Uni-Address Threads: Scalable Thread Management for RDMA-Based Work Stealing

Shigeki Akiyama
The University of Tokyo
7-3-1 Hongo Bunkyo-ku
Tokyo, Japan
akiyama@eidos.ic.i.u-tokyo.ac.jp

Kenjiro Taura
The University of Tokyo
7-3-1 Hongo Bunkyo-ku
Tokyo, Japan
tau@eidos.ic.i.u-tokyo.ac.jp

ABSTRACT

Task-parallel systems have been widely used to parallelize programs. They provide automatic load balancing and programmers can easily parallelize sequential programs, including irregular ones, without considering task placement to physical processors.

Despite the success of shared memory task parallelism, task parallelism on large-scale distributed memory environments is still challenging. 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 64bit 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 this paper, we propose *uni-address*, a new thread management scheme for distributed memory task parallelism. It significantly reduces virtual memory usage for thread migration and enables us to implement RDMA-based work stealing. We implement a lightweight multithread library supporting RDMA-based work stealing based on the uni-address scheme, and demonstrate its lightweight thread operations and scalable work stealing on Fujitsu FX10 supercomputing system with three benchmarks: Binary Task Creation, Unbalanced Tree Search, and NQueens solver. As a result, we confirmed all the benchmarks works with less than 144KB virtual memory for thread migration in each processor and achieved more than 95% parallel efficiency on 3840 processing cores, relative to the results on 480 processing cores.

HPDC'15, June 15–20, 2015, Portland, Oregon, USA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-3550-8/15/06 ...$15.00.
http://dx.doi.org/10.1145/2749246.2749272 .

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel Programming*

General Terms

Algorithms, Design, Performance

Keywords

Task parallelism; lightweight multithreading; thread migration; distributed work stealing; remote direct memory access

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 [11, 5, 16, 21, 28, 18]. On large-scale distributed memory machines, 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 [6, 7]; 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 [4, 26], which renders them difficult to reuse across multiple languages. The situation contrasts with shared memory machines, where we have task-parallel *libraries* [16, 28, 21], 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. We propose a new implementation scheme, *uni-address*, which overcomes a scalability limitation of a previously proposed *iso-address* [2] scheme. We implemented the library, *uni-address threads*, on Fujitsu FX10 system [12] and tested its scalability up to 3840 cores.

Novel contributions of this paper are as follows:

- We propose a new native thread migration scheme, called *uni-address*, which significantly reduces the usage of virtual address space.
- Based on this technique, we design and implement a work stealing scheduler with *one-sided* task stealing, which can take advantage of Remote Direct Memory Access (RDMA).
- We evaluated virtual memory usage and performance of uni-address threads on 3840 processing cores in three benchmark programs: Binary Task Creation, Unbalanced Tree Search, and NQueens. We confirmed all benchmarks works with less than 144KB virtual memory for thread migration in each processor and achieved more than 95% parallel efficiency on 3840 processing cores, relative to the results on 480 processing cores.

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 iso-address thread migration scheme and its scalability limitations. In Section 5, we describe uni-address scheme and presents the implementation of RDMA-based work stealing and inter-task synchronization on top of it. Section 6 shows an experimental evaluation of uni-address scheme, and Section 7 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. Implementation strategies and complexities are heavily affected by synchronization patterns supported by the system.

Bag of tasks.

Some systems such as Scioto [8, 9] and X10/GLB [30] support only 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. Bag of tasks are 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. Bag of tasks is clearly very restrictive and cannot express many important divide-and-conquer algorithms naturally.

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 [4, 29]. We say such systems support "atomic tasks" model in the rest of the discussion.

Atomic tasks admit an implementation strategy similar to that for bag of tasks, with only the 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.

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 [11], OpenMP tasks [5], TBB [16], MassiveThreads [21], Qthreads [28], Java fork-join [18]), but scarce on distributed memory environments; notable exceptions are Satin [26], HotSLAW [19], and Grappa [22]. 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 balancing 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.

Implementing fork-join with tied tasks.

To avoid 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 an yet-to-be-started task can be simply represented by

```
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(x, 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 - 1);
    sync;
    return r0 + r1;
  }
}
```

Figure 1: Fibonacci in atomic tasks model (left [4]); and in fork-join model (right [11])

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 [16, 18, 26, 19]; 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 [27], is particularly attractive when implemented in high-level languages, e.g., Java, that do not support non-local jumps.

"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 [20] and adopted in Cilk [11] and other systems [21, 28], 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 [3] applies only when any task, started or not, can be stolen by any idle worker. A bound on extra cache misses [1] 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 environment, 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 [11, 21]. 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 [14] and Charm++ [17]. Adaptive MPI uses iso-address to migrate MPI processes for dynamic load balancing on distributed memory systems, and Charm++ uses iso-address to support migratable threads as threaded entry methods for concurrent objects.

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

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

Figure 2: Fork-join task API

Other Task-Parallel Systems.

Tascell [13] is a "logical thread"-free task-parallel framework based on backtracking-based approach for shared and distributed memory environments. Tascell supports a fork-join model and genuine task migration by both compiler support and explicit packing of task's continuation, similar to atomic tasks, in language level.

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 parallelism of computational resources, a programmer has to spawn new tasks. Our model provides fork-join primitives for creating a task and waiting for completion of a task, shown in Figure 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 migrate 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.

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 [2] 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 continues 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 [2], 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 iso-address scheme is that it just works with languages with ambiguous pointers and their ordinary compilers unaware of migration. Also, as the simple bit-wise 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) [3]; thus, with concurrency of largest machines already surpassing three million [25], 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 16KB (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 21K 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 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 [11], and has been proved important in distributed memory environments [9].

   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 (16KB) 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., 4KB) 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 iso-address scheme maintains the validity of intra-stack pointers by copying a stack into the same address upon migration. The key idea behind uni-address scheme is that, in order to maintain 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 made 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 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, 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 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 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 that all workers can allocate *the* uni-address region at the same virtual address. In practice, it means we need to

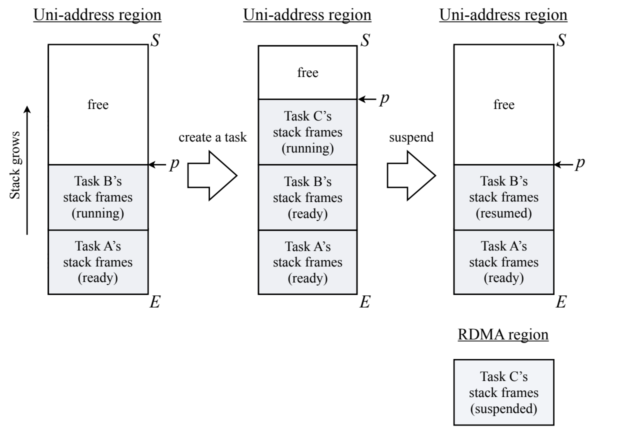

Uni-address region
S
free
Stack grows
Task B's stack frames (running)
Task A's stack frames (ready)
p
E
create a task
Task C's stack frames (running)
Task B's stack frames (ready)
suspend
Task B's stack frames (resumed)
RDMA region
Task C's stack frames (suspended)

Figure 3: Uni-address region.

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 (Figure 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

```
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 = ctx->parent_ctx->rsp + 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 void create_thread(thread_func_t f, void *arg) {
18     context *parent = current_worker()->parent;
19     save_context_and_call(parent, do_create_thread,
20                           f, arg);
21     current_worker()->parent = parent;
22 }
```

Figure 4: Optimized implementation of task creation function.

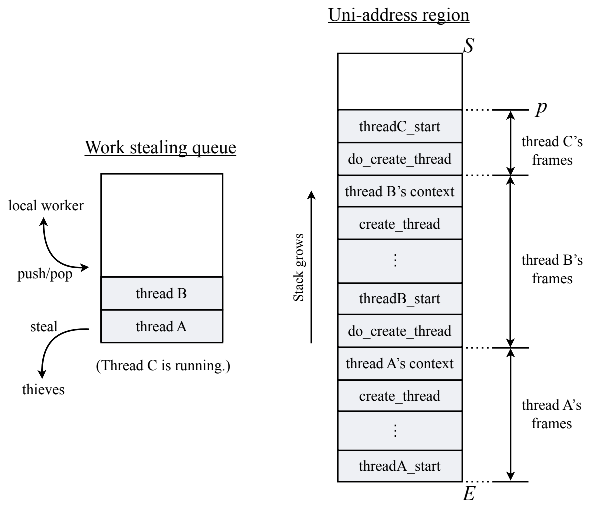


Figure 5: A work stealing queue and the corresponding uni-address region.

an invariant that the running task occupies the lowest addresses of the used region.

4. When a task is suspended, its task is copied out from the uni-address region into any free address in the RDMA region, and resume the task just above it 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 Figure 5 illustrates a work stealing queue and the corresponding uni-address region. The thread 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. 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 thread 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, naive locking scheme for mutual exclusion does not scale well especially on large-scale distributed memory machines [9]. To address this issue, we implement THE protocol [11] with RDMA READ, WRITE, and fetch-and-add. THE protocol is used in several task-parallel systems on shared memory machines, such as Cilk [11] and MassiveThreads [21], and because it eliminates locking from local accesses to a task queue, it reduces tasking overhead and improves 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 failed, 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 thread in the stolen task queue entry. In thread migration, we first calculate the remote address of the stack region for RDMA operation, 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.

```
1   void resume_remote_context(saved_context_t *sctx,
            taskq_entry e) {
2       WAIT_QUEUE_PUSH(sctx);
3       void *remote_base = get_remote_base(e.stack_base,
            victim);
4       RDMA_GET(e.stack_base, remote_base, e.stack_size,
            victim);
5       taskq_unlock(q, victim);
6       resume_context(e.ctx);
7   }
8   void steal() {
9       int victim = select_victim_randomly();
10      taskq *q = get_remote_taskq(victim);
11      if (!try_lock(q, victim))
12          return;
13
14      taskq_entry e;
15      if (!taskq_steal(q, victim, &e)) {
16          taskq_unlock(q, victim);
17          return;
18      }
19      suspend(resume_remote_context, e);
20  }
```

Figure 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  }
```

Figure 7: Implementation of join function

5.4 Inter-Thread Synchronization

In this section, we describe an implementation of inter-thread 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,
                  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  }
```

Figure 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. EXPERIMENTAL RESULTS

This section evaluates the efficiency and performance of uni-address threads. We conducted experiments on a Fujitsu PRIMEHPC FX10 supercomputing system and a single

Fujitsu PRIMEHPC FX10 system	
CPU	SPARC64IXfx 1.848GHz, 16 cores
Memory	32GB/node
Interconnect	Custom interconnect (Tofu)
OS	XTCOS (GNU/Linux 2.6.25.8 based)
Compiler	GCC 4.6.1 (option -O3)
MPI	Fujitsu MPI Library 1.2.1
a single node x86-64 server	
CPU	Xeon E5-2660 2.2GHz * 2, total 16 cores
Memory	64GB/node
OS	Debian 6.0.4 (GNU/Linux 2.6.32.5)
Compiler	GCC 4.9.1 (option -O3)

Table 1: Experimental setup.

node Xeon server. Table 1 shows the hardware and software configuration in our experiments. We used up to 256 nodes for the experiments.

Although FX10 system provides RDMA READ and WRITE operations as Fujitsu RDMA interface, RDMA fetch-and-add is not provided. Therefore, we implemented a software implementation of remote fetch-and-add operation. To simulate hardware remote fetch-and-add operation, the fetch-and-add implementation reserves a processing core within a node in advance and use it as a communication server handling fetch-and-add requests from other nodes. The fetch-and-add requests are sent with "RDMA Write with remote notice" operation, which is an RDMA WRITE operation that notifies the target node of the completion of the operation. Because there is a communication server within a node, our experiments use only 15 cores within a node for computation. Figure 9 shows the communication latencies of RDMA READ/WRITE operations in FX10 system. The latency of the software-based remote fetch-and-add operation is 9.8K cycles on average.

For comparison, we used two existing lightweight multithread frameworks—MassiveThreads and MIT Cilk. MassiveThreads is a lightweight multithread library written in C, which can be extended to support inter-node load balancing with iso-address thread migration. Cilk is a lightweight multithread framework implemented with source code processing specialized for a fork-join model. These frameworks support a child-first work stealing scheduler similar to uni-address threads. In our experiments, we used MassiveThreads 0.95 and MIT Cilk 5.4.6.

Confidence intervals in the following figures are calculated with 95% confidence level.

6.1 Benchmark Programs

To evaluate 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. It has two parameters *depth* and *iter*. *Depth* means the depth of a generated task tree, and each task repeats, *iter* times, spawning two child tasks and waiting for their completions. When *iter* ≥ 2, parallelism rapidly grows and shrinks during execution; therefore, it requires high load balancing performance.

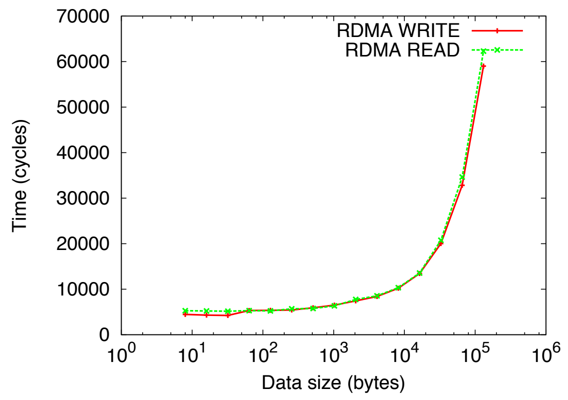


Figure 9: RDMA READ/WRITE latencies of FX10 system.

	SPARC64IXfx	Xeon E5-2660
Uni-address threads	413 cycles	100 cycles
MassiveThreads	658 cycles	110 cycles
Cilk	47 cycles	59 cycles

Table 2: Thread creation overhead.

UTS Unbalanced Tree Search benchmark [23] is a benchmark to evaluate performance of dynamic load balancing algorithms and implementations. UTS benchmark traverses an unpredictable, tree-based state space generated by a probability distribution. The detailed description of parameters of UTS benchmark are in [23]. 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 = 17 and 18. The command-line arguments is "`-t 1 -r 0 -b 4 -a 3 -d {17,18}`".

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 [10].

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 [15] performs such an optimization for its `cilk_for` statement.

6.2 Task Creation Overhead

We measured the overhead of a task creation in uni-address threads on a SPARC64 IXfx processor and a Xeon E5-2660 processor. For comparison, we also measured the overhead of task creation in MassiveThreads and MIT Cilk.

Table 2 shows the results. The task creation overhead of uni-address threads is 413 cycles and 100 cycles on average on the SPARC64IXfx processor and the Xeon E5-2660 processor, respectively. Here, we can see that uni-address threads achieved a comparable performance to an

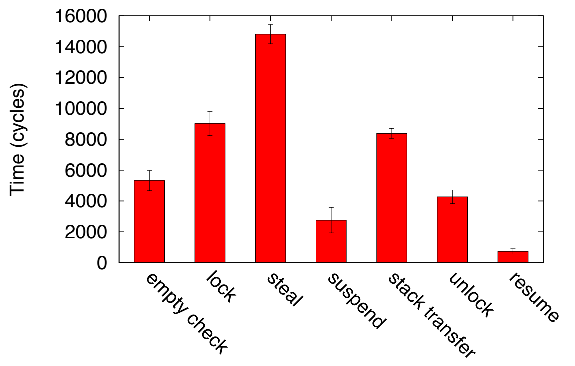


Figure 10: Breakdown of work stealing time

existing lightweight multithread library, MassiveThreads, on both of the processors. For reference, task creation of uni-address threads takes 8.8x and 1.7x more time than Cilk in SPARC64IXfx and Xeon E5-2660.

6.3 Work Stealing Overhead

We measured the overhead of work stealing in uni-address threads on the FX10 system. 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 3055 bytes. To eliminate effects of performance noises, we removed the time of the first steal from the results.

Figure 10 shows the execution time breakdown of inter-node work stealing, and Table 3 shows the operations constituting work stealing. A work stealing takes 42K cycles in total, and suspend and resume operations, which is the main source of overhead of uni-address scheme, take 3.5K cycles, or 7.7% of the total work stealing time. The other overheads are mostly from RDMA operations.

Here, we attempt to compare uni-address and iso-address scheme. As described in Section 4, iso-address scheme frequently causes page faults at thread migration, which takes 21K cycles in FX10 system; therefore, we can estimate that work stealing time of uni-address threads is approximately 71% of iso-address scheme, assuming the stack transfer time are comparable in both schemes [2].

6.4 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, execution time, and stack memory usage in the uni-address region on 3840 cores. Note that all benchmarks worked with less than 144KB virtual memory.

Figure 11 shows the parallel performance of uni-address threads with the three benchmark programs. In all benchmarks, the parallel performance is reported as the total throughput of processed tasks or nodes per second.

In summary, all benchmarks scale well in large problems. In Figure 11(a), BTC benchmark (*iter* $= 1$) scales well to 3840 cores. The throughput on 3840 cores is 16.7 and 16.5 billion tasks per second for *depth* $= 38$ and 39, respectively, and the efficiencies are 97% and 98%, respectively, compared to the performance on 480 cores. In Figure 11(b), the throughput of BTC benchmark (*iter* $= 2$) on 3840 cores is 11.1 and 16.6 billion tasks per second for *depth* $= 19$ and 20, respectively, and the efficiencies are 97% and 98%, respectively, compared to the performance on 480 cores.

In Figure 11(c), UTS benchmark scales well to 3840 cores. The throughput on 3840 cores is 1.53 and 1.55 billion nodes per second for *depth* $= 17$ and 18, respectively, and the efficiencies are 97% and 99%, respectively, compared to the performance on 480 cores. These results are comparable to existing dynamic load balancing frameworks for distributed memory supercomputers [9, 24] when normalizing their clock frequencies and integer performance.

In Figure 11(d), NQueens benchmark scales well to 3840 cores in the case $N = 18$. The throughput on 3840 cores is 168 and 187 million nodes per second, and the performance achieves 78% and 95% efficiency relative to 480 cores in the case $N = 17$ and 18.

7. CONCLUSION

In this paper, we presented uni-address, a scalable thread management technique for RDMA-based work stealing. Uni-address scheme solves scalability problems in applying an existing thread migration scheme, iso-address, to large-scale distributed memory supercomputers. 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 multithread library supporting distributed 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 evaluated the performance and efficiency of uni-address threads with microbenchmarks and three benchmarks: Binary Task Creation, Unbalanced Tree Search, and NQueens. Microbenchmark results indicate that the task creation takes 413 cycles on a SPARC64 processor and 100 cycles on an x86-64 processor, and the context switching takes 3.5K cycles, 7% of the total work stealing time on a Fujitsu PRIMEHPC FX10 system. On the three benchmarks, uni-address threads worked with less than 144KB virtual memory for thread migration in each processor and achieved more than 95% parallel efficiency with 3840 processing cores on the FX10 system.

8. ACKNOWLEDGMENTS

This project was partially supported by JST, CREST through its research program: “Highly Productive, High Performance Application Frameworks for Post Petascale Computing.” In addition, this research was conducted using the Fujitsu PRIMEHPC FX10 System (Oakleaf-FX,Oakbridge-FX) in the Information Technology Center, The University of Tokyo.

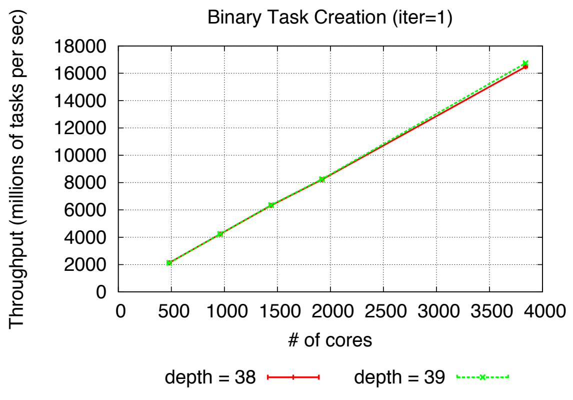
[2]Actually, stack transfer in iso-address scheme requires assistance of a remote node, so it takes longer time than stack transfer in uni-addrss scheme, which is performed by an RDMA READ operation.

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.

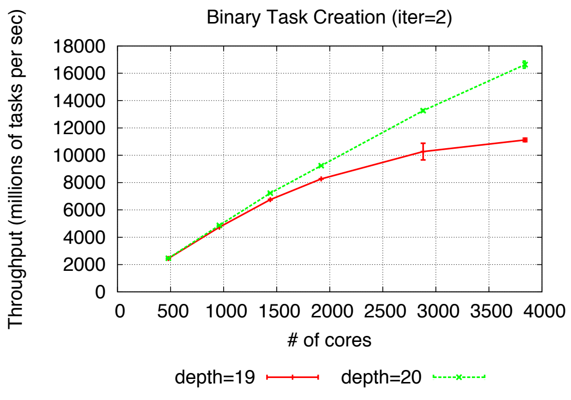
Table 3: Operations consisting of work stealing.

Benchmark	Parameters	Total tasks or nodes	Time	Stack usage
Binary Task Creation (*iter* = 1)	*depth* = 38	550 billion tasks	65.67 sec	43,568 bytes
	depth = 39	1,099 billion tasks	33.37 sec	44,688 bytes
Binary Task Creation (*iter* = 2)	*depth* = 19	367 billion tasks	32.96 sec	22,288 bytes
	depth = 20	1,466 billion tasks	88.14 sec	23,408 bytes
Unbalanced Tree Search	*depth* = 17	110 billion nodes	71.62 sec	139,536 bytes
	depth = 18	439 billion nodes	282.2 sec	147,392 bytes
NQueens	*N* = 17	8 billion nodes	47.60 sec	74,272 bytes
	N = 18	59 billion nodes	317.8 sec	79,120 bytes

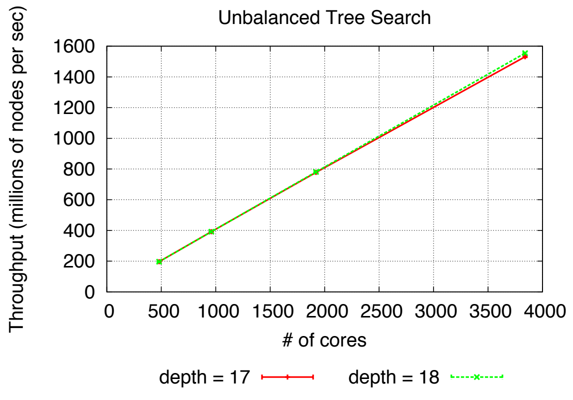
Table 4: The number of generated tasks or nodes in three benchmark. *Time* is average execution time on 3840 cores. *Stack usage* is maximum usage of the uni-address region.



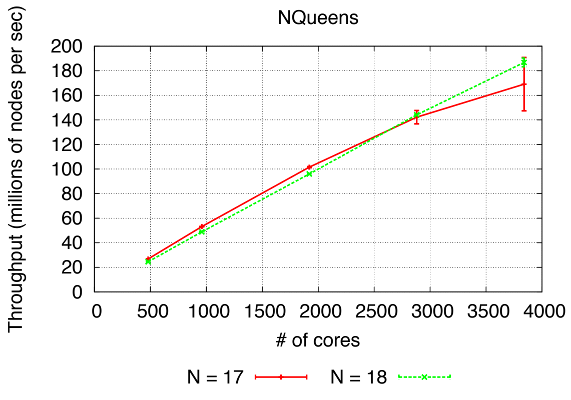
(a) Binary Task Creation (iter=1)



(b) Binary Task Creation (iter=2)



(c) Unbalanced Tree Search



(d) NQueens

Figure 11: Parallel performance in three benchmarks.

9. REFERENCES

[1] U. A. Acar, G. E. Blelloch, and R. D. Blumofe. The data locality of work stealing. In *Proceedings of the twelfth annual ACM symposium on Parallel algorithms and architectures*, SPAA '00, pages 1–12, 2000.

[2] G. Antoniu, L. Bougé, and R. Namyst. An efficient and transparent thread migration scheme in the pm2 runtime system. In *Proceedings of the 11 IPPS/SPDP'99 Workshops Held in Conjunction with the 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing*, pages 496–510, 1999.

[3] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, Sept. 1999.

[4] R. D. Blumofe and P. A. Lisiecki. Adaptive and reliable parallel computing on networks of workstations. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '97, pages 10–10, 1997.

[5] A. R. Board. OpenMP application program interface version 3.0. Technical report, May 2008.

[6] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '05, pages 519–538, 2005.

[7] H. P. Z. David Callahan, Bradford L. Chamberlain. The cascade high productivity language. *International Workshop on High-Level Programming Models and Supportive Environments*, 0:52–60, 2004.

[8] J. Dinan, S. Krishnamoorthy, D. B. Larkins, J. Nieplocha, and P. Sadayappan. Scioto: A framework for global-view task parallelism. In *Proceedings of the 2008 37th International Conference on Parallel Processing*, ICPP '08, pages 586–593, Washington, DC, USA, 2008. IEEE Computer Society.

[9] J. Dinan, D. B. Larkins, P. Sadayappan, S. Krishnamoorthy, and J. Nieplocha. Scalable work stealing. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 53:1–53:11, 2009.

[10] A. Duran, X. Teruel, R. Ferrer, X. Martorell, and E. Ayguade. Barcelona OpenMP Tasks Suite: A set of benchmarks targeting the exploitation of task parallelism in OpenMP. In *Proceedings of the 2009 38th International Conference on Parallel Processing*, ICPP '09, pages 124–131, Sept 2009.

[11] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the Cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, PLDI '98, pages 212–223, 1998.

[12] Fujitsu Co., Ltd. Fujitsu supercomputer PRIMEHPC FX10. `http://www.fujitsu.com/global/products/computing/servers/supercomputer/primehpc-fx10/`. [Online; accessed 20-January-2015].

[13] 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*, PPoPP '09, pages 55–64, 2009.

[14] C. Huang, G. Zheng, L. Kalé, and S. Kumar. Performance evaluation of Adaptive MPI. In *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '06, pages 12–21, 2006.

[15] Intel Corporation. Intel® Cilk^{TM} Plus. `http://www.cilkplus.org/`.

[16] Intel Corporation. *Intel® Threading Building Blocks reference manual*, 2009.

[17] L. Kale and J. Lifflander. Controlling concurrency and expressing synchronization in Charm++ programs. In G. Agha, A. Igarashi, N. Kobayashi, H. Masuhara, S. Matsuoka, E. Shibayama, and K. Taura, editors, *Concurrent Objects and Beyond*, volume 8665 of *Lecture Notes in Computer Science*, pages 196–221. 2014.

[18] D. Lea. A Java fork/join framework. In *Proceedings of the ACM 2000 conference on Java Grande*, JAVA '00, pages 36–43, 2000.

[19] S.-J. Min, C. Iancu, and K. Yelick. Hierarchical work stealing on manycore clusters. In *In Fifth Conference on Partitioned Global Address Space Programming Models*, PGAS '11, 2011.

[20] E. Mohr, D. A. Kranz, and R. H. Halstead, Jr. Lazy task creation: A technique for increasing the granularity of parallel programs. *IEEE Trans. Parallel Distrib. Syst.*, 2:264–280, July 1991.

[21] J. Nakashima and K. Taura. MassiveThreads: A thread library for high productivity languages. In G. Agha, A. Igarashi, N. Kobayashi, H. Masuhara, S. Matsuoka, E. Shibayama, and K. Taura, editors, *Concurrent Objects and Beyond*, volume 8665 of *Lecture Notes in Computer Science*, pages 222–238. 2014.

[22] J. Nelson, B. Holt, B. Myers, P. Briggs, L. Ceze, S. Kahan, and M. Oskin. Grappa: A latency-tolerant runtime for large-scale irregular applications. In *International Workshop on Rack-Scale Computing (WRSC w/EuroSys)*, April 2014.

[23] 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*, LCPC'06, pages 235–250, 2007.

[24] V. A. Saraswat, P. Kambadur, S. Kodali, D. Grove, and S. Krishnamoorthy. Lifeline-based global load balancing. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, PPoPP '11, pages 201–212, 2011.

[25] TOP500.org. TOP500 supercomputer site. `http://www.top500.org`. [Online; accessed 20-January-2015].

[26] R. V. Van Nieuwpoort, G. Wrzesińska, C. J. H. Jacobs, and H. E. Bal. Satin: A high-level and efficient grid programming model. *ACM Trans. Program. Lang. Syst.*, 32:9:1–9:39, March 2010.

[27] 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, PPoPP '93, pages 208–217, 1993.

[28] K. Wheeler, R. Murphy, and D. Thain. Qthreads: An API for programming with millions of lightweight threads. In *2008 IEEE International Symposium on Parallel and Distributed Processing*, IPDPS '08, pages 1–8, April 2008.

[29] A. YarKhan. *Dynamic Task Execution on Shared and Distributed Memory Architectures*. PhD thesis, University of Tennessee, 2012.

[30] W. Zhang, O. Tardieu, D. Grove, B. Herta, T. Kamada, V. A. Saraswat, and M. Takeuchi. GLB: lifeline-based global load balancing library in X10. http://arxiv.org/abs/1312.5691, Dec. 2013.

APPENDIX

A. 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 */
```