Itoyori: Reconciling Global Address Space and Global Fork-Join Task Parallelism

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

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

ABSTRACT

This paper introduces Itoyori, a task-parallel runtime system designed to tackle the challenge of scaling task parallelism (more specifically, nested fork-join parallelism) beyond a single node. The partitioned global address space (PGAS) model is often employed in task-parallel systems, but naively combining them can lead to poor performance due to fine-grained and redundant remote memory accesses. Itoyori addresses this issue by automatically caching global memory accesses at runtime, enabling efficient cache sharing among parallel tasks running on the same processor. As a real-world case study, we ported an existing task-parallel implementation of the Fast Multipole Method (FMM) to distributed memory with Itoyori and achieved a 7.5× speedup when scaled from a single node to 12 nodes and up to 6.0× faster performance than without caching. This study demonstrates that global-view fork-join programming can be made practical and scalable, while requiring minimal changes to the shared-memory code.

CCS CONCEPTS

• **Computing methodologies** → **Distributed computing methodologies**; **Parallel computing methodologies**.

KEYWORDS

PGAS, task parallelism, fork-join, work stealing, cache coherence

ACM Reference Format:
Shumpei Shiina and Kenjiro Taura. 2023. Itoyori: Reconciling Global Address Space and Global Fork-Join Task Parallelism. In *The International Conference for High Performance Computing, Networking, Storage and Analysis (SC '23), November 12–17, 2023, Denver, CO, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3581784.3607049

1 INTRODUCTION

In order to effectively handle dynamic and irregular parallelism, parallel runtime systems have evolved over time to accommodate task parallelism, more specifically, nested fork-join parallelism. Fork-join parallelism enables the dynamic creation and arbitrary nesting of parallel tasks, facilitating the clear and succinct representation of dynamic and irregular parallel algorithms. The runtime task scheduler, such as work stealing [16], takes the responsibility of assigning parallel tasks to processor cores, allowing programmers to concentrate on expressing the inherent parallelism of algorithms without needing to consider the underlying hardware details. Its well-structured, compositional parallel primitives align well with recursive fine-grained parallelism and yield good analytical properties [1, 16]. Runtime systems such as Cilk [15, 31], OpenCilk [62], oneTBB (formarly Intel TBB [61]), and OpenMP [6] support fork-join parallelism; however, most of them are designed for shared-memory programming. Scaling fork-join programs from a single node to distributed-memory clusters remains a challenge.

The challenge in distributed-memory fork-join parallelism is two-fold: inter-node dynamic load balancing and remote memory access. Inter-node dynamic load balancing, such as distributed work stealing, has been intensively researched [3, 4, 20, 26, 27, 33, 50, 55, 65], with reported scalability reaching up to thousands of nodes [65]. Given that the nodes executing the tasks are determined at runtime, it is natural to adopt a unified, global view of distributed memory. This concept, known as a global address space, enables all tasks to perceive the same global memory view, irrespective of the specific nodes on which they are executed.

Thus far, researchers have investigated the integration of a global address space and inter-node dynamic load balancing. An early attempt involves combining fork-join parallelism and distributed shared memory (DSM) [13, 14], which enables transparent access to the global virtual address space. DSM systems typically provide a software cache for remote memory access, and cache coherence actions are performed by trapping memory protection faults for transparency. However, DSM systems have generally not gained widespread acceptance, likely due to their performance penalty resulting from their too strict constraints. Instead, the partitioned global address space (PGAS) model [21–23, 28, 44] has emerged, offering programmers increased programmability to optimize performance. In contrast to DSM systems, PGAS systems necessitate the use of explicit APIs for global memory access in order to distinguish it from local memory access. The majority of existing PGAS systems are designed for the Single Program Multiple Data (SPMD) model, wherein the programmer is responsible of mapping computations to nodes. To the best of our knowledge, only a few PGAS systems [50, 55] have been designed for the *global* fork-join model, in which tasks are automatically load balanced by the runtime system across node boundaries.

It is challenging, however, to reconcile the PGAS model and the global fork-join model. The PGAS model is about distinguishing between local and global data to optimize data movement, while the very point of the global fork-join model is that tasks can move across nodes for load balancing. For instance, even if two tasks that access the same data are likely to be executed on the same node,

SC '23, November 12–17, 2023, Denver, CO, USA

ACM ISBN 979-8-4007-0109-2/23/11.
https://doi.org/10.1145/3581784.3607049

aggregating communication for them is difficult for programmers because they could potentially run on different nodes. Consequently, each task communicates independently for the data it uses, resulting in fine-grained and redundant communication.

A viable solution to this issue is to incorporate a software cache within the PGAS runtime. When a processor executes a set of parallel tasks that access adjacent or overlapping memory regions, their memory accesses are likely to be cached in the local memory, thereby reducing redundant communication. We consider this approach is effective because (1) most tasks usually do not migrate when scheduled by work stealing if parallelism is sufficient [16, 31, 51], and (2) tasks that are close in the computation graph often access the same data [1, 12]. This approach can be seen as a compromise between DSM and PGAS, as it employs a software cache while still requiring explicit APIs, although this idea is not novel. For example, PGAS systems such as MuPC [75], Chapel [30], CLaMPI [25], GAM [19], and Falcon [73] have implemented a software cache, albeit not in the context of fork-join parallelism.

To demonstrate that the fork-join model can be effective even on distributed memory with the help of software caching, we developed a new runtime system *Itoyori*[1]. We designed Itoyori to offer a simple programming model and a portable implementation. It provides a simple and compact set of APIs for basic fork-join operations and global memory access. Itoyori is implemented as a C++17 library, often referred to as a "compiler-free" PGAS library [32, 76]. For communication, it employs MPI-3 RMA [39] for enhanced portability, as also adopted by recent PGAS libraries [32, 35, 67]. The tasking (threading) layer follows the *uni-address* scheme [3, 4, 65], which enables dynamic suspension and migration of user-level threads across nodes, thus realizing Cilk-like child-first work stealing [15, 16] on distributed memory at the library level. These migrating tasks access global memory through Itoyori's PGAS APIs, and global memory accesses are cached by the runtime.

Contributions. Unlike previous approaches that integrated the PGAS and global fork-join model [50, 55], Itoyori was designed with software caching in mind, which differentiated its APIs and implementation from theirs. Specifically, this paper introduces:

- New PGAS APIs designed for space-efficient access to cached global data, called *checkout/checkin* APIs (Section 3). They are designed to avoid creating unnecessary copies by directly exposing the runtime-managed cache memory to the user, which is impossible in the conventional GET/PUT APIs. Programmability is also improved by supporting unified virtual addresses for both local and global memory, as detailed in Section 3.2.
- A fixed-size, private cache implementation for checkout/checkin APIs (Section 4). Although the checkout/checkin APIs are designed to enable the possibility of exposing a shared cache to multiple cores within the same node, this feature is not currently implemented and beyond the scope of this paper.
- The cache implementation that adheres to the *work-first* principle [31] (Section 5). This principle suggests that for efficient work-stealing scheduler implementations, the overhead at each fork/join should be moved to the less frequent work-stealing events. As such, we aim to delay costly coherence actions (e.g., cache invalidation and write-back) until work-stealing events occur. To achieve this property, we designed an efficient cache coherence protocol that leverages Remote Direct Memory Access (RDMA).

Our primary contribution in this paper is to show the practicality of global-view fork-join programming using the Itoyori platform. Specifically, we experimentally demonstrate the following:

- Software caching plays a key role in scaling the fork-join model to distributed memory. We carried out experiments using three applications (Cilksort, UTS-Mem, ExaFMM) that exhibit dynamic and irregular parallelism. On 36 nodes (1728 cores), caching improved their performance by 1.4×, 6.9×, and 4.3×, respectively.
- Itoyori is not merely a toy, but a practical system for distributed-memory programming. As a real-world case study, we ported a fork-join implementation of the Fast Multipole Method (ExaFMM) [69] to distributed memory. Despite a few coding refinements and additional API calls, the primary structure of the fork-join algorithm, centered on irregular tree-based computations, remained unchanged. The Itoyori implementation exhibited a 7.5× speedup when scaled from a single node to 12 nodes and displayed comparable performance to a hand-optimized MPI implementation, highlighting its high productivity and performance.

2 BACKGROUND

Before explaining the details of Itoyori, we give more background on fork-join parallelism and distributed work stealing (in Section 2.1) and the PGAS model and systems (in Section 2.2).

2.1 Fork-Join Parallelism and Work Stealing

In this section, we first discuss the advantages of the fork-join model with a program example of Cilksort [31], a recursive parallel merge sort algorithm. Figure 1 shows the Cilksort program rewritten in C++ with Itoyori APIs. The program uses the *span* container (C++20) to represent a contiguous memory region as a pair of its address and size. The input spans are recursively divided into smaller spans according to the divide-and-conquer strategy. The `cilksort()` function takes two spans `a` and `b` and sorts the elements in `a` by using `b` as a temporary buffer. First, it logically splits both `a` and `b` into four equal-sized spans (line 8–13), each of which is then sorted recursively in parallel (line 14–18). The `parallel_invoke()` function forks multiple closures (lambda expressions) as parallel tasks and returns control when all these tasks are joined. After sorting for the four spans is completed, two pairs of them are merged into the temporary buffers in parallel (line 19–21). Then, they are merged into the original span `a` (line 22). The recursion continues until the span becomes sufficiently small (less than the `cutoff` value) and then switches to the serial quicksort algorithm (line 4–6). The details for the checkout/checkin calls and `cilkmerge()` will be explained later.

As shown above, the fork-join model allows for the concise and high-level expression of parallel algorithms. Parallel constructs (such as `parallel_invoke()`) can be nested arbitrarily, allowing spawning numerous parallel tasks regardless of the actual hardware parallelism. This is made possible by the runtime task scheduler, which maps parallel tasks to the processor cores.

Work stealing [16] is arguably the most popular task scheduler for fork-join parallelism. In work stealing, one *worker* is created per

[1]Itoyori is the Japanese name of the fish "threadfin breams." The latest version of Itoyori is being developed at https://github.com/itoyori/itoyori.

```
template <typename T>
void cilksort(span<T> a, span<T> b) {
  if (a.size() < cutoff) {
    checkout(a.data(), a.size(), mode::ReadWrite);
    quicksort_serial(a);
    checkin(a.data(), a.size(), mode::ReadWrite);
  } else {
    auto [a12, a34] = split_two(a);
    auto [a1 , a2 ] = split_two(a12);
    auto [a3 , a4 ] = split_two(a34);
    auto [b12, b34] = split_two(b);
    auto [b1 , b2 ] = split_two(b12);
    auto [b3 , b4 ] = split_two(b34);
    parallel_invoke(
      [=]{ cilksort(a1, b1); },        // sort a1
      [=]{ cilksort(a2, b2); },        // sort a2
      [=]{ cilksort(a3, b3); },        // sort a3
      [=]{ cilksort(a4, b4); });       // sort a4
    parallel_invoke(
      [=]{ cilkmerge(a1, a2, b12); },  // merge a1 and a2 -> b12
      [=]{ cilkmerge(a3, a4, b34); }); // merge a3 and a4 -> b34
    cilkmerge(b12, b34, a);            // merge b12 and b34 -> a
  }
}
template <typename T>
void cilkmerge(span<T> s1, span<T> s2, span<T> d) {
  if (d.size() < cutoff) {
    checkout(s1.data(), s1.size(), mode::Read);
    checkout(s2.data(), s2.size(), mode::Read);
    checkout(d.data() , d.size() , mode::Write);
    merge_serial(s1, s2, d);
    checkin(s1.data(), s1.size(), mode::Read);
    checkin(s2.data(), s2.size(), mode::Read);
    checkin(d.data() , d.size() , mode::Write);
  } else {
    size_t p1 = (s1.size() + 1) / 2;
    size_t p2 = binary_search(s2, &s1[p1 - 1]);
    auto [s11, s12] = split_at(s1, p1);
    auto [s21, s22] = split_at(s2, p2);
    auto [d1 , d2 ] = split_at(d , p1 + p2);
    parallel_invoke(
      [=]{ cilkmerge(s11, s21, d1); },
      [=]{ cilkmerge(s12, s22, d2); });
  }
}
```

Figure 1: Cilksort written with Itoyori in C++.

processor core or hardware thread, and each worker has its own deque to store ready tasks. A worker pushes tasks to one end of the local deque and pops tasks from the same end. When the local deque is empty, a worker tries to steal a task from another worker's deque, which is chosen uniformly at random. Because a task is stolen from the other end of the deque than local push/pop, the oldest task in each deque is stolen. In systems such as Cilk, the newly spawned task is immediately executed, pushing the continuation of the current task to the local deque to make it stealable by other workers. This policy is called the *work-first* or *child-first* policy and known to have good asymptotic bounds on execution time, space, communication [16], and data locality [1].

Recent advances in network interconnects, especially RDMA, have motivated researchers to investigate efficient inter-node work stealing [3, 4, 20, 26, 27, 33, 50, 55, 65]. However, many of these implementations come with limitations. For example, some cannot follow the child-first policy [33, 50, 55], and others only support the *bag-of-tasks* model, where tasks have no dependencies [20, 26, 27]. Excluding language-level approaches [37, 60], to the best of our knowledge, only the *uni-address scheme* [3, 4] supports the child-first policy on distributed memory at the library level. The uni-address scheme spawns tasks as user-level threads and enables dynamic migration of user-level threads across nodes. This is achieved by dynamically copying call stacks of threads to other nodes while preserving their virtual addresses on different processes. For the child-first policy, its work-stealing scheduler steals the continuation (call stacks) of threads in a fully one-sided (asynchronous) manner by utilizing RDMA. As good scalability on over 100k cores has been demonstrated [65], this strategy is considered viable even on distributed memory. Due to these benefits, Itoyori's threading layer adopts the uni-address scheme.

2.2 The PGAS Model and Systems

The PGAS model provides a global view of distributed memory, which we argue is appropriate for global fork-join parallelism. Although there might not be a clear definition of PGAS, in this paper, we define PGAS as a model that explicitly distinguishes between global and local memory access, in contrast to DSM. To date, many PGAS languages, such as Co-Array Fortran [57], Unified Parallel C (UPC) [28], XcalableMP [47], Chapel [21], and X10 [23], and PGAS libraries, such as OpenSHMEM [22], Global Arrays [56], UPC++ [76], DASH [32], and HPX [40][2] have been developed. Some PGAS systems (e.g., X10, HPX) do not offer a direct way for accessing remote memory; instead, they encourage to move computations to the data owners (i.e., *active messages*). However, our focus is on PGAS systems that allow read/write operations to remote memory without the use of active messages, because the task scheduler determines the mapping of computations. The GET/PUT APIs are commonly used in PGAS systems to copy data between global and local memory, although some PGAS languages (e.g., UPC, Chapel) implicitly insert these APIs where global objects are dereferenced. Although the details vary, the GET/PUT APIs typically appear as follows:

- `void GET(gptr_t from_ptr, void* to_addr, size_t size);`
- `void PUT(void* from_addr, gptr_t to_ptr, size_t size);`

They copy `size` bytes of data between the given local and global memory. The local memory has to be pre-allocated at the user level. The representation of global pointers (of type `gptr_t`) varies across PGAS systems and is not necessarily raw virtual addresses.

Commonly, the user can specify the memory distribution policy for global memory at the allocation time. Popular memory distribution policies are *block distribution*, which distributes memory evenly among the nodes so that each node's memory is contiguous, and *block-cyclic distribution*, which distributes fixed-size memory chunks among nodes in a round-robin fashion. Most PGAS systems have a way to directly access the local portions of global memory determined by the memory distribution policy. This helps users to follow the *owner-computes rule* [38] (i.e., the data owner node should compute on the local data) for better performance.

Arguably, the owner-computes rule assumes SPMD and is difficult to apply to irregular parallelism. Nevertheless, systems including Scioto [26], HotSLAW [50], Grappa [55], and extensions

[2]HPX refers to its global address space as *Active Global Address Space (AGAS)* to differentiate it from PGAS in that AGAS allows transparent relocation of global memory. However, in this paper, we view it within a broader scope of PGAS.

to X10 [59, 74] support inter-node work stealing to handle irregular parallelism under the PGAS model. However, for the reasons mentioned in Section 1, they often incur fine-grained and redundant communication for fine-grained parallelism. This situation motivated us to investigate software caching techniques for PGAS.

3 ITOYORI PROGRAMMING MODEL

3.1 Overview

In this section, we explain the programming model of Itoyori, a C++ library over MPI-3 RMA (assuming the `MPI_WIN_UNIFIED` model). Itoyori is composed of the threading layer and PGAS layer.

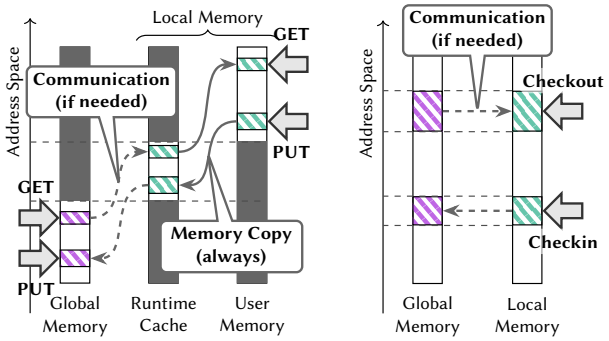
Itoyori assumes that one process is created for each core at program startup. Thus, a worker corresponds to an MPI process. Multiple kernel-level threads are not created within each process, eliminating the need for the `MPI_THREAD_MULTIPLE` support in MPI. This also means that a virtual address space is not shared among processes on the same node. Nevertheless, they can access the same physical memory through inter-process shared memory allocated for global memory (see Section 4).

An Itoyori program begins with the SPMD mode, as launched by the `mpiexec` command. Later, it can switch between the SPMD region and fork-join region by spawning the root thread. In the fork-join region, Itoyori can dynamically spawn user-level threads by using low-level threading primitives such as futures (see [65]), or high-level parallel constructs such as `parallel_invoke()` shown in Figure 1. Itoyori also supports high-level parallel patterns for range-based algorithms, similar to Intel TBB [61] and C++17 parallel STL, although we do not cover the details in this paper.

As mentioned in Section 2.1, Itoyori's threading layer employs the uni-address scheme [3, 4]. The implementation is based on our prior work [65], which uses MPI-3 RMA for fully one-sided work stealing. As it supports thread migration during both fork and join calls, the running process can change across fork-join calls. While access to local variables in the current thread's stack remains valid even after migration, accessing local variables in other threads' stacks is prohibited. This restriction arises because the uni-address scheme only copies the call stacks of the current thread upon migration. Therefore, in Itoyori, any pointers or references to local variables should not be passed to any other threads, including parents and children.

Global objects are allocated/deallocated from the global heap through PGAS APIs, which resemble typical malloc/free calls but with an additional parameter of the memory distribution policy (Section 4.2). The returned global addresses are merely raw, often 64-bit, virtual addresses. However, a program cannot directly access the virtual addresses unless checkout/checkin calls are made for the accessing region. A checkout call grants access to the requested memory region until a checkin call is made for that region. In the meantime, the region can be directly accessed with ordinary memory load/store instructions using the same virtual addresses. Also, multiple processes can concurrently check out the same region, provided that they ensure data-race-freedom.

Updates to global memory should be propagated to other processes to ensure a consistent global view of memory. Itoyori's memory consistency model is *sequential consistency for data-race-free programs (SC-for-DRF)* [2], which is also used in many languages



(a) GET/PUT with a software cache. **(b) Checkout/Checkin.**

Figure 2: Cache management for different PGAS APIs.

such as C/C++11, Java, UPC [28], and Chapel [21]. SC-for-DRF is a relaxed memory model that ensures well-defined memory ordering (sequential consistency) as long as a program has no data race. Hence, in Itoyori, global memory access (i.e., checkout/checkin calls) should be performed in a data-race-free manner. As Itoyori currently does not support other synchronization primitives (e.g., locks) than fork-join, global memory updates are propagated by following the fork-join relationships. Under this memory model, cache coherence is properly managed by the runtime system.

3.2 Rationale of Checkout/Checkin APIs

Before getting into the details, we explain why we are introducing new checkout/checkin APIs. Previous approaches added a software caching layer without changing the conventional GET/PUT APIs [19, 25, 30, 73, 75]. However, they have shortcomings in terms of both efficiency and programmability.

First, GET/PUT APIs introduce unnecessary data copying between the runtime cache and user memory, given their semantics of memory copying between the global and local memory. This issue is depicted in Figure 2a. For instance, even if a GET request results in a cache hit and thus omitting communication, the data still needs to be copied from the runtime cache to the user memory. In contrast, checkout/checkin APIs simply require a unified global address. This allows the direct exposure of the runtime cache memory to the user without any redundant copying, as illustrated in Figure 2b. In addition, even if a process checks out the same or overlapping regions at the same time, it does not lead to any space overhead. With this API design, a cache can be shared among multiple processes within the same node in the future, although this has not been implemented yet.

Similarly, GET/PUT APIs incur unnecessary data copying, even when the requested global region is local. Although many PGAS libraries offer global-to-local pointer conversion for portions of global memory that are known to be local [22, 28, 32, 56, 76], this feature assumes the SPMD model with no inter-node dynamic load balancing. In scenarios with global task parallelism, programmers would be required to insert a conditional branch for each global memory access to check if the current process owns the data. This is particularly difficult when the access region spans both local and remote memory. Arguably, checkout/checkin APIs offer a much

simpler and more straightforward interface, as they can be consistently used for both global-to-local pointer conversion and remote data access without any copying overhead.

From a programmability standpoint, checkout/checkin APIs allow a broader range of data types in line with C++ semantics. Since GET/PUT calls essentially act as a `memcpy()` function, only "trivially copyable" objects can be stored in global memory, as also noted in the UPC++ documentation [7]. This implies that certain data types, including vector containers, cannot be made global. This limitation is an actual issue in ExaFMM (Section 6.4). In contrast, checkout/checkin APIs neither create copies nor change the virtual addresses of objects, which allows for nontrivially copyable types. Admittedly, data are physically copied across nodes as raw bytes, but from the perspective of programmers, this is not considered a copy operation because virtual addresses are never changed (similar to how hardware caches work). In addition, checkout/checkin APIs simplify array indexing by consistently preserving the virtual addresses across their calls.

3.3 Programming with Checkout/Checkin APIs

With the above advantages in mind, this section explains how to program with checkout/checkin APIs. Checkout/checkin APIs must be called in pairs, and each pair requires the exact same arguments.

`void checkout(void* addr, size_t size, Mode mode);` claims that the program will access the memory of the half-open region [*addr*, *addr* + *size*) in the specified access `mode`. The requested memory region becomes accessible until `checkin()` is called for that region. The `mode` can be either `Read`, `ReadWrite`, or `Write`. If the `mode` is `Read` or `ReadWrite`, the system considers a read event for [*addr*, *addr* + *size*) happens at this point and may fetch the latest data from remote nodes. If the `mode` is `Write` (write-only access), the region may be left uninitialized.

`void checkin(void* addr, size_t size, Mode mode);` claims that access to the previously checked-out memory is completed. The arguments for `addr`, `size`, and `mode` must be exactly the same as those passed to the previous, corresponding `checkout` call. This checkin function should be called once and only once for each checkout call as a pair. If the `mode` is `ReadWrite` or `Write`, the system considers a write event for [*addr*, *addr* + *size*) happens at this point, and this region is considered dirty.

Note that in the `ReadWrite` or `Write` mode, all bytes of the checked-out data are considered dirty, even if the program did not actually update the data. In other words, the access mode in Itoyori is not like an access privilege, but more like memory load/store operations. Thus, for example, always specifying the `ReadWrite` mode is not a conservative approach; it is a data race if different processes concurrently checks out the same region in the `ReadWrite` mode, even if they do not actually write to the region.

As long as the program is data-race-free, multiple processes can simultaneously check out the same region. In other words, multiple processes can check out the same region only in the `Read` mode; otherwise only one process can check out the region at the same time. Within each process, multiple checkout requests can be simultaneously made for the same region in any access mode, but they must be checked in before program points where threads can migrate (e.g., fork-join points as explained in Section 4.4).

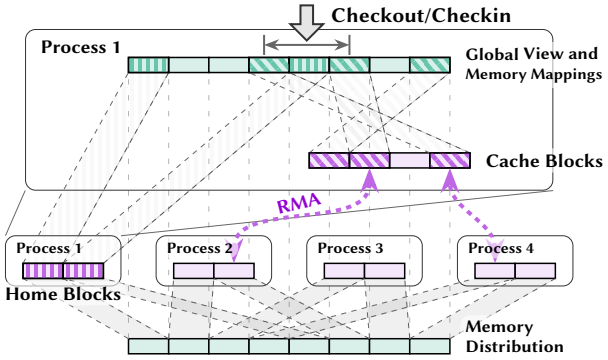


Figure 3: Overview of memory management in Itoyori.

Figure 1 shows an example usage of the checkout/checkin calls for Cilksort. At the cutoff of the recursion for `cilksort()` (line 4–6), the span `a` is checked out in the read-write access mode. Similarly, at the cutoff for `cilkmerge()` (line 28–34), the source memory (`s1` and `s2`) and the destination memory (`d`) are checked out in read-only and write-only access mode, respectively. The `cilkmerge()` function is also recursively parallelized by searching for an appropriate point to split an array. Although not shown in the code example, the binary search algorithm (line 37) internally performs sparse memory access by checking out each element in the `Read` mode.

Because the (user-configurable) cache size is fixed in Itoyori, the amount of memory that can be simultaneously checked out by each process is limited. If it exceeds the cache size limit, a checkout function returns an error. For example, if a process sweeps over a large global array that does not fit into the cache, it cannot check out the entire array at once. Instead, it has to break checkout/checkin requests into sufficiently small chunks and process each chunk in turn. While this may seem cumbersome, the details can be abstracted away by using high-level patterns for range-based operations (e.g., map, reduce). By using these high-level patterns, the system can automatically determine proper chunk sizes. This design allows us to easily handle huge data that do not fit into a single node.

4 SOFTWARE CACHE IMPLEMENTATION

This section explains the design and implementation of the cache system of Itoyori. How to integrate it with distributed work stealing is explained in Section 5.

4.1 Overview

To realize unified global addresses for the checkout/checkin APIs, Itoyori preserves the same virtual address space as a *global view* for each process and uses its addresses as global addresses. To limit the physical memory usage, physical pages are dynamically mapped to/unmapped from the global view on demand. Figure 3 illustrates the virtual-to-physical memory mappings. The memory mappings are updated at the granularity of *memory blocks* of fixed size (a multiple of the system page size). The upper part of the figure shows the global view and memory mappings to the two types of physical memory blocks: *home blocks* and *cache blocks*. Home blocks are the local portions of global memory and are mapped directly to the corresponding virtual memory addresses. Cache blocks are

used to store local copies of remote memory and mapped to the global view on demand. Cache blocks can be remapped to other locations when the memory is not being checked out. The number of cache blocks is fixed in the current implementation and can be configured by the user at program startup.

Physical memory blocks are allocated as POSIX shared memory with the `shm_open()` call. POSIX shared memory can be used to dynamically change the virtual-to-physical memory mappings with the `mmap()` call. In addition, this enables sharing of physical memory blocks among intra-node processes, even though Itoyori spawns one process per core. Home blocks are shared among intra-node processes when created, so that they can be directly mapped to each process's global view. Therefore, processes can directly access the home blocks owned by other processes within the same node. Cache blocks are not shared in the current implementation, so each process only has private caches.

4.2 Memory Distribution Policies

Itoyori extends the common `malloc()` interface so that the user can specify a memory distribution policy. The policy is either one of the *collective* policies or the *noncollective* policy.

Collective distribution policies are used to allocate a large amount of memory that spans over multiple nodes. Itoyori currently supports the block and block-cyclic distribution policies (see Section 2.2) as collective policies. The bottom part of Figure 3 shows the block-cyclic distribution. For collective policies, the allocation and deallocation function must be called collectively by all processes in the SPMD region or the root thread. At the allocation time, the same virtual address space of the requested memory size is newly preserved in all processes, and physical home blocks are allocated and exposed to other processes (calling `MPI_Win_create()`).

In contrast, the noncollective policy allows efficient fine-grained memory allocation asynchronous to other processes, even in any threads in the fork-join region. With the noncollective policy, memory objects are allocated from the local home blocks without the involvement of any other process. The allocated memory can be remotely accessed and freed by any process. Unlike collective policies, Itoyori pre-allocates a sufficiently large virtual address space for noncollective allocation at program startup. This virtual address space is divided evenly among all processes, and each process allocates memory from its local portion. We can either pre-allocate physical memory of fixed size at program startup (using `MPI_Win_create()`) or dynamically attach physical memory as the heap size grows (using `MPI_Win_create_dynamic()` and `MPI_Win_attach()`) for noncollective allocation.

4.3 Global View Management

4.3.1 Checkout/Checkin Implementation. The primary task of the checkout call is to fetch remote data to cache blocks and map them to the global view. If cache blocks are already mapped and have up-to-date data (i.e., not invalidated by the synchronization calls in Section 4.4), it can skip communication and immediately return.

Figure 4 shows an implementation of the checkout/checkin APIs. The MemBlock structure (line 1–7) is allocated for each physical memory block. Both the Checkout and Checkin functions iterate over the virtual memory blocks that overlap with the requested region $[addr, addr + size)$ and operates on each block (line 10–24 and line 32–37). *mbID* is a unique ID for each virtual memory block, which is calculated by dividing the starting virtual address by the fixed size of memory blocks (*MBSize*).

```
1  Struct MemBlock
2      type :: HomeBlock | CacheBlock
3      physMem :: handler for physical memory allocated for this block.
4      addr :: virtual address to which this block should be mapped.
5      mappedAddr :: virtual address to which this block is now mapped.
6      validRegions :: set of up-to-date memory regions within this block.
7      refCount :: reference count for this block (default: 0).
8  Function Checkout(addr, size, mode)
9      memBlocksToMap ← {}
10     for mbID ← ⌊addr/MBSize⌋ to ⌈(addr+size)/MBSize⌉ − 1 do
11         mb ← GetMemBlock(mbID)
12         mb.addr ← mbID × MBSize
13         if mb.type = CacheBlock then
14             reqRegion ← [mb.addr, mb.addr+MBSize) ∩ [addr, addr+size)
15             if mode = Write then
16                 mb.validRegions ← mb.validRegions ∪ reqRegion
17             else if reqRegion ⊄ mb.validRegions then
18                 paddedRegion ← GetOverlappingSubBlocks(reqRegion)
19                 fetchRegions ← {paddedRegion} \ mb.validRegions
20                 BeginFetch(fetchRegions, mb.physMem)
21                 mb.validRegions ← mb.validRegions ∪ fetchRegions
22         if mb.addr ≠ mb.mappedAddr then
23             memBlocksToMap ← memBlocksToMap ∪ {mb}
24         mb.refCount ← mb.refCount + 1
25     for mb ∈ memBlocksToMap do
26         if mb.mappedAddr ≠ NULL then
27             UnmapPhysMem(mb.mappedAddr, MBSize)
28         MapPhysMem(mb.addr, MBSize, mb.physMem)
29         mb.mappedAddr ← mb.addr
30     WaitFetchCompletion()
31 Function Checkin(addr, size, mode)
32     for mbID ← ⌊addr/MBSize⌋ to ⌈(addr+size)/MBSize⌉ − 1 do
33         mb ← GetMemBlock(mbID)
34         if mode ≠ Read and mb.type = CacheBlock then
35             reqRegion ← [mb.addr, mb.addr+MBSize) ∩ [addr, addr+size)
36             RegisterDirtyRegion(mb, reqRegion)
37         mb.refCount ← mb.refCount − 1
```

Figure 4: Implementation of checkout/checkin operations.

The GetMemBlock function (line 11 and line 33) queries the physical memory block associated with *mbID*. This translation involves two separate fixed-size hash tables for home and cache blocks, respectively. If the hash table already has an entry for the given ID, the handler for the associated physical memory block (*mb* of type MemBlock) is returned; otherwise, the ID is associated with a free memory block and its handler is returned. If no free memory block is found, an existing mapping entry is evicted based on the least recently used (LRU) policy. The LRU priority is managed with a doubly-linked LRU list. When a memory block is queried (GetMemBlock()), its LRU entry is moved to the tail of the LRU list. Upon eviction, the LRU list is traversed from the head to the tail

until an evictable memory block is found. If no evictable block is found, the checkout function raises a too-much-checkout exception.

A memory block is evictable if it is not dirty (see Section 4.4) and its reference count is zero. The reference count is incremented on the checkout call (line 24) and decremented on the checkin call (line 37). This ensures that physical memory is present while the region is being checked out[3]. We do the same for home blocks, the reason for which will be explained in Section 4.3.2.

After getting a cache block, we make sure that the requested data are up-to-date (line 13–21). In order to manage which parts of a block are up-to-date, each cache block maintains a set of valid regions (*mb.validRegions*). This is currently implemented as a linked list of byte-granularity intervals, although a bitmap would be another option. If the access mode is write-only, the exact region requested by the user (*reqRegion*) is added to the valid regions without fetching remote data (line 16). Otherwise, we check if the requested region is up-to-date (line 17), and if not, the remote data are fetched. The remote fetch is performed at the *sub-block* granularity to exploit spatial data locality. That is, each memory block is logically divided into one or more equal-sized sub-blocks, and the sub-blocks that overlap with the requested region are fetched at once (line 18). Note that the already valid regions should not be fetched (line 19), so as not to overwrite the dirty data. Then, nonblocking communication (using `MPI_Get()`) is issued to fetch the selected regions *fetchRegions* from the data owner (line 20). The completion of communication is awaited at the end of the Checkout function (using `MPI_Win_flush_all()` at line 30).

Virtual memory mappings are updated *after* starting nonblocking communication for all blocks, in order to hide the overhead of the `mmap()` system call. In Figure 4, *mb.mappedAddr* is the virtual address that the block is currently mapped to, and *mb.addr* is the address that it needs to be mapped to. If they are different (line 22), its memory mapping needs to be updated. Such memory blocks are added to a list *memBlocksToMap*, and after all communication requests are issued, their memory mappings are updated with `mmap()` (line 25–29). Therefore, virtual addresses of cache blocks can change during communication, but this is not a problem because they are assigned different virtual addresses for communication in advance.

The Checkin function is responsible for managing dirty data in each cache block, which is registered by the RegisterDirtyRegion function if not read-only (at line 36). The management of dirty data depends on the cache coherence protocol explained in Section 4.4.

4.3.2 Saving the Number of Memory Mapping Entries. Itoyori potentially creates many memory mappings with the `mmap()` system call, but unfortunately, Linux has a limit to the number of memory mapping entries. In our experimental environment (Section 6.1), we can create only 65530 mapping entries per process[4], which is problematic in practice. Therefore, we explicitly unmap[5] the previous mapping when the mapping changes (line 27 in Figure 4), although we do not need to do this if there is no such limitation.

Nevertheless, the total cache size in Itoyori is restricted by this limitation. A memory mapping entry is counted for each contiguous region of virtual memory that is also contiguous in physical memory. For N cache blocks, we would need $2N + 1$ entries in the worst case (i.e., when the mappings are interleaved), because we also need to preserve the addresses to which no block is mapped. As the minimum block size is 64 KB in our environment, the maximum cache size is approximately $65530/2 \times 64\,\text{KB} \sim 2\,\text{GB}$ for each process.

Even for home blocks, this limitation is problematic for certain memory distribution policies with interleaved memory mappings (e.g., block-cyclic distribution). As a workaround, we limit the number of home blocks that can be simultaneously mapped for each process. This is why home blocks are managed similarly to cache blocks, using a hash table and reference counts in Figure 4. This means that home blocks are also subject to eviction and no longer statically mapped to the global view. Because of this reason, Itoyori requires all global memory accesses go through the checkout/checkin calls, even if the requested region is known to be local. Note that we can skip dynamic home block management for block distribution, in which consumption of memory mapping entries is small.

4.4 Cache Coherence Protocol

Itoyori employs a relaxed memory consistency model of SC-for-DRF [2], as briefly mentioned in Section 3.1. Under this relaxed memory model, a simple cache coherence protocol can be used, assuming that the data-race-freedom is already ensured by programmers. Following the convention (e.g., [34]), Itoyori offers *release* and *acquire* memory fences to ensure a consistent global view of memory. Informally, if a release fence happened before an acquire fence, there is a synchronization order, i.e., all updates made before the release fence must be observed after the acquire fence. These fences are typically hidden from the user and encapsulated by synchronization primitives, such as locks, barriers, and fork-join calls. As Itoyori currently supports only fork-join, the memory model is equivalent to *DAG consistency* [13, 14]. We will explain how to insert memory fences to fork-join primitives in Section 5.

To follow the synchronization order in Itoyori, each process performs coherence actions for its local cache blocks. A release fence ensures that all dirty data in the local cache are written to their homes, and an acquire fence self-invalidates all local caches (by clearing *validRegions* in Figure 4) so that successive checkout operations will fetch the latest data from their homes. In this way, Itoyori ensures the order between the write events before a release fence and the read events after the associated acquire fences. Although one might feel that this coherence protocol is too naive, similar approaches are taken by GPU's hardware caches [68] and Chapel's software cache [30] because of its simplicity.

In this paper, we consider two approaches to handling dirty data (RegisterDirtyRegion() at line 36 in Figure 4): the *write-through* and *write-back* policies. With the write-through policy, the dirty data are written to their homes immediately on each checkin call, without remembering the dirty regions. On the other hand, with the write-back policy, we delay flushing dirty data until the next release fence by remembering the dirty regions. The dirty regions are managed on a per-block basis and are maintained in the same way as the valid regions (*validRegions*) as a linked list of memory regions. When a release fence is executed, all dirty regions are written back to their homes in byte granularity. As long as a cache

[3]The reference count cannot be less than zero, as long as the checkout/checkin APIs are called in pairs (see the API usage in Section 3.3).
[4]checked with the "`sysctl vm.max_map_count`" command.
[5]`mmap()` with no access privilege (`PROT_NONE`) is used to preserve the virtual addresses while unmapping physical memory. The `munmap()` call is not used for this purpose.

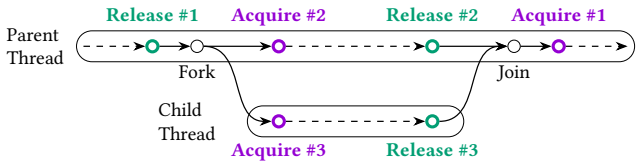


Figure 5: Possible release/acquire fences in fork-join calls.

block has any dirty region, the block is not evictable. If all cache blocks are not evictable when a free cache block is needed, then the system writes back all dirty data and retries the eviction procedure.

5 INTEGRATION WITH WORK STEALING

This section explains how Itoyori integrates the cache system with work stealing, with an aim to to follow the work-first principle [31] by delaying costly coherence actions until work stealing occurs.

5.1 Release/Acquire Fences in Work Stealing

As Itoyori's threads can be dynamically migrated to other processes at fork-join calls, release/acquire fences are inserted to fork-join points. Figure 5 shows possible program points to insert release/acquire fences. Since the modifications made before the fork must be read by the child and the continuation of the parent, we insert release/acquire fences accordingly. Similarly, the process that executes the continuation of the join must read the modifications made by the child and the parent before the join.

Obviously, this naive approach is not efficient for fine-grained parallelism, but if we assume certain scheduling policies, we can reduce the number of fences. As mentioned earlier (Section 2.1), the work-stealing scheduler of Itoyori follows the child-first policy. As the child thread is immediately executed after the fork, Acquire #3 in Figure 5 can always be skipped. In addition, as long as the parent thread is not stolen, Release #2, #3 and Acquire #1, #2 can be skipped, because the child thread can be treated as a serialized function call [31]. Conversely, if the parent thread is stolen by another process, all of these fences are executed. Release #1 is the only fence that is nontrivial to skip, because we do not know in advance if the parent thread will be stolen or not.

5.2 Lazy Execution of Release Fences

The release fence before forking (Release #1) is, unlike the fences that are conditionally executed only when work stealing happens, performance critical for fine-grained parallelism. This is because fork-join calls are usually much more frequent than work stealing events (cf. the work-first principle [31]). That is, the more fine-grained the threads are, the more release fences will be executed.

Therefore, we consider delaying the execution of Release #1 until the parent thread is stolen. This would require the thief to notify the victim of the steal event and wait for the release to complete. However, naive implementations would cause frequent interruptions on the victim (e.g., by active messages), which can diminish the benefits of RDMA-based asynchronous work stealing.

Following the work-first principle, we designed an algorithm that can minimize the victim's overhead. In our algorithm, the thief sends a release request to the victim and the victim polls the

```
38 Struct ProcessLocalData
39     myProcID :: ID (rank) of the local process.
40     currentEpoch :: current epoch of the release operation.
41     requestEpoch :: epoch of the data requested to release by others.
42 Struct ReleaseHandler
43     procID :: ID (rank) of the process who owns the data to be released.
44     epoch :: epoch of the data to be released.
45 Function ReleaseLazy() :: ReleaseHandler
46     if all cache blocks are clean then return Unneeded
47     else return (myProcID, currentEpoch + 1)
48 Function Acquire(handler :: ReleaseHandler)
49     if handler ≠ Unneeded then
50         while Get(currentEpoch at handler.procID) < handler.epoch do
51             if first time then
52                 atomic op at handler.procID do
53                     requestEpoch ← max(requestEpoch, handler.epoch)
54     Invalidate all cache blocks.
55 Function DoReleaseIfRequested()
56     if currentEpoch < requestEpoch then
57         Write back all dirty data to their homes.
58         currentEpoch ← currentEpoch + 1
```

Figure 6: Implementation of lazy execution for release fences.

requests. Each polling operation can be performed quickly because the check does not involve a communication call, since it only reads local variables that may be modified by remote processes via MPI-3 RMA (assuming the `MPI_WIN_UNIFIED` model).

Figure 6 shows our implementation. The ReleaseLazy function (line 45–47) is the function to be executed at Release #1. It returns a release handler (ReleaseHandler at line 42–44), which is later passed to the thief and used to request a write-back for the dirty data at this point. A release handler is a pair of the process ID (MPI rank) and an *epoch*. An epoch is managed by each process (*currentEpoch*) and incremented at each write-back operation by the process. If the local cache is dirty, *currentEpoch* + 1 is returned as an epoch for the release handler (line 47). This indicates that the next write-back operation must be completed by this process to ensure the synchronization order. If the cache is clean, then no write-back request is needed and *Unneeded* is returned as a release handler. Release handlers are then passed to the corresponding acquire fences (Acquire #2) by value.

The Acquire fence function (line 48–54) is called only when the parent thread is stolen. If the handler is not *Unneeded*, it first fetches the current epoch of the releaser to check if the next write-back operation has already been performed (line 50). If it is still smaller than the required epoch (*handler.epoch*), then a write-back request is sent to the releaser only once (line 52–53). Our insight is that, even if multiple acquirers simultaneously send write-back requests to the same releaser, only the maximum epoch among them is sufficient. Therefore, we use a remote atomic operation to set the maximum epoch at the releaser's memory[6]. The acquirer

[6]The maximum value can be set by using the `MPI_Fetch_and_op()` function with the `MPI_MAX` operation, but this operation is usually not offloaded to RDMA. Instead, we emulate this operation using the `MPI_Compare_and_swap()` function with a loop.

Table 1: Experimental environment.

Processor	Fujitsu A64FX (@ 2.2 GHz, 48 cores/node)
Architecture	ARMv8.2-A + SVE
Memory	HBM2 (32 GiB/node)
Communication	Fujitsu MPI 4.0.1 over Tofu Interconnect D
C++ Compiler	Fujitsu compiler 4.8.1 (with `-O3 -Nclang` options)
OS	RHEL 8.5 (kernel `4.18.0-348.20.1.el8_5`)

then waits until the remote epoch reaches the required epoch by repeatedly getting the remote epoch (using `MPI_Get()`).

To periodically check the update on the requested epoch, the polling function (DoReleaseIfRequested at line 55–58) is inserted to each fork and join call. If the requested epoch is greater than the current epoch, the process writes back all dirty data and increments the current epoch, so that the acquirers can break the loop. Note that long-running tasks can delay the execution of the polling function for a long time in this implementation. Another approach would be to call DoReleaseIfRequested in a dedicated kernel-level thread, but this has not been implemented yet.

6 EVALUATION

The primary goal of our performance evaluation is to demonstrate that the fork-join model can successfully scale to distributed memory with the help of software caching. To this end, we ran three fork-join applications: Cilksort (Section 6.2), UTS-Mem (Section 6.3), and ExaFMM (Section 6.4), which all involve a reasonable amount of global memory access. These applications are written in a shared-memory-like, global fork-join model using Itoyori APIs, and no explicit load balancing is performed in their code.

6.1 Experimental Settings

Table 1 summarizes the configuration of our experimental environment. Its configuration is similar to that of the supercomputer Fugaku, which consists of the Fujitsu A64FX CPUs and the Tofu interconnect D. We used Fujitsu MPI, which offloads MPI-3 RMA calls to RDMA operations. We allocated our jobs with 1, 2, 6 (2×3), 12 ($2 \times 3 \times 2$), and 36 ($3 \times 4 \times 3$) nodes as a torus topology and used all cores by spawning 48 MPI processes per node. We set the memory block size as 64 KB, which was the minimum page size in our environment. We also set the software cache size as 128 MB per process and the sub-block size as 4 KB. The block-cyclic distribution policy was employed for collective memory allocation. We repeated executions for 30 times after one warm-up run and plotted their mean and the 95% confidence interval as error bars, which were sufficiently small in most cases. Serial execution times for calculating speedups were measured by eliding all Itoyori runtime calls (e.g., fork-join, checkout/checkin) from the program.

As a baseline for the naive integration of the PGAS and fork-join models, we implemented GET/PUT APIs without caching in Itoyori. This paper does not include a performance comparison with other existing PGAS systems, mainly due to the portability issue in the network layer. Nevertheless, we consider our GET/PUT implementation as a reasonable baseline. This is because these GET/PUT APIs are merely a thin wrapper for MPI calls (`MPI_Get()` and `MPI_Put()`), and MPI-3 RMA is also considered a PGAS library.

6.2 Cilksort

Cilksort is a recursive parallel merge sort algorithm shown in Figure 1. We measured the time to sort an array of 4-byte integers that were generated uniformly at random.

First, we evaluate different caching policies by varying the task cutoff count (the `cutoff` value in Figure 1). Figure 7 shows the result on 12 nodes (576 cores). *No Cache* is the GET/PUT implementation without caching, executed by replacing the checkout/checkin calls with the GET/PUT calls by allocating user buffers for them. *Write-Through* and *Write-Back* represent the write-through and write-back cache discussed in Section 4.4. *Write-Back (Lazy)* means the lazy write-back policy for release fences (Section 5.2). The result clearly shows that the more we delayed the write-back operation, the better performance we got. In particular, when the cutoff count was as low as 64, *Write-Back (Lazy)* was 1.58×, 2.13×, and 12.4× faster than *Write-Back*, *Write-Through*, and *No Cache*, respectively. This demonstrates that *Write-Back (Lazy)*, which faithfully adheres to the work-first principle, is the most robust to fine-grained parallelism.

Then, we conducted a scalability study for two array sizes (1G and 10G elements) with the best-performing cutoff count of 16K. We show only *Write-Back (Lazy)* as the cache-enabled version because the performance difference was subtle with sufficiently large cutoff counts. Figure 8 shows the scalability. Itoyori achieved a 325× speedup on 36 nodes (1728 cores) compared to serial execution with 1G elements. Even compared to the `std::sort()` implementation, it was a 266× speedup on this number of nodes. Software caching improved performance by 37% on 36 nodes with 10G elements, but not with 1G elements. This is because 10G elements offered abundant parallelism, which enhanced cache reuse within each process. Note that the 10G-element experiments demonstrate that the multi-node execution with Itoyori can handle working sets larger than the single-node memory (32 GB), as it requires at least $10\text{G} \times 4$ bytes (integer) $\times 2$ (double buffering) = 80 GB of memory.

Although the execution times decreased as the number of nodes increased, parallel efficiency of Itoyori was nevertheless low on a large core count. Figure 9 shows the performance breakdown reported by Itoyori's profiler. The y-axis represents the times accumulated over all processes, normalized to the total accumulated time on 1728 cores for each problem size. *Get* is the accumulated time to load a single element during binary search in the merge phase, while *Checkout* and *Checkin* correspond to the calls shown in Figure 1, The *Release* time is accumulated for normal release operations (Release #2 and #3 in Figure 5), and the *Lazy Release* time is for delayed write-back operations (required by Release #1). The *Acquire* time is mostly the wait time for the lazy release. *Serial Quicksort* and *Serial Merge* are serial computations at leaf tasks. The *Others* time is mostly the time for scheduling events (e.g., work stealing trials). The result shows that, while the accumulated times for serial computations were almost constant, those for other communication-related events increased as the number of cores increased. On 1728 cores, the ratio of computation time was only about 20%. The *Others* time on 1728 cores was particularly long for 1G elements but not for 10G elements, because the workload for 1G elements was not enough to keep all processes busy. We believe that locality-aware schedulers would greatly reduce communication, as we currently use purely random work stealing, but we leave this for future work.

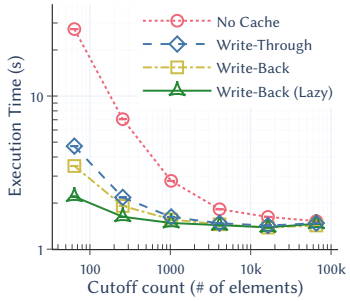


Figure 7: Execution time of Cilksort (1G elements) with various cutoff counts on 12 nodes (576 cores).

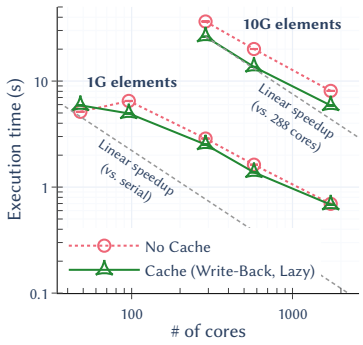


Figure 8: Strong scaling of Cilksort.

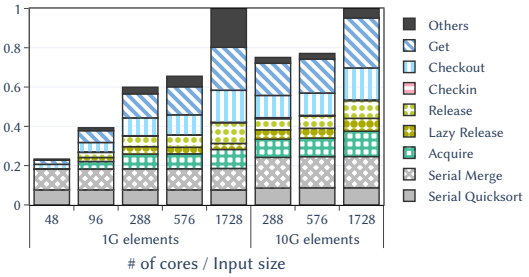


Figure 9: Performance breakdown of *Write-Back (Lazy)* in Cilksort. Accumulated time are normalized to those on 1728 cores for each input size.

6.3 UTS-Mem

The memory access granularity in Cilksort is easy to enlarge by increasing the cutoff count, but this is not always the case in more dynamic and irregular workloads. UTS-Mem [54], an extension to the unbalanced tree search (UTS) benchmark [58], involves dynamic, irregular, and fine-grained memory access. The task of the original UTS benchmark is to count the total number of nodes in an unbalanced tree. However, the workload of UTS is not realistic, as it does not involve global memory access. The tree is not in memory but is dynamically generated from the root in a deterministic way, by using hash calculation during tree traversal. In contrast, UTS-Mem generates the same tree as the original UTS and stores it in memory by allocating memory objects from the global heap.

In our experiment, we measure the traversal time for the tree constructed in global memory in advance. As the tree traversal is performed by chasing global pointers, many fine-grained memory accesses are performed. Although each tree node is accessed only once, runtime caching can help improve performance by exploiting spatial data locality. In this benchmark, close tree nodes are likely to be located in close memory regions (e.g., within the same memory block), because the tree construction is also done in parallel by work stealing. During tree construction, the memory objects for tree nodes are locally allocated with the noncollective mode.

Figure 10 shows the scalability for two different tree sizes: T1L (102,181,082 nodes) and T1XL (1,635,119,272 nodes). The y-axis shows the throughput (the number of tree nodes counted per second). Note that we do not compare different caching policies because all global memory accesses are read-only during tree traversal. Overall, Itoyori scales well and greatly outperforms the no-cache version by exploiting spatial data locality. For T1XL, Itoyori showed a good scalability (a 2.5× speedup from 12 nodes to 36 nodes) and 7.1× better performance than the no-cache version on 36 nodes.

6.4 ExaFMM

ExaFMM [72] is a Fast Multipole Method (FMM) library for N-body simulation. It manages particles using a tree (called an *octree*) by recursively partitioning the 3D space into eight parts until the number of particles becomes less than a threshold. By leveraging the octree, it approximates the force interactions between far enough particles to reduce computation. This makes its workload highly dynamic and irregular. On shared memory, nested fork-join parallelization for ExaFMM has been explored [69]. This implementation straightforwardly parallelizes the tree-based computation with a recursive divide-and-conquer method.

As a real-world case study, we ported this fork-join implementation of ExaFMM [71] to Itoyori. A major change from the original code is the insertion of the checkout/checkin calls to where global memory is accessed (e.g., computation kernels). In addition, we modified the program so that the parent thread stack is never accessed by its children (see Section 3.1). This is mostly accomplished by passing variables to child threads by value. Overall, we did not have to change the whole structure of the original parallel algorithm. Itoyori allows for much easier porting than the message-passing model, which would require redesigning the parallel algorithm [72].

In our experiments, we computed the Laplace kernel for particles distributed in a cube with the parameters $\theta = 0.2$, $N_{crit} = 32$, $P = 4$, $nspawn = 1000$ (see [69] for these parameters). Figure 11 shows the results for 1M and 10M particles. Technically, the GET/PUT implementation (*No Cache*) is illegal in C++ because each octree node has a nontrivially copyable, global vector container (see Section 3.2). Overall, the cache-enabled versions outperformed the no-cache version (up to 6.0× faster). This large performance improvement is due to ExaFMM's high degree of temporal and spatial locality for both particles and octree nodes, although the computation pattern is irregular. The performance of the write-back cache was better than that of the write-through cache, but the lazy execution for release fences did not improve the performance in this application. The scalability for 10M bodies was better than that for 1M bodies because of sufficient parallelism, and notably, it showed a 313× speedup on 12 nodes (576 cores) compared to the serial execution.

We also report a comparison with the MPI implementation of ExaFMM [71, 72]. Its shared-memory algorithm is the same fork-join algorithm, but particles are distributed across nodes with MPI. We used MassiveThreads [53] for thread scheduling within each node. As shown in Figure 11, although the MPI version outperforms Itoyori for some cases (up to 1.7× faster on six nodes), Itoyori shows comparable performance to MPI overall. The main reason why MPI sometimes performs worse than Itoyori is load imbalance. The MPI version performs only static load balancing based on the particle count, which results in load imbalance due to the dynamic and irregular nature of tree-based computation. To validate this, we

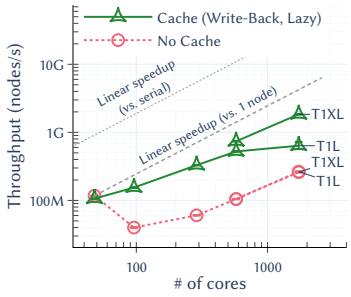


Figure 10: Throughput of UTS-Mem (strong scaling).

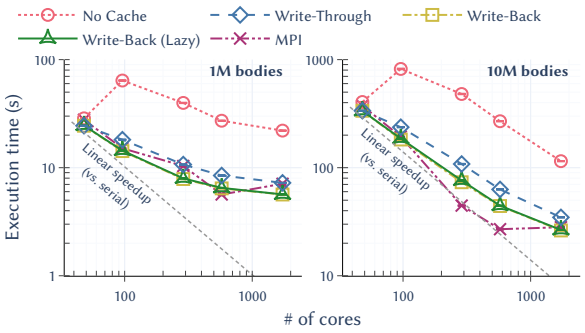


Figure 11: Execution times of ExaFMM (strong scaling).

Table 2: Load balance in ExaFMM (MPI) with 10M bodies.

# of nodes (cores)	Idleness
1 (48)	0
2 (96)	0.01
6 (288)	0.04
12 (576)	0.14
36 (1728)	0.27

measured the "idleness" metric of the MPI program, which is the ratio of the total idle time during which MPI processes await the completion of others to the overall execution time. Table 2 shows the idleness for each node count. On 36 nodes, as much as 27% of the total time was consumed due to load imbalance.

7 RELATED WORK

The concept of a global address space is old; the research has begun in the form of distributed shared memory (DSM) systems, such as IVY [48], Munin [10], TreadMarks [42, 43], and Midway [11]. *DAG consistency* [13, 14] for fork-join parallelism was also explored in 1990s. Recent advances in RDMA-capable network interconnects have motivated researchers to investigate RDMA-friendly DSM systems, such as ArgoDSM [41], Popcorn [24], and MENPS [29]. In these DSM systems, transparency is achieved by trapping memory protection faults at the page granularity, but it comes at a cost.

As discussed in Section 2.2, the PGAS model has attracted attention because of its performance and programmability. Most PGAS systems prefer explicit communication and thus do not have a caching mechanism, but exceptionally, some PGAS systems [19, 25, 30, 73, 75] implement a software cache. However, they assume the SPMD model and do not target the global fork-join model, in which tasks are dynamically scheduled across nodes. In addition, they implement a software cache over the GET/PUT APIs, which fall short for our purpose as discussed in Section 3.2.

While some PGAS systems support inter-node dynamic load balancing for tasks [26, 50, 55, 59, 74], a majority of "asynchronous" PGAS systems support only intra-node dynamic load balancing, including Chapel [21], HPX [40], GMT [52], HabaneroUPC++ [45], AsyncSHMEM [36], X10 [23], and tasking extensions to Dash [63] and XcalableMP [70]. In these systems, inter-node load balancing is still on the user's responsibility. Their intent of intra-node dynamic task scheduling is partly to overlap communication and computation by context switching to other tasks while communication is ongoing, which is orthogonal to our caching approach. Grappa [54, 55] is a system that takes this idea into account while supporting inter-node work stealing, but without software caches. The idea of communication-computation overlap may also be applicable to Itoyori, but we leave this investigation for future work.

Another spectrum of research is task-based distributed runtime systems that tightly couple tasks and data to perform inter-node dynamic load balancing. These systems include Legion [9], KAAPI [33], PaRSEC [17], StarPU [5], OmpSs@Cluster [18, 49], and Tascell [37]. In these systems, the input/output data of each task are implicitly or explicitly specified by users with access privileges, and the data are automatically moved to the nodes where the associated tasks are executed. Itoyori is different from these systems in that it decouples data from the task specifications for simplicity of APIs and flexibility of global memory access.

Compiling shared-memory programs to distributed-memory programs is an alternative approach to achieving better productivity on distributed memory. Compilation from regular, loop-based OpenMP programs to MPI programs has been demonstrated through data-flow analysis [8, 46]. However, this approach is not applicable to dynamic and irregular applications that our work focuses on.

8 CONCLUSION AND FUTURE WORK

In this paper, we introduced Itoyori, a global-view fork-join runtime system. Itoyori effectively addresses the challenge of combining the PGAS and fork-join model by incorporating software caching for global memory access. Our evaluation reveals that software caching substantially improves performance, especially for fine-grained parallelism. The three fork-join applications we tested were written in a concise and intuitive manner, akin to shared-memory fork-join programs, while demonstrating good scalability on multiple nodes. In conclusion, we believe that Itoyori presents a viable solution for achieving an optimal balance between productivity and performance in distributed-memory programming.

Arguably, there is substantial work for improving Itoyori's performance in the future. The top priority is improving the scheduler to consider the memory hierarchy to reduce communication. Locality-aware schedulers, such as *almost deterministic work stealing* [64, 66], would be well-suited for distributed memory. Other future directions include improvements to the cache coherence protocol, cache sharing among intra-node processes, communication-computation overlap, and so on.

ACKNOWLEDGMENTS

This research was conducted using the FUJITSU Supercomputer PRIMEHPC FX1000 (Wisteria/BDEC-01) at the Information Technology Center, The University of Tokyo. This work was supported by JSPS KAKENHI Grant Number 21J22305 and JST, CREST Grant Number JPMJCR21M2, Japan.

REFERENCES

[1] Umut A. Acar, Guy E. Blelloch, and Robert D. Blumofe. 2000. The Data Locality of Work Stealing. In *Proceedings of the Twelfth Annual ACM Symposium on Parallel Algorithms and Architectures* (Bar Harbor, Maine, USA) *(SPAA '00)*. 1–12.

[2] Sarita V. Adve and Mark D. Hill. 1990. Weak Ordering – A New Definition. In *Proceedings of the 17th Annual International Symposium on Computer Architecture* (Seattle, Washington, USA) *(ISCA '90)*. 2–14.

[3] Shigeki Akiyama and Kenjiro Taura. 2015. Uni-Address Threads: Scalable Thread Management for RDMA-Based Work Stealing. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing* (Portland, Oregon, USA) *(HPDC '15)*. 15–26.

[4] Shigeki Akiyama and Kenjiro Taura. 2016. Scalable Work Stealing of Native Threads on an x86-64 Infiniband Cluster. *Journal of Information Processing* 24, 3 (May 2016), 583–596.

[5] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. 2009. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. In *Proceedings of the 15th International European Conference on Parallel and Distributed Computing* (Delft, The Netherlands) *(Euro-Par '09)*. 863–874.

[6] Eduard Ayguadé, Nawal Copty, Alejandro Duran, Jay Hoeflinger, Yuan Lin, Federico Massaioli, Xavier Teruel, Priya Unnikrishnan, and Guansong Zhang. 2008. The Design of OpenMP Tasks. *IEEE Transactions on Parallel and Distributed Systems* 20, 3 (June 2008), 404–418.

[7] John Bachan, Scott Baden, Dan Bonachea, Johnny Corbino, Johnathan Grossman, Paul H. Hargrove, Steven Hofmeyr, Mathias Jacquelin, Amir Kamil, Brian Van Straalen, and Daniel Waters. 2022. *UPC++ v1.0 Programmer's Guide, Revision 2022.9.0.* Technical Report LBNL-2001479. Lawrence Berkeley National Laboratory, USA.

[8] Ayon Basumallik and Rudolf Eigenmann. 2006. Optimizing Irregular Shared-Memory Applications for Distributed-Memory Systems. In *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (New York, New York, USA) *(PPoPP '06)*. 119–128.

[9] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. 2012. Legion: Expressing Locality and Independence with Logical Regions. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (Salt Lake City, Utah, USA) *(SC '12)*. 66:1–66:11.

[10] J. K. Bennett, J. B. Carter, and W. Zwaenepoel. 1990. Munin: Distributed Shared Memory Based on Type-Specific Memory Coherence. In *Proceedings of the Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Seattle, Washington, USA) *(PPOPP '90)*. 168–176.

[11] B.N. Bershad, M.J. Zekauskas, and W.A. Sawdon. 1993. The Midway Distributed Shared Memory System. In *Digest of Papers. The 38th IEEE Computer Society International Conference* (San Francisco, California, USA) *(COMPCON Spring '93)*. 528–537.

[12] Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, and Harsha Vardhan Simhadri. 2011. Scheduling Irregular Parallel Computations on Hierarchical Caches. In *Proceedings of the Twenty-Third Annual ACM Symposium on Parallelism in Algorithms and Architectures* (San Jose, California, USA) *(SPAA '11)*. 355–366.

[13] Robert D. Blumofe, Matteo Frigo, Christopher F. Joerg, Charles E. Leiserson, and Keith H. Randall. 1996. An Analysis of Dag-Consistent Distributed Shared-Memory Algorithms. In *Proceedings of the Eighth Annual ACM Symposium on Parallel Algorithms and Architectures* (Padua, Italy) *(SPAA '96)*. 297–308.

[14] Robert D. Blumofe, Matteo Frigo, Christopher F. Joerg, Charles E. Leiserson, and Keith H. Randall. 1996. Dag-Consistent Distributed Shared Memory. In *Proceedings of the 10th International Parallel Processing Symposium* (Honolulu, Hawaii, USA) *(IPPS '96)*. 132–141.

[15] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. 1995. Cilk: An Efficient Multithreaded Runtime System. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Santa Barbara, California, USA) *(PPoPP '95)*. 207–216.

[16] Robert D. Blumofe and Charles E. Leiserson. 1999. Scheduling Multithreaded Computations by Work Stealing. *J. ACM* 46, 5 (Sept. 1999), 720–748.

[17] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Mathieu Faverge, Thomas Herault, and Jack J. Dongarra. 2013. PaRSEC: Exploiting Heterogeneity to Enhance Scalability. *Computing in Science & Engineering* 15, 6 (2013), 36–45.

[18] Javier Bueno, Luis Martinell, Alejandro Duran, Montse Farreras, Xavier Martorell, Rosa M Badia, Eduard Ayguade, and Jesús Labarta. 2011. Productive Cluster Programming with OmpSs. In *Proceedings of the 17th International European Conference on Parallel and Distributed Computing* (Bordeaux, France) *(Euro-Par '11)*. 555–566.

[19] Qingchao Cai, Wentian Guo, Hao Zhang, Divyakant Agrawal, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, Yong Meng Teo, and Sheng Wang. 2018. Efficient Distributed Memory Management with RDMA and Caching. *Proceedings of the VLDB Endowment* 11, 11 (July 2018), 1604–1617.

[20] Hannah Cartier, James Dinan, and D. Brian Larkins. 2021. Optimizing Work Stealing Communication with Structured Atomic Operations. In *Proceedings of the 50th International Conference on Parallel Processing* (Lemont, Illinois, USA) *(ICPP '21)*. 36:1–36:10.

[21] Bradford L. Chamberlain, David Callahan, and Hans P. Zima. 2007. Parallel Programmability and the Chapel Language. *International Journal of High Performance Computing Applications* 21, 3 (2007), 291–312.

[22] Barbara Chapman, Tony Curtis, Swaroop Pophale, Stephen Poole, Jeff Kuehn, Chuck Koelbel, and Lauren Smith. 2010. Introducing OpenSHMEM: SHMEM for the PGAS Community. In *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model* (New York, New York, USA) *(PGAS '10)*. 1–3.

[23] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. 2005. X10: An Object-Oriented Approach to Non-Uniform Cluster Computing. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (San Diego, California, USA) *(OOPSLA '05)*. 519–538.

[24] Ho-Ren Chuang, Robert Lyerly, Stefan Lankes, and Binoy Ravindran. 2020. Scaling Shared Memory Multiprocessing Applications in Non-Cache-Coherent Domains. In *Proceedings of the 13th ACM International Systems and Storage Conference* (Haifa, Israel) *(SYSTOR '20)*. 13–24.

[25] Salvatore Di Girolamo, Flavio Vella, and Torsten Hoefler. 2017. Transparent Caching for RMA Systems. In *Proceedings of the 31st IEEE International Parallel and Distributed Processing Symposium* (Orlando, Florida, USA) *(IPDPS '17)*. 1018–1027.

[26] James Dinan, Sriram Krishnamoorthy, D. Brian Larkins, Jarek Nieplocha, and P. Sadayappan. 2008. Scioto: A Framework for Global-View Task Parallelism. In *Proceedings of the 37th International Conference on Parallel Processing* (Portland, Oregon, USA) *(ICPP '08)*. 586–593.

[27] James Dinan, D. Brian Larkins, Ponnuswamy Sadayappan, Sriram Krishnamoorthy, and Jarek Nieplocha. 2009. Scalable Work Stealing. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis* (Portland, Oregon, USA) *(SC '09)*. 53:1–53:11.

[28] Tarek El-Ghazawi, William Carlson, Thomas Sterling, and Katherine Yelick. 2005. *UPC: Distributed Shared Memory Programming.* John Wiley & Sons.

[29] Wataru Endo, Shigeyuki Sato, and Kenjiro Taura. 2020. MENPS: A Decentralized Distributed Shared Memory Exploiting RDMA. In *Proceedings of 2020 IEEE/ACM Fourth Annual Workshop on Emerging Parallel and Distributed Runtime Systems and Middleware* (Virtual Event) *(IPDRM '20)*. 9–16.

[30] Michael P. Ferguson and Daniel Buettner. 2015. Caching Puts and Gets in a PGAS Language Runtime. In *Proceedings of the 2015 9th International Conference on Partitioned Global Address Space Programming Models* (Washington, District of Columbia, USA) *(PGAS '15)*. 13–24.

[31] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. 1998. The Implementation of the Cilk-5 Multithreaded Language. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation* (Montreal, Quebec, Canada) *(PLDI '98)*. 212–223.

[32] Karl Fuerlinger, Tobias Fuchs, and Roger Kowalewski. 2016. DASH: A C++ PGAS Library for Distributed Data Structures and Parallel Algorithms. In *Proceedings of the 2016 IEEE 18th International Conference on High Performance Computing and Communications* (Sydney, NSW, Australia) *(HPCC '16)*. 983–990.

[33] Thierry Gautier, Xavier Besseron, and Laurent Pigeon. 2007. KAAPI: A Thread Scheduling Runtime System for Data Flow Computations on Cluster of Multiprocessors. In *Proceedings of the 2007 International Workshop on Parallel Symbolic Computation* (London, Ontario, Canada) *(PASCO '07)*. 15–23.

[34] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. 1990. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture* (Seattle, Washington, USA) *(ISCA '90)*. 15–26.

[35] Sayan Ghosh, Yanfei Guo, Pavan Balaji, and Assefaw H. Gebremedhin. 2021. RMACXX: An Efficient High-Level C++ Interface over MPI-3 RMA. In *Proceedings of the 2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing* (Melbourne, Australia) *(CCGrid '21)*. 143–155.

[36] Max Grossman, Vivek Kumar, Zoran Budimlić, and Vivek Sarkar. 2016. Integrating Asynchronous Task Parallelism with OpenSHMEM. In *Proceedings of the Third Workshop on OpenSHMEM and Related Technologies* (Baltimore, Maryland, USA) *(OpenSHMEM '16)*. 3–17.

[37] Tasuku Hiraishi, Masahiro Yasugi, Seiji Umatani, and Taiichi Yuasa. 2009. Backtracking-Based Load Balancing. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Raleigh, North Carolina, USA) *(PPoPP '09)*. 55–64.

[38] Seema Hiranandani, Ken Kennedy, and Chau-Wen Tseng. 1992. Compiling Fortran D for MIMD Distributed-Memory Machines. *Commun. ACM* 35, 8 (1992), 66–80.

[39] Torsten Hoefler, James Dinan, Rajeev Thakur, Brian Barrett, Pavan Balaji, William Gropp, and Keith Underwood. 2015. Remote Memory Access Programming in MPI-3. *ACM Transactions on Parallel Computing* 2, 2 (July 2015), 1–26.

[40] Hartmut Kaiser, Thomas Heller, Bryce Adelstein-Lelbach, Adrian Serio, and Dietmar Fey. 2014. HPX: A Task Based Programming Model in a Global Address

Space. In *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models* (Eugene, Oregon, USA) *(PGAS '14)*. 6:1–6:11.

[41] Stefanos Kaxiras, David Klaftenegger, Magnus Norgren, Alberto Ros, and Konstantinos Sagonas. 2015. Turning Centralized Coherence and Distributed Critical-Section Execution on Their Head: A New Approach for Scalable Distributed Shared Memory. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing* (Portland, Oregon, USA) *(HPDC '15)*. 3–14.

[42] Pete Keleher, Alan L. Cox, Sandhya Dwarkadas, and Willy Zwaenepoel. 1994. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In *Proceedings of the USENIX Winter 1994 Technical Conference on USENIX Winter 1994 Technical Conference* (San Francisco, California, USA) *(WTEC '94)*.

[43] Pete Keleher, Alan L. Cox, and Willy Zwaenepoel. 1992. Lazy Release Consistency for Software Distributed Shared Memory. In *Proceedings of the 19th Annual International Symposium on Computer Architecture* (Queensland, Australia) *(ISCA '92)*. 13–21.

[44] Charles H. Koelbel, David Loveman, Robert S. Schreiber, Guy L. Steele Jr., and Mary Zosel. 1993. *High Performance Fortran Handbook*. The MIT Press.

[45] Vivek Kumar, Yili Zheng, Vincent Cavé, Zoran Budimlić, and Vivek Sarkar. 2014. HabaneroUPC++: A Compiler-Free PGAS Library. In *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models* (Eugene, Oregon, USA) *(PGAS '14)*. 5:1–5:10.

[46] Okwan Kwon, Fahed Jubair, Rudolf Eigenmann, and Samuel Midkiff. 2012. A Hybrid Approach of OpenMP for Clusters. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (New Orleans, Louisiana, USA) *(PPoPP '12)*. 75–84.

[47] Jinpil Lee and Mitsuhisa Sato. 2010. Implementation and Performance Evaluation of XcalableMP: A Parallel Programming Language for Distributed Memory Systems. In *Proceedings of the 39th International Conference on Parallel Processing Workshops* (San Diego, California, USA) *(ICPPW '10)*. 413–420.

[48] Kai Li and Paul Hudak. 1989. Memory Coherence in Shared Virtual Memory Systems. *ACM Transactions on Computer Systems* 7, 4 (Nov. 1989), 321–359.

[49] Jimmy Aguilar Mena, Omar Shaaban, Vicenç Beltran, Paul Carpenter, Eduard Ayguade, and Jesus Labarta. 2022. OmpSs-2@Cluster: Distributed Memory Execution of Nested OpenMP-style Tasks. In *Proceedings of the 28th International European Conference on Parallel and Distributed Computing* (Glasgow, Scotland, UK) *(Euro-Par '22)*. 319–334.

[50] Seung-Jai Min, Costin Iancu, and Katherine Yelick. 2011. Hierarchical Work Stealing on Manycore Clusters. In *Proceedings of the Fifth Conference on Partitioned Global Address Space Programming Models* (Galveston Island, Texas, USA) *(PGAS '11)*. 1–10.

[51] Eric Mohr, David A. Kranz, and Robert H. Halstead. 1990. Lazy Task Creation: A Technique for Increasing the Granularity of Parallel Programs. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming* (Nice, France) *(LFP '90)*. 185–197.

[52] Alessandro Morari, Antonino Tumeo, Daniel Chavarría-Miranda, Oreste Villa, and Mateo Valero. 2014. Scaling Irregular Applications through Data Aggregation and Software Multithreading. In *Proceedings of the 28th IEEE International Parallel and Distributed Processing Symposium* (Phoenix, Arizona, USA) *(IPDPS '14)*. 1126–1135.

[53] Jun Nakashima and Kenjiro Taura. 2014. MassiveThreads: A Thread Library for High Productivity Languages. *Concurrent Objects and Beyond* 8665 (Jan. 2014), 222–238.

[54] Jacob Nelson, Brandon Holt, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, and Mark Oskin. 2014. Grappa: A Latency-Tolerant Runtime for Large-Scale Irregular Applications. In *Proceedings of the First International Workshop on Rack-Scale Computing* (Amsterdam, The Netherlands) *(WRSC '14)*. 1–7.

[55] Jacob Nelson, Brandon Holt, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, and Mark Oskin. 2015. Latency-Tolerant Software Distributed Shared Memory. In *Proceedings of the 2015 USENIX Annual Technical Conference* (Denver, Colorado, USA) *(USENIX ATC '15)*. 291–305.

[56] Jaroslaw Nieplocha, Robert J. Harrison, and Richard J. Littlefield. 1996. Global Arrays: A Nonuniform Memory Access Programming Model for High-Performance Computers. *The Journal of Supercomputing* 10, 2 (1996), 169–189.

[57] Robert W. Numrich and John Reid. 1998. Co-Array Fortran for Parallel Programming. *SIGPLAN Fortran Forum* 17, 2 (Aug. 1998), 1–31.

[58] Stephen Olivier, Jun Huan, Jinze Liu, Jan Prins, James Dinan, P. Sadayappan, and Chau-Wen Tseng. 2006. UTS: An Unbalanced Tree Search Benchmark. In *Proceedings of the 19th International Conference on Languages and Compilers for Parallel Computing* (New Orleans, Los Angeles, USA) *(LCPC '06)*. 235–250.

[59] Jeeva Paudel, Olivier Tardieu, and José Nelson Amaral. 2013. On the Merits of Distributed Work-Stealing on Selective Locality-Aware Tasks. In *Proceedings of the 42nd International Conference on Parallel Processing* (Lyon, France) *(ICPP '13)*. 100–109.

[60] Keith H. Randall. 1998. *Cilk: Efficient Multithreaded Computing*. Ph. D. Dissertation. Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology.

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

[62] Tao B. Schardl and I-Ting Angelina Lee. 2023. OpenCilk: A Modular and Extensible Software Infrastructure for Fast Task-Parallel Code. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming* (Montreal, QC, Canada) *(PPoPP '23)*. 189–203.

[63] Joseph Schuchart and José Gracia. 2019. Global Task Data-Dependencies in PGAS Applications. In *High Performance Computing: the 34th International Conference, ISC High Performance 2019* (Frankfurt/Main, Germany) *(ISC '19)*. 312–329.

[64] Shumpei Shiina and Kenjiro Taura. 2019. Almost Deterministic Work Stealing. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Denver, Colorado, USA) *(SC '19)*. 47:1–47:16.

[65] Shumpei Shiina and Kenjiro Taura. 2022. Distributed Continuation Stealing is More Scalable than You Might Think. In *Proceedings of the 2022 IEEE International Conference on Cluster Computing* (Heidelberg, Germany) *(Cluster '22)*. 129–141.

[66] Shumpei Shiina and Kenjiro Taura. 2022. Improving Cache Utilization of Nested Parallel Programs by Almost Deterministic Work Stealing. *IEEE Transactions on Parallel and Distributed Systems* 33, 12 (Dec. 2022), 4530–4546.

[67] Min Si, Huansong Fu, Jeff R. Hammond, and Pavan Balaji. 2021. OpenSHMEM over MPI as a Performance Contender: Thorough Analysis and Optimizations. In *Proceedings of the 8th Workshop on OpenSHMEM and Related Technologies* (Virtual Event) *(OpenSHMEM '21)*. 39–60.

[68] Matthew D. Sinclair, Johnathan Alsop, and Sarita V. Adve. 2015. Efficient GPU Synchronization without Scopes: Saying No to Complex Consistency Models. In *Proceedings of the 48th International Symposium on Microarchitecture* (Waikiki, Hawaii, USA) *(MICRO-48)*. 647–659.

[69] Kenjiro Taura, Jun Nakashima, Rio Yokota, and Naoya Maruyama. 2012. A Task Parallel Implementation of Fast Multipole Methods. In *2012 SC Companion: High Performance Computing, Networking Storage and Analysis–Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems* (Salt Lake City, Utah, USA) *(ScalA' 12)*. 617–625.

[70] Keisuke Tsugane, Jinpil Lee, Hitoshi Murai, and Mitsuhisa Sato. 2018. Multi-Tasking Execution in PGAS Language XcalableMP and Communication Optimization on Many-Core Clusters. In *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region* (Chiyoda, Tokyo, Japan) *(HPC Asia 2018)*. 75–85.

[71] Rio Yokota and Lorena Barba. 2020. GitHub repository: exafmm/exafmm-beta. Retrieved 2022-11-30 from https://github.com/exafmm/exafmm-beta

[72] Rio Yokota, Lorena A. Barba, Tetsu Narumi, and Kenji Yasuoka. 2013. Petascale Turbulence Simulation Using a Highly Parallel Fast Multipole Method on GPUs. *Computer Physics Communications* 184, 3 (2013), 445–455.

[73] Jin Zhang, Xiangyao Yu, Zhengwei Qi, and Haibing Guan. 2022. Falcon: A Timestamp-based Protocol to Maximize the Cache Efficiency in the Distributed Shared Memory. In *Proceedings of the 36th IEEE International Parallel and Distributed Processing Symposium* (Lyon, France) *(IPDPS '22)*. 974–984.

[74] Wei Zhang, Olivier Tardieu, David Grove, Benjamin Herta, Tomio Kamada, Vijay Saraswat, and Mikio Takeuchi. 2014. GLB: Lifeline-Based Global Load Balancing Library in X10. In *Proceedings of the First Workshop on Parallel Programming for Analytics Applications* (Orlando, Florida, USA) *(PPAA '14)*. 31–40.

[75] Zhang Zhang, Jeevan Savant, and Steven Seidel. 2006. A UPC Runtime System Based on MPI and POSIX Threads. In *Proceedings of the 14th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing* (Montbeliard-Sochaux, France) *(PDP '06)*. 195–202.

[76] Yili Zheng, Amir Kamil, Michael B. Driscoll, Hongzhang Shan, and Katherine Yelick. 2014. UPC++: A PGAS Extension for C++. In *Proceedings of the 28th IEEE International Parallel and Distributed Processing Symposium* (Phoenix, Arizona, USA) *(IPDPS '14)*. 1105–1114.

Appendix: Artifact Description/Artifact Evaluation

ARTIFACT DOI

https://zenodo.org/record/8086461

ARTIFACT IDENTIFICATION

Our computational artifacts include the implementation of the Itoyori runtime system and the benchmarks for evaluating Itoyori. These artifacts were used to support our primary contribution of demonstrating the practicality of global-view fork-join programming using the Itoyori platform. The threading layer of Itoyori is based on the uni-address threads implementation (the version used in the evaluation of [59]) with a little modification to insert the release/acquire fences to the fork/join calls. We developed the PGAS layer from scratch over MPI-3 RMA. Our artifact of the PGAS layer demonstrates that the checkout/checkin APIs can be implemented as we have explained in Section 4 and 5, which supports our contribution of proposing software caching mechanisms.

The artifacts used in our evaluation are publicly available in the following GitHub repositories:

- The top-level Itoyori interface and the benchmarks used in this paper:
  https://github.com/s417-lama/ityrbench
- The thraeding layer:
  https://github.com/s417-lama/massivethreads-dm
- The PGAS layer:
  https://github.com/s417-lama/pcas

The benchmarks included in the Itoyori benchmark repository are Cilksort, UTS-Mem (`uts++`), and ExaFMM. Our Cilksort program was ported from the Cilk v5.4.6 repository. The UTS-Mem program was modified from the UTS implementation used in [59] to allocate tree nodes in the global memory region. ExaFMM (in the `exafmm/` directory) is based on the `exafmm-beta` repository (https://github.com/exafmm/exafmm-beta) and modified to run on Itoyori. The MPI version of ExaFMM (in the `exafmm_mpi/` directory) was also included in the `exafmm-beta` repository but we made a little modification to make it run on A64FX CPUs. For the MPI version of ExaFMM, we used MassiveThreads (https://github.com/massivethreads/massivethreads, commit hash: `80809e588ea`) for thread scheduling within each node.

REPRODUCIBILITY OF EXPERIMENTS

Overview

To manage our experiments, we use the Kochi workflow management tool v0.0.1 (https://github.com/s417-lama/kochi/tree/0.0.1). Kochi is designed to minimize the system-specific settings for the node allocation in clusters (e.g., slurm) and manage the version-controlled experimental results in git. For more details about Kochi, see the Kochi tutorial (https://github.com/s417-lama/kochi-tutorial/tree/0.0.1).

In the following, we assume that the Itoyori benchmark repository (https://github.com/s417-lama/ityrbench) is cloned to the local computer and the current directory is the clone (`ityrbench`). First, we need to install the dependencies for the benchmarks (including the threading and PGAS layer) by running the `kochi install` command. To run the benchmarks on remote compute nodes, some additional configurations are needed. In our case, the remote compute nodes are on the Wisteria/BDEC-01 Odyssey supercomputer (`wisteria-o`), whose configuration can be found in the `kochi.yaml` configuration file.

Kochi has three components: jobs, job queues, and workers. A job specifies how to build and run each benchmark, which is enqueued to a job queue. A worker is a process which executes each job by repeatedly popping jobs from a specified job queue. The user first submits jobs to job queues and then launch workers on compute nodes allocated by the system's job manager. The user can also specify a set of jobs with different parameters (e.g., the number of nodes, the input size), which is called a batch job. To reproduce the figures in this paper, we submit batch jobs in each job configuration file (`cilksort.yaml`, `uts++.yaml`, and `exafmm.yaml`).

To submit a batch job, we run:

```
kochi batch <job_config_file> <batch_name>
```

After the batch job submission, we allocate nodes from the system's job manager by running the following command:

```
kochi alloc -m wisteria-o -q node_\$nodes \
  -n 1 -n 2:torus -n 2x3:torus -n 2x3x2:torus ...
```

This command will launch Kochi workers on each node count. The node specification like `2x3x2:torus` is passed to the system's job manager. In our case, it is passed to Fujitsu's job manager (`pjsub`) in the Wisteria/BDEC-01 Odyssey supercomputer, and `2x3x2:torus` indicates that the nodes should be allocated in a torus topology and its 3D topology should be as close to a cube as possible. When the system jobs start, the Kochi workers will also start to pop jobs from the job queue and execute them.

After each job is completed, the result files are automatically gathered into a separate git branch. To collect these results, we need to run the following command on the `ityrbench_artifacts` branch:

```
kochi artifact sync -m wisteria-o
```

The `ityrbench_artifacts` branch now contains the raw experimental results, which will be plotted by the Python scripts in the `plot` directory. The required Python packages can be installed by:

```
pip3 install numpy scipy pandas plotly
```

The Itoyori benchmark repository (https://github.com/s417-lama/ityrbench) contains more information for benchmarking and plotting. In the following, we explain how to submit batch jobs and generate a plot for each figure in this paper.

Workflow for Figure 7

The common workflow in the following is:

(1) Submit batch jobs specified by "Batch job submission."
(2) After all submitted jobs are completed, pull the experimental results by running `kochi artifact sync`.
(3) Run plotting scripts specified by "Plotting."

(4) Open the file specified by "Output file location" by a web browser.

To generate Figure 7:

- Batch job submission:
  `kochi batch cilksort.yaml granularity`
- Estimated total execution time: 1h
- Plotting:
  `python3 ./plot/cilksort/granularity.py`
- Output file location:
  `./figs/cilksort/granularity_wisteria-o.html`

Note that the estimated total execution time depends on how nodes are allocated from the system's job manager. The estimated time in this paper is based on the worst-case scenario, in which all jobs are executed one by one.

Workflow for Figure 8

To generate Figure 8:

- Batch job submission:
  ```
  kochi batch cilksort.yaml serial
  kochi batch cilksort.yaml scale1G
  kochi batch cilksort.yaml scale10G
  ```
- Estimated total execution time: 3h
- Plotting:
  `python3 ./plot/cilksort/scaling.py`
- Output file location:
  `./figs/cilksort/scaling_exectime_wisteria-o.html`

Workflow for Figure 9

The required experimental data are already gathered in the previous workflow (for Figure 8).

To generate Figure 9:

- Plotting:
  `python3 ./plot/cilksort/stats.py`
- Output file location:
  `./figs/cilksort/stats_wisteria-o.html`

Workflow for Figure 10

To generate Figure 10:

- Batch job submission:
  ```
  kochi batch uts++.yaml serial
  kochi batch uts++.yaml T1L
  kochi batch uts++.yaml T1XL
  ```
- Estimated total execution time: 0.5h
- Plotting:
  `python3 ./plot/uts++/scaling.py`
- Output file location:
  `./figs/uts++/scaling_wisteria-o.html`

Workflow for Figure 11

To generate Figure 11:

- Batch job submission:
  ```
  kochi batch exafmm.yaml serial
  kochi batch exafmm.yaml scale1M
  kochi batch exafmm.yaml scale10M
  kochi batch exafmm_mpi.yaml scale1M
  kochi batch exafmm_mpi.yaml scale10M
  ```
- Estimated total execution time: 90h
- Plotting:
  `python3 ./plot/exafmm/scaling.py`
- Output file location:
  `./figs/exafmm/scaling_wisteria-o.html`

Workflow for Table 2

The required experimental data are already gathered in the previous workflow (for Figure 11).

The following script will show the busyness for the execution of the MPI version of ExaFMM.

`python3 ./plot/exafmm/mpi_balance.py`

The idleness is calculated by $(1.0 - \textit{busyness})$, where "busyness" is the value shown by the script for each node count.

ARTIFACT DEPENDENCIES REQUIREMENTS

Hardware A CPU cluster connected with high-performance interconnects (e.g., InfiniBand, Tofu-D Interconnect) is required for experiments. To reproduce the performance, the interconnect should have relatively high bandwidth and support RDMA and network atomic operations. The CPU architecture must be `x86_64` or `aarch64`, which is currently supported by Itoyori. In our experiments, Fujitsu A64FX CPUs (`aarch64`) and Tofu-D Interconnect were used.

OS Linux is required, as Itoyori depends on some Linux-specific system calls.

Software Libraries MPI is required. The implementation of MPI-3 RMA should be sufficiently sophisticated; MPI-3 RMA calls are expected to be directly mapped to RDMA. Otherwise, Itoyori might result in poor performance or a deadlock. Other minor software dependencies will be downloaded during the workflow.

Input Dataset All of our experiments generate synthetic input data at the program startup. No external input data are needed.

ARTIFACT INSTALLATION DEPLOYMENT PROCESS

The above explanation of the workflow is written for general users who might want to use Itoyori on their platform, particularly for supercomputers with a batch job management system. For the artifact evaluation purpose, we provide a step-by-step instruction to setup an experimental environment on the ChameleonCloud platform. Please consult `README.md` of our artifact (https://zenodo.org/record/8086461) for that information.