accessed 2015-01-20).
[35] Olivier, S., Huan, J., Liu, J., Prins, J., Dinan, J., Sadayappan, P. and Tseng, C.-W.: UTS: An unbalanced tree search benchmark, *Proc. 19th international conference on Languages and compilers for parallel computing, LCPC'06*, pp.235–250 (2007).
[36] Duran, A., Teruel, X., Ferrer, R., Martorell, X. and Ayguade, E.: Barcelona OpenMP Tasks Suite: A Set of Benchmarks Targeting the Exploitation of Task Parallelism in OpenMP, *Proc. 2009 38th International Conference on Parallel Processing, ICPP '09*, pp.124–131 (2009).
[37] Intel Corporation: Intel® Cilk^{TM} Plus. available from ⟨http://www.cilkplus.org/⟩.

Appendix

A.1 x86-64 Assembly to Save Context

```
/*
typedef struct {
    void *rip, *rsp, *rbp, *rbx, *r12, *r13, *r14, *r15;
    context *parent;
} context_t;

typedef void (*context_func_t)(context_t *ctx,void *arg);
void save_context_and_call(context_t *parent,
                           context_func_t f, void *arg);
*/
save_context_and_call:
    push    %rdi        /* save parent context */
    push    %r15,%r14   /* save callee-saved regs */
    push    %r13,%r12,%rbx,%rbp
    lea     -16(%rsp), %rax  /* save current SP */
    push    %rax
    lea     1f(%rip), %rax   /* save IP for resume */
    push    %rax
    /* call a thread start function */
    mov     %rsi, %rax  /* function f */
    mov     %rsp, %rdi  /* argument ctx */
    mov     %rdx, %rsi  /* argument arg */
    call    *%rax
    add     $8, %rsp    /* pop IP */
1:  /* here, jumped from resume_context */
    add     $8, %rsp    /* pop SP */
    pop     %rbp,%rbx   /* restore callee-saved regs */
    pop     %r12,%r13,%r14,%r15
    add     $8, %rsp    /* pop parent context */
    ret

/* void resume_context(context_t *ctx); */
resume_context:
    mov %rdi, %rsp      /* restore SP (== ctx) */
    ret                 /* pop IP and restore it */
```

Shigeki Akiyama is a Ph.D. candidate at the Department of Information and Communication Engineering, the University of Tokyo. He was born in 1988, and received his B.S. from Osaka University in 2010, and his M.S. degree from the University of Tokyo in 2012. His research interest is parallel and distributed computing and programming languages.

Kenjiro Taura is a professor at the Department of Information and Communication Engineering, the University of Tokyo. He was born in 1969, and received his B.S., M.S., and D.S.c degrees from the University of Tokyo in 1992, 1994, and 1997. His major research interests include parallel/distributed computing and programming languages. He is a member of ACM and IEEE.