Electronic Preprint for Journal of Information Processing Vol.30

Regular Paper

Cost-aware Programming on Page-based Distributed Shared Memory

Takato Hideshima[1,a)] Shigeyuki Sato[1,b)] Kenjiro Taura[1]

Received: August 31, 2021, Accepted: January 11, 2022

Abstract: Page-based distributed shared memory (PDSM) is a programming environment on distributed-memory computers that allows to freely allocate shared regions in the virtual address space accessible from any computer. It hides distributed physical memory from programmers and enables shared-memory programming over the uniform virtual address space. PDSM systems are typically equipped with coherent cache to improve performance while hiding communication, but the management cost is considered implementation details and is complex and implicit. Consequently, it is easy to fail in gaining speedup, and it is difficult to perform cost-aware programming to solve it. In this study, we explore cost-aware programming for ArgoDSM, a state-of-the-art PDSM. Particularly, based on the observation that there are three effective measures for reducing PDSM-derived costs: 1) informing PDSM of changes in access patterns to shared regions, 2) inspecting the data to be placed in shared regions, and 3) performing writes with an awareness of the original owner of the shared region, we extend the ArgoDSM with APIs to help in these measures. We performed cost-aware programming on the extended ArgoDSM for benchmark programs, and experimentally showed that PDSM-derived costs can be significantly reduced. The proposed programming measures significantly improve the situation, where the performance is below the sequential performance, and allows to benefit from the scalability of distributed-memory computers under the high-level abstraction of PDSM.

Keywords: page-based distributed shared memory, coherence protocol, performance analysis

1. Introduction

The use of distributed-memory computers becomes practically inevitable when seeking scalability of computing resources. Distributed-memory programming, where the distribution and communication of data are thoroughly under the programmer's responsibility, has more room for performance tuning yet high programming burden. Shared-memory programming, which does not require programmers to describe data distribution and communication, has less burden on programmers. An attempt to bridge the gap between the two is distributed shared memory (DSM), which realizes shared memory on distributed memory.

There are two typical forms of software-implemented DSM (software DSM): first, remote memory access (RMA) [13] or partitioned global address space (PGAS) [29], which involves explicit operations on globally (i.e., among multiple computing nodes) shared memory regions. This makes it easier to tune performance with awareness of the distributed memory layer because the timing at which communication occurs is explicit. However, because it requires dedicated APIs for data access, the code becomes complicated and difficult in interoperability with existing codes that do not use RMA. Second, page-based DSM (PDSM) [2], [4], [6], [8], [14], [16], [22], [26], [31], which implicitly handles communication through per-page coherent caches. Because access to globally shared memory regions (i.e., where coherent caches exist) is captured by page protection, and data is synchronized between nodes transparently to the application, PDSM provides the same ease of coding as normal shared-memory programming and high interoperability with it, however, the cost of coherent actions to synchronize cache data is large, implicit, and intricate. Consequently, it suffers from performance problems, and often underperforms sequential execution without gaining speedup.

Although existing studies often indicate the large cost of coherent actions as a performance problem of PDSM [5], [23], [24], we believe that the implicit and intricate nature of the cost is a critical problem, because if the cause of the cost is clear and the means to control it are provided, then the runtime performance can be improved by devoting more effort to cost-aware programming. Certainly, the natural tendency of RMA programming to be aware of the cost of distributed memory makes it easier to improve performance. Further, when seeking high performance on hardware-implemented DSM in the form of NUMA, it is common to program with NUMA nodes in mind using tools such as `numactl`. On PDSM, however, the cost is implicit and intricate, making such cost-aware programming difficult.

In this study, we explore cost-aware programming for ArgoDSM [16], a state-of-the-art PDSM. Particularly, based on our observations of the coherence protocol of ArgoDSM, and the costs incurred by its various parts, we propose the following three measures to reduce such PDSM-derived costs:

(1) informing PDSM of changes in access patterns to shared regions
(2) inspecting the data to be placed in shared regions

[1] Graduate School of Information Science and Technology, The University of Tokyo, Bunkyo, Tokyo 113–8656, Japan
[a)] hideshima@eidos.ic.i.u-tokyo.ac.jp
[b)] sato.shigeyuki@mi.u-tokyo.ac.jp

(3) performing writes with an awareness of the original owner of the shared region

We also extend the ArgoDSM with APIs to help in these measures. We performed the above measures on the extended ArgoDSM for benchmark programs, and experimentally showed that PDSM-derived costs can be significantly reduced.

The ideas to reduce the cost proposed in this study are commonly found in the context of PGAS. However, the current study investigates the cost quantitatively with the coherence protocol of ArgoDSM in mind, extends the APIs to put the measures into practice, and then shows the experimental evaluation of the effects. To the best of our knowledge, there are no examples of similar efforts in the context of PDSM. Therefore, it is expected that the cost-aware programming practice on PDSM, and the knowledge gained through the performance analysis in this study would be useful in evaluating the performance aspects of RMA and PDSM from the programming perspective.

The contributions of this study are summarized as follows:

- We propose cost-aware programming on PDSM and extend ArgoDSM for this purpose (Section 4). Beyond programming that takes advantage of the same ease of coding as normal shared memory, it provides means and direction for reducing latent costs, taking advantage of the scalability of distributed-memory computers.
- We experimentally evaluated the benefits of cost-aware programming measures on ArgoDSM (Section 5). We confirmed that the proposed measures reduce the occurrence of coherent actions, significantly improving the situation where the performance is below the sequential performance, and allows for gain speedup from distributed-memory computers under high-level abstraction of PDSM.

2. MPI-3 RMA

MPI, a library standard widely used for communication between distributed-memory computers, includes an RMA programming environment designed to leverage the remote direct memory access (RDMA) support of hardware. ArgoDSM, the subject of this study, uses RMA specified in MPI-3 [*1] [13] as its communication backend.

In MPI-3 RMA, the regions from each node that accept RMA from other computing nodes compose a shared region. For RMA operations, the access target is determined by explicitly specifying the shared region, the computing node, and the offset in the region from the node.

The main RMA operations are normal read/write (`MPI_Get`, `MPI_Put`) and atomic operations (Accumulate), where atomicity is guaranteed for each object of basic data types. Combining these with lock operations (`MPI_Win_lock`, `MPI_Win_unlock`), RMA programming without data race is realized.

As mentioned earlier, MPI is a library standard, and there are multiple implementations. We used Open MPI, which makes the most use of RDMA's features [25].

[*1] https://www.mpi-forum.org/mpi-30/

3. ArgoDSM

3.1 Overview

Figure 1 shows an overview of a parallel program using the PDSM system ArgoDSM [16]. ArgoDSM provides applications with a globally shared memory region, and access to this region by applications is captured by a segfault handler prepared by ArgoDSM. Additionally, when applications spontaneously call release or acquire, ArgoDSM manages memory using inter-process communication with MPI-3 RMA. When processes are running on multiple computing nodes, it works as distributed shared memory. Subsequently, we assume that one process is running on each node, and treat a process as a computing node.

Let's take Listing 1 as an example to see where ArgoDSM performs memory management in parallel programming.

In line 10 and below, shared regions are allocated by `argo::conew_array`. Shared-memory programming is realized as one computing node initializes the values in the conditional branch from line 15, and all nodes use them in the main iterative calculation from line 25.

To achieve this, `argo::barrier()` in lines 21, 32, and 37 is spontaneously called by the programmer. This function call has the same meaning as the sequence of release, execution order barrier synchronization, and acquire, and the overall meaning requires that the changes made before the call are visible to all nodes after the call. The reason for having spontaneous calls to release and acquire in addition to normal memory access is that, ArgoDSM adopts the memory model of release consistency [11]. It is known that the results of accesses in this model are equal to those of sequential consistency, a model that fits well with our intuition, when there are no data race.

How it reads or writes to the shared regions is implicitly captured as read-miss or write-miss respectively. Concretely, at the time of writing to the shared regions in lines 17, 19, 29, and 36, write-miss may occur and be captured by ArgoDSM. At the time of reading from the shared regions in lines 27, 33, and 43, read-miss may be captured.

3.2 Cache Coherence Protocol

The purpose of memory management in DSM is to prepare caches for access by applications and to provide coherence to the caches such that changes made will be visible to other computing nodes as required.

The mechanism for this is called coherence protocol. Listings 2, 3, 4, 5, and 6 show pseudo code for the coherence protocol of ArgoDSM, which can be described in terms of what to do at the release, acquire, read-miss, and write-miss times.

Notably, the pseudo code represents the caches with a data layout that does not have consecutive virtual addresses between consecutive pages. In the implementation, `Page` elements are held in a different array, and address continuity is guaranteed when viewed from the application.

The heart of the ArgoDSM coherence protocol is a cache invalidation method called self-invalidation. In this section, we focus on the write-miss process (Listing 6) to explain self-invalidation.

ArgoDSM records the computing nodes that write to a page

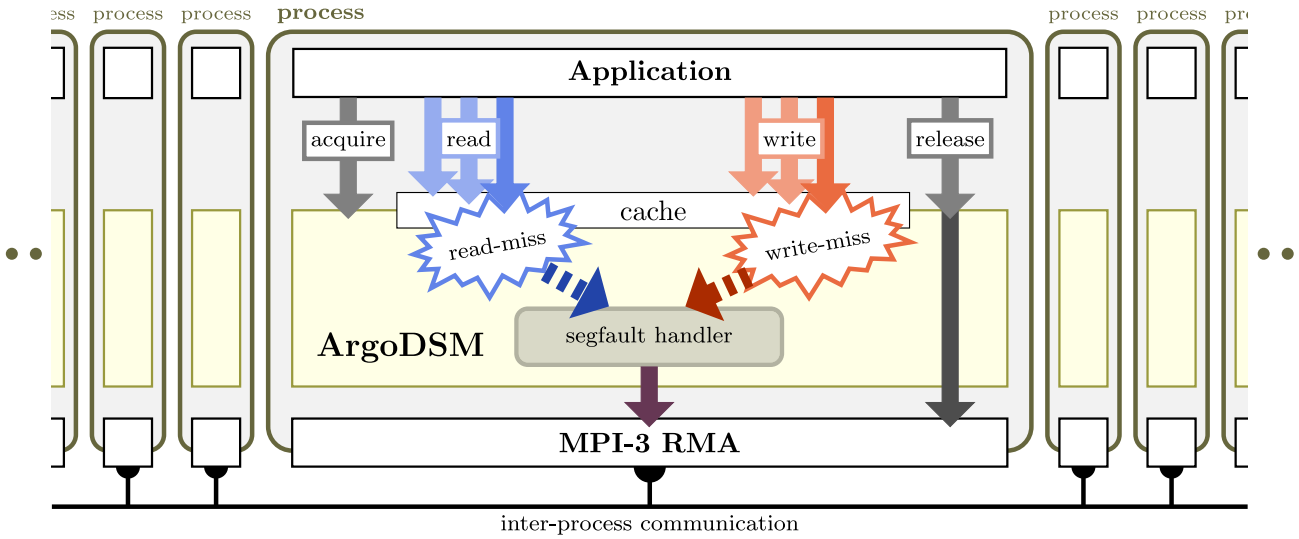


Fig. 1 Overview of ArgoDSM.

(writers) in a shared data structure called a directory (array `dir`). Three states are recorded: no-writer, single-writer, and multiple-writer. In the single-writer state, the ID number of the writer is recorded (line 16), but in the multiple-writer state, all nodes are considered to be writers (line 20). Furthermore, ArgoDSM limits the state transitions to one-way, from no- to single-writer and from single- to multiple-writer.

From abstraction of states and limitation of state transitions, communication for directory can be reduced by caching directory values (specifically, `dir[page_id].writer`) locally. Moreover, when transitioning from single- to multiple-writer, it is sufficient to update only the original directory and the directory cache of one writer recorded in the directory, and other subsequent writers can spontaneously transition to multiple-writer based on their own directory cache. Consequently, the `acquire` function can invalidate the cache without receiving any invalidation message. This is called self-invalidation.

In ArgoDSM, the writer states are coarsely managed as zero, one, and all, and the state transitions are one-way. Although this may appear to be an oversimplification at first glance, the simplification allows for efficient implementation of self-invalidation. Self-invalidation works effectively for repetitive-write situations such that, a page written once by a computing node continues to get written by the same node.

3.3 Overhead

Prior to handling the ArgoDSM overhead, we change the usage of MPI-3 RMA in cache fetch and diff sending. First, we streamline the RMA operations by combining them into a single call instead of invoking them for each interval that constitutes a diff. Moreover, we eliminate the acquisition of the exclusive lock for diff sending by replacing `MPI_Get` used for fetch and `MPI_Put` used for diff sending with accumulate operations that guarantees atomicity, reducing the overhead under communication contention. This change did not make a significant difference in the overhead when communication contention did not occur.

Table 1 summarizes the measured overhead of the four primitives that constitute the coherence protocol of ArgoDSM after the change. The experimental environment is described later in Section 5.

Table 1 Average execution time of ArgoDSM primitives.

primitive	execution time [ns]
release (max. diffs)	180,077
release (no diffs)	5,237
acquire	1,123
read-miss	12,828
write-miss	2,862

For release, we measured the case of calculating and sending the diffs generated in one page. Therein, the pattern of the diff (argument `pat` of `remote_page_write` in Listing 3) greatly affected the overhead. The most time-consuming case we measured was 34.4 times longer than the case where nothing was sent.

Figure 2 shows the relationship between the length of each segment and the time required for release when 2048 bytes of diff is sent in equally divided segments. It is not the total amount of data to be sent, but the number of segments sent that is important, and the more the segments, the larger the overhead. The result of sending 2048 1-byte segments (the leftmost point) is labeled "max. diffs" in Table 1.

Moreover, we measured the following cases: for acquire, eliminating the access privilege of one page without sending diffs; for read-miss, a single read-miss; and for write-miss, a single write-miss on a page that existed in the cache at that time.

Release, which sends out fragmented diffs, has the largest overhead by far, followed by read-miss, which fetches a cache from other computing node. Acquire and write-miss have an order of magnitude smaller overhead.

4. Cost-aware Programming on ArgoDSM

4.1 Basic Policy

As mentioned in Section 3.3, most overhead of ArgoDSM comes from part of fetching caches and sending the changes made to the caches as diffs.

As for diff sending, the overhead per page varies greatly de-

Listing 1 Example of ArgoDSM usage.

```
size_t SHARED_REGION_SIZE;
size_t DIM;
unsigned int NUM_ITERS;

int main() {
  argo::init(SHARED_REGION_SIZE);
  int my_rank = argo::node_id();
  int num_nodes = argo::number_of_nodes();

  using argo::conew_array;
  float *mat = conew_array<float>(DIM * DIM),
    *vec = conew_array<float>(DIM),
    *reduc = conew_array<float>(num_nodes),
    *local = new float[DIM];
  if (my_rank == 0) {
    ifstream ifs{"input"};
    copy_n(istream_iterator<float>{ifs},
      DIM * DIM, mat);
    fill_n(vec, DIM, 1 / sqrt(DIM));
  }
  argo::barrier();   // release + sync + acquire

  size_t beg = DIM * my_rank / num_nodes;
  size_t end = DIM * (my_rank+1) / num_nodes;
  for (unsigned i = 0; i < NUM_ITERS; i++) {
    for (size_t idx = beg; idx < end; idx++)
      local[idx] = inner_product(&vec[0],
        &vec[DIM], &mat[DIM * idx], 0.);
    reduc[my_rank] = inner_product(
      &local[beg], &local[end], &local[beg],
      0.);
    argo::barrier();
    float scale = 1 / sqrt(accumulate(
      &reduc[0], &reduc[num_nodes], 0.));
    for (size_t idx = beg; idx < end; idx++)
      vec[idx] = local[idx] * scale;
    argo::barrier();
  }

  if (my_rank == 0) {
    ofstream ofs{"output"};
    for (size_t idx = 0; idx < DIM; idx++)
      ofs << vec[idx] << endl;
  }

  argo::codelete_array(mat);
  argo::codelete_array(vec);
  argo::codelete_array(reduc);
  delete[] local;
  argo::finalize();
}
```

pending on the diff pattern that occurs in the page. However, because the variation is largely owing to the continuity of changes in bytes, rather than the total amount of changes, there is almost no chance for application-side devices.

Therefore, our focus is to reduce the total number of pages to fetch and send the diffs. We propose the following three concrete policies to achieve this.

First, to reset the cache management state when the access pattern to the shared regions changes (Section 4.2). As described in Section 3.2, self-invalidation by ArgoDSM is effective when a computing node writes the same set of pages repetitively. However, once this condition is no longer satisfied, it increases unnecessary cache invalidation and future fetches. Therefore, by conveying the knowledge of access pattern changes from the application side, we mitigate this disadvantage.

Second, to inspect carefully the objects to be placed on the

Listing 2 Data types pertaining to caches.

```
struct Page {
  bool dirty;
  int8_t data[PAGESIZE];
  int8_t twin[PAGESIZE];
};

map<unsigned int, Page&> cache;

struct DirectoryEntry {
  const int home;
  const intptr_t origin_addr;
  int writer;
  void global_sync(); // one-sided comm.
};

const int NO_WRITER, ANY_WRITER;

DirectoryEntry dir[NUM_PAGES]; // globally shared

struct IntraPageInterval {
  size_t offset;
  size_t length;
};
using Pattern = vector<IntraPageInterval>;
```

Listing 3 Processing in release.

```
void remote_page_write(
  int dst_rank, intptr_t remote_ptr,
  int8_t local[PAGESIZE], Pattern pat);
Pattern diff_page(
  int8_t cur[PAGESIZE], int8_t old[PAGESIZE]);

void release() {
  for (auto& [page_id, page] : cache) {
    if (page.dirty) {
      mprotect(page.data,PAGESIZE, PROT_READ);
      remote_page_write(dir[page_id].home,
        dir[page_id].origin_addr, page.data,
        diff_page(page.data, page.twin));
      page.dirty = false;
    }
  }
}
```

Listing 4 Processing in acquire.

```
void acquire() {
  release();
  map<unsigned int, Page&> new_cache;
  for (auto& [page_id, page] : cache) {
    if (dir[page_id].writer == my_rank
        || dir[page_id].writer == NO_WRITER)
      new_cache[page_id] = page;
    else
      mprotect(page.data,PAGESIZE, PROT_NONE);
  }
  cache.swap(new_cache);
}
```

Listing 5 Processing in read-miss.

```
void remote_page_read(
  int dst_rank, intptr_t remote_ptr,
  int8_t local[PAGESIZE], Pattern pat);

void read_miss(unsigned int page_id) {
  assert(dir[page_id].home != my_rank);
  auto& page = get_page(page_id);
  remote_page_read(dir[page_id].home,
    dir[page_id].origin_addr, page.data,
    Pattern{{0, PAGESIZE}});
  cache[page_id] = page;
  mprotect(page.data, PAGESIZE, PROT_READ);
}
```

Listing 6 Processing in write-miss.

```
void write_miss(unsigned int page_id) {
  if (dir[page_id].home == my_rank) {
    auto& page = get_page(page_id);
    mprotect(page.data, PAGESIZE,
      PROT_READ | PROT_WRITE);
  } else {
    if (!cache.contains(page_id))
      read_miss(page_id);
    auto& page = cache[page_id];
    page.dirty = true;
    memcpy(page.twin, page.data, PAGESIZE);
    mprotect(page.data, PAGESIZE,
      PROT_READ | PROT_WRITE);
  }
  if (dir[page_id].writer == NO_WRITER) {
    dir[page_id].writer = my_rank;
    dir[page_id].global_sync();
  } else if (dir[page_id].writer != my_rank) {
    if (dir[page_id].writer != ANY_WRITER) {
      dir[page_id].writer = ANY_WRITER;
      dir[page_id].global_sync();
    }
  }
}
```

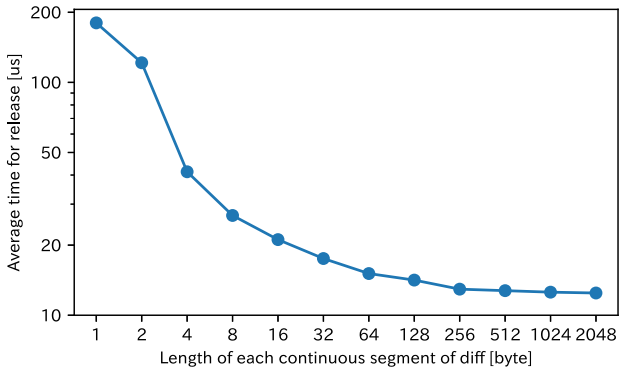


Fig. 2 Relationship between the occurrence pattern of diff and the time required for sending.

shared regions (Section 4.3). Objects known not to be updated for a long time are not worth placing on shared regions in practice. We can reduce fetch and diff sending by privatizing, that is, releasing such shared objects from the management of PDSM and copying them into the local memory of each node.

Third, write to the shared region on the node that has the origin of the cache, called home node (Section 4.4). On home nodes, we take advantage of the fact that we can write directly to the origin and reduce the diff sending.

Listing 7 Example of the reset API usage.

```
if (my_rank == 0) {
  // initialization
}
argo::reset_protocol_state();

size_t beg = DIM * my_rank / num_nodes;
size_t end = DIM * (my_rank + 1) / num_nodes;
for (unsigned i = 0; i < NUM_ITERS; i++) {
  // main calculation
}
```

Listing 8 Example of privatization.

```
float *mat = new float[DIM * DIM],
  /* ... */;
{
  ifstream ifs{"input"};
  copy_n(istream_iterator<float>{ifs},
    DIM * DIM, mat);
}
if (my_rank == 0) {
  fill_n(vec, DIM, 1 / sqrt(DIM));
}
```

4.2 Reset API

We add a function `argo::reset_protocol_state` as an API to reset the cache management state. It has the effect of sending diffs for all caches and then discarding them, and resetting `DirectoryEntry::writer` mentioned in Section 3.2 to `NO_WRITER`.

The resulting meaning in the memory model is equal to that of `argo::barrier` described in Section 3.1. The signature is also the same except for the name, and it is used to replace `argo::barrier` which gets called when the access pattern changes.

Listing 7 shows an example of the use of reset API based on Listing 1, replacing the `argo::barrier` with the reset API at the timing when the access pattern changes from the initialization phase where one computing node writes to the entire array to the iterative calculation phase where all nodes write to each of the equally divided arrays.

4.3 Privatization

Listing 8 shows an example of privatization to reduce the number of shared objects in Listing 1. The array `mat`, which was placed on the shared region in Listing 1, but did not change during the iterations, is relocated to the normal memory region and initialized on every computing node.

In this way, data that is not to be updated frequently is not hard to share without PDSM. This can be achieved by prior replication or use of shared file systems if file input is used, or by calculation on all nodes, if the results of calculations in the program are to be stored. The influence of these measures on the critical path of the program is small, and we expect that the overhead reduction of ArgoDSM to outweigh it.

Listing 9 Example of the on-home API usage.

```
for (unsigned i = 0; i < NUM_ITERS; i++) {
  // ...
  argo::barrier();
  float scale = /* ... */;
  float* vec_iter = &vec[0];
  const float* vec_end = &vec[DIM];
  while (vec_iter != vec_end) {
    const float* [beg, end] = argo::backend::
      subrange_local(vec_iter, vec_end);
    for (auto idx = beg - vec;
         idx < end - vec; idx++) {
      vec[idx] = local[idx] * scale;
    }
  }
}
```

As privatization replicates data onto local memory, it generally takes spatial cost. However, existing studies [3], [5], [16] on software DSM systems pursuing runtime performance adopted settings to allocate sufficiently large cache space to accommodate all shared regions. Under such settings, privatization does not increase total space usage.

4.4 On-home API

We add `template<class T> pair<T*, T*> argo::backend::subrange_local(T* l, T* r)` as an API to obtain a subrange of a shared region that satisfies the condition that the node is the home node. The argument represents an interval `[l, r)` in an array of type `T[]` on the shared region. If there are regions in this interval that satisfy the condition, the closest one `[a, b)` to the beginning of the array is returned as `pair{a, b}`. If not, `pair{r, r}` is returned.

Listing 9 shows an example of using the on-home API based on Listing 1. Instead of the even division used in Listing 1, it is updated based on the division of home nodes.

The same access pattern can be achieved with a simple API that receives a single pointer, and returns whether it points to the region where the current node is the home node, but in that case, the API would have to be called with all the elements of the array. Thus, the API that handles intervals makes sense in terms of cost. The number of iterations is reduced, and the number of API calls is also reduced. Additionally, the elimination of one conditional branch from the innermost loop makes compiler optimization easier.

5. Evaluation

We experimentally confirmed the following three points:

- As described in Section 4.2, using the reset API reduces the number of cache fetches, which can shorten the execution time of the program (Section 5.2.1).
- As described in Section 4.3, programming of privatizing appropriate shared objects reduces the number of cache fetches and diff sending, reducing the execution time of the program (Section 5.2.2).
- As described in Section 4.4, using the on-home API to make changes to the shared data on the home node reduces the number of diff sending, which can shorten the execution time of the program (Section 5.2.3).

5.1 Experimental Setup

We used an extended ArgoDSM based on the publicly available implementation [*2] (commit ea4c504) with the changes and additions to the APIs described in Sections 3.3, 4.2, and 4.4. We implemented the following two workloads, and measured the total number of pages subject to fetch and diff sending, and the time taken for the sum of the main computation and the initialization of the data used in it.

- CG class B from NAS Parallel Benchmark (NPB)
- Himeno Benchmark class L

For each workload, we created a total of eight different benchmarks, varying whether or not the three measures under evaluation were performed, and measured them with different numbers of computing nodes running in parallel. The parallelism within each node was set to 1. The time taken for sequential execution without ArgoDSM was also measured to serve as a basis for comparison of execution time.

We determined the following three points in the benchmark implementation on ArgoDSM.

First, division of the processing among the computing nodes. In the benchmarks using the on-home API, all changes to the shared data were divided to be made on the home node, and in the other benchmarks, each loop was statically divided into equal parts.

Second, which objects were to be managed by PDSM. In the benchmarks, where we did not privatize the shared objects, after the division of the process described above, all the objects where the value written by a computing node could be read by other nodes were defined as shared objects, while the others were not. Notably, data that was treated as an array in the original implementation of the workload was considered on a per-array rather than per-element basis. In the benchmarks where we privatized the shared objects, we limited the shared objects to those written during the main computation phase too, and excluded those written only during the initialization phase. For the excluded objects, the initialization values were calculated on all nodes.

Last, the size of the shared space and the space where the caches are placed. These were both the same size, and were set to be the smallest possible size within which all the shared data could fit (**Table 2**). The shared space was divided equally by the number of computing nodes (fractions smaller than the page size 4 kiB were rounded up), and each node became the home node of one of these.

We used Oakbridge-CX, a supercomputer operated by the Information Technology Center of the University of Tokyo, as the experimental environment. Its specifications are listed in **Table 3**.

The ArgoDSM system and benchmarks were compiled using `g++` 7.5.0 with the optimization option `-O3`. Communication between computing nodes was done using Open MPI 4.0.5.

[*2] https://github.com/etascale/argodsm

Table 4 Relationship between the number of fetches N_f, the number of diff sending N_d, and the execution time Δt.

Workload	Reset	Privatized	On-home	N_f max. (avg.)	N_d max. (avg.)	Δt [sec]
Himeno				666 k (637 k)	516 k (93 k)	143.5
Himeno			✓	7.5 M (584 k)	444 k (14 k)	228.6
Himeno		✓		174 k (167 k)	99 k (81 k)	35.3
Himeno		✓	✓	140 k (88 k)	64 k (2.0 k)	10.4
Himeno	✓			472 k (50 k)	516 k (93 k)	122.7
Himeno	✓		✓	928 k (120 k)	444 k (14 k)	98.9
Himeno	✓	✓		64 k (15 k)	99 k (81 k)	26.5
Himeno	✓	✓	✓	140 k (88 k)	64 k (2.0 k)	9.8
NPB CG				2.8 M (25 M)	2.3 M (96 k)	204.4
NPB CG			✓	78 M (2.5 M)	2.2 M (74 k)	1762.1
NPB CG		✓		296 k (285 k)	27 k (24 k)	17.6
NPB CG		✓	✓	299 k (148 k)	3.9 k (3.8 k)	18.0
NPB CG	✓			332 k (293 k)	2.3 M (96 k)	116.3
NPB CG	✓		✓	26 M (824 k)	2.2 M (74 k)	701.1
NPB CG	✓	✓		296 k (285 k)	27 k (24 k)	16.9
NPB CG	✓	✓	✓	299 k (148 k)	3.9 k (3.8 k)	18.2

Table 2 Size of the shared space requested for ArgoDSM.

Workload	w/ Privatization	w/o Privatization
Himeno	65,537 pages	459,753 pages
NPB CG	295 pages	47,027 pages

Table 3 Specifications of experimental environment.

CPU	Intel® Xeon® Platinum 8280
CPU frequency	2.7 GHz
#CPUs (#cores)	2 (28 + 28)
Memory	192 GiB
Interconnect	Intel® Omni-Path (100 Gbps)
Network Topology	Full-bisection Fat Tree

Table 5 Ratio of speed improvement r by making each programming effort.

(a) Reset API

Workload	Privatized	On-home	r
Himeno			1.17x
Himeno		✓	2.31x
Himeno	✓		1.33x
Himeno	✓	✓	1.06x
NPB CG			1.76x
NPB CG		✓	2.51x
NPB CG	✓		1.03x
NPB CG	✓	✓	0.99x

(b) Privatization

Workload	Reset	On-home	r
Himeno			4.06x
Himeno		✓	21.90x
Himeno	✓		4.63x
Himeno	✓	✓	10.02x
NPB CG			11.61x
NPB CG		✓	97.56x
NPB CG	✓		6.85x
NPB CG	✓	✓	38.46x

(c) On-home API

Workload	Reset	Privatized	r
Himeno			0.63x
Himeno		✓	3.38x
Himeno	✓		1.24x
Himeno	✓	✓	2.68x
NPB CG			0.12x
NPB CG		✓	0.97x
NPB CG	✓		0.17x
NPB CG	✓	✓	0.93x

5.2 Experimental Results

Table 4 shows the results of the number of fetches, number of diff sending, and execution time for all 16 benchmarks in 32-node runs. **Table 5** summarizes the change in performance when each programming effort is made.

Figure 3 shows the speedup of the parallel execution for each workload, based on sequential execution without ArgoDSM. Note that the two missing results in Fig. 3 (b) with 64 computing nodes are owing to the fact that the programs did not finish after 30 minutes of execution, thus, it was stopped mid-way before getting the results.

5.2.1 Reset API

Looking at the impact of using the reset API when moving from the initialization phase to the computation phase, the performance improved for six of the eight benchmarks, as listed in Table 5 (a). For the cases where performance improved, a reduction in the number of fetches was observed. The largest improvement was achieved when the reset API was used on the NPB CG without privatizing the shared objects and with the on-home API, the execution speed increased by 2.51 times, and the maximum number of fetches among all computing nodes reduced by 67%.

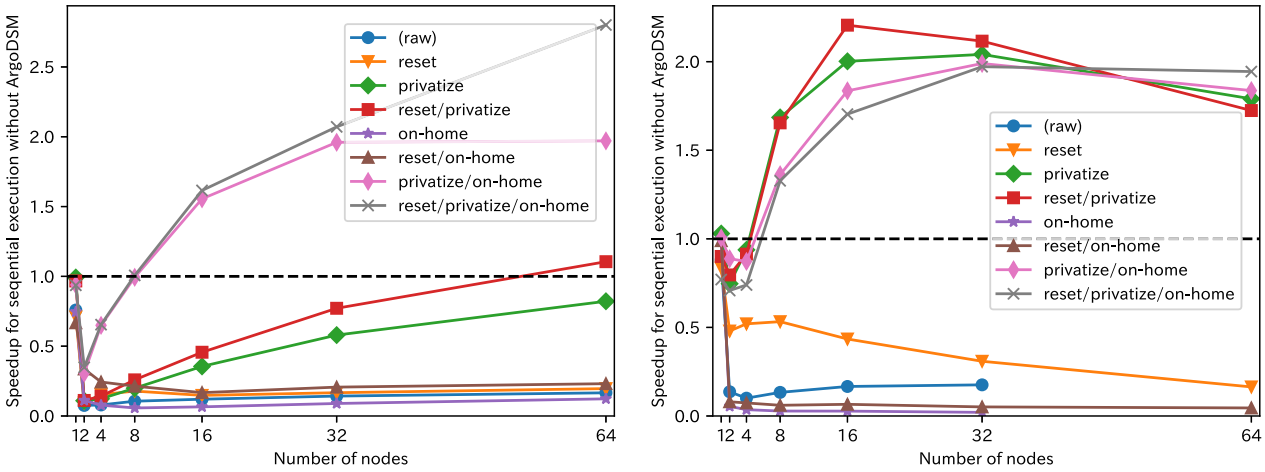
In the two privatized cases of NPB CG, no access to the shared data arose before the computation phase. Therefore, the use of the reset API increased the number of fetches slightly that, it did not appear in Table 4, and the execution speed of one of them was 1% slower and the other was 3% faster.

This indicates that the number of fetches can be greatly reduced and the execution time can be shortened by using the reset API in-line with the large changes in access patterns between the initialization and computation phases. It also shows that even when the benefits are not obtained, the disadvantages are small.

Because the reset API unconditionally discards the entire caches, depending on how it is called, fetches that do not occur in normal barrier synchronization may occur. In our case, the caches in the node responsible for initialization were destroyed, causing a fetch on the next access. From the experimental results, we can say that the limitations of this do not outweigh the benefits in this setup, but this is not the case for other uses.

5.2.2 Privatization

As summarized in Table 5 (b), the shared object reduction improves the performance in all eight pairs of benchmarks. In all of them, both the number of fetches and the number of diff sending are reduced. The reduction in the number of fetches ranges from 74 to 99.8%, comparing the maximum number of fetches



(a) Himeno (b) NPB CG

Fig. 3 Speedup of the parallel execution for sequential execution without ArgoDSM.

per computing node; similarly, the reduction in the number of diff sending ranges from 81 to 99.8%, and the change in execution speed ranges from 4.06 to 97.56x.

Table 6 lists the numbers of fetches and diff sending in the initialization phase and the computation phase for the two workloads with the use of reset API and no use of on-home API and with/without privatization.

There are two reasons for extracting these cases. One is that the effects of privatization and the use of the reset API are not orthogonal, as presented in the cases of NBP CG in the previous section. To examine effects specific to privatization more clearly, it is necessary to compare the results under the use of the reset API. The other is that benchmarks using the on-home API are not suitable for comparison because privatization changes the home assignments of the regions of shared objects and consequently changes computational patterns.

Table 6 indicates that most of the reduction in fetches and diff sending occurred in the initialization phase under privatization.

As shown from Fig. 3, the speed of multi-node execution eventually exceeded that of the original serial execution in seven out of the eight benchmarks with privatization. The remaining one also approached the serial performance, as the number of nodes for parallel execution increased. Contrary, in the eight benchmarks without privatization, multi-node execution was slower than the original serial execution.

This shows that privatization reduces the number of fetches and diff sending, particularly in the initialization phase, which was less parallelized, and that this allows programs on ArgoDSM to benefit from the increase in the number of computing nodes in terms of execution speed, leading to programs on ArgoDSM outperforming sequential programs.

Notably, scalability depended on workloads. NPB CG had poor spatial locality of access to shared regions. Consequently, the nodes participating in the computation accessed all pages containing (unprivatized) shared objects. It bounded reduction in the number of fetches. Probably because of it, the parallel performance peaked around 16 nodes, as shown in Fig. 3 (b). Contrary, Himeno had good locality and each node fetched only a portion of pages. The parallel performance improved smoothly up to 64 nodes.

5.2.3 On-home API

As listed in Table 5 (c), the performance improvement is in the three benchmarks that combine the on-home API and one or more other measures in Himeno.

In all eight pairs, the use of the on-home API reduces the number of diff sending. The reduction ranges from 22 to 97% comparing the average of 32 nodes. As for the number of fetches, it decreases in two pairs when comparing the maximum among 32 computing nodes, and in five pairs when comparing the average, but increases in the other pairs.

Figure 3 (a) shows that in Himeno the two benchmarks that use the on-home API along with privatization are faster than the other benchmarks by a large margin as the number of computing nodes increases. Among these, the one that also uses the reset API, which is summarized in Table 4 to increase both the maximum and average number of fetches with the on-home API, was the fastest of all the benchmarks when run on 64 nodes, running 2.80 times faster than sequential execution without ArgoDSM. As for the execution speed of NPB CG, Fig. 3 (b) shows that the benchmark without the on-home API was the fastest among all the measurements, but when running on 64 nodes, the top two benchmarks used the on-home API.

The above results confirmed that programming with the on-home API greatly reduced the diff sending. Although the number of fetches was not always reduced but sometimes increased, it greatly improved the execution speed even when it was increased.

From here on, we try to explain as much as possible on how the impact of using the on-home API varies with workloads and benchmarks. Notably, in programming with the on-home API, the computations that each computing node was responsible for and the data it accessed were determined by how they are assigned homes of the shared regions. In this experiment, the home assignment was as follows. In both workloads, two arrays of equal size account for the majority of the data that was modified during the computation phase, and in the privatized benchmarks, the home nodes of each array were occupied by approximately

Table 6 Change in the number of fetches N_f and the number of diff sending N_d by phase.

				Initialization phase		Computation phase	
Workload	Reset	Privatized	On-home	N_f max. (avg.)	N_d max. (avg.)	N_f max. (avg.)	N_d max. (avg.)
Himeno	✓			444 k (14 k)	444 k (14 k)	38 k (36 k)	82 k (79 k)
Himeno	✓	✓		63 k (2.0 k)	63 k (2.0 k)	13 k (13 k)	82 k (79 k)
NPB CG	✓			42 k (1.3 k)	2.3 M (71 k)	302 k (291 k)	27 k (25 k)
NPB CG	✓	✓		0 (0)	0 (0)	296 k (285 k)	27 k (24 k)

half of the computing nodes. In the non-privatized benchmarks, three nodes were responsible for each, when running Himeno on 32 nodes, and the same single node was responsible for both when running NPP CG on the same number of nodes.

Based on this, the following explanation can be given for the experimental results. The change in the average number of fetches in the 32 nodes depends on which of the following is larger: the effect of decreasing the number of fetches of the data being modified, or the effect of increasing the number of fetches, because the node where the data is modified and the node where it is read out are more likely to be different than if each loop is divided equally each time. The maximum value tends to increase more than the average value because access to the shared objects is concentrated on some nodes. Notably, we were able to confirm that the statement in this paragraph is correct for Himeno, for which the analysis of access patterns was relatively easy, but not for NPP CG.

The various changes in execution speed can be explained by the various combined results of the following factors: the faster speed owing to the reduction of diff sending, which is the costliest of all coherent actions in ArgoDSM, the performance degradation owing to the worse load balancing of computation, and the increase or decrease in fetches, which has a different direction of impact for each benchmark. The reason why there was no performance improvement when using the on-home API in the benchmarks without privatization in Fig. 3 is that, the computation and fetches were unevenly distributed among some nodes. Himeno showed greater improvement than NPB-CG in privatized benchmarks with use of the on-home API, because Himeno is less affected by poor load balancing of computation, as it requires less computation per write to the shared data.

6. Related Work and Discussion

6.1 Partitioned Global Address Space

Partitioned global address space (PGAS) [29] is a type of cost-aware programming style for distributed shared memory environments. Its core idea is to distinguish between globally shared objects and locally stored objects. Privatization (Section 4.3) and the on-home API (Section 4.4) proposed in this study are PGAS-like ideas. In particular, the fact that the on-home API is a function that returns an array interval, rather than a pointer predicate, follows the design of the PGAS languages Chapel [7] and X10 [27].

There are approximately two categories of PGAS. One is PGAS languages that provide language-level abstraction, with UPC [28], Chapel [7], and X10 [27] being typical examples. The other is RMA libraries, which provide one-sided access to remote memory. MPI-3 RMA [13], OpenSHMEM [*3], and UPC++ [30] are typical examples, with MPI-3 RMA being the most widely used API standard. Although PGAS languages provide high-level abstraction while RMA libraries provide low-level abstraction, both have in common that they distinguish globally shared objects from others in operations. Thus, programmers can be aware of the point at which communication occurs.

To avoid interference with programmers' awareness, PGAS runtime systems are basically designed without coherent caches such as PDSM. Although caches specific to the communication of each PGAS runtime system have been studied [10], [12], and in some cases, prefetching to caches is provided as APIs [10], [17]. The reset API proposed in this study (Section 4.2) is an API based on the ArgoDSM's coherence protocol and it is not addressed by existing PGAS systems.

Among the existing PGAS systems, the closest one to the extended ArgoDSM in this study is UPC. In UPC, pointers to shared objects are distinguished by a type modifier. However, access through pointers itself can be done in the same way as normal pointers. Thus, compared to other PGAS languages such as Chapel and X10, communication occurs more implicitly in source codes. This is a property common to PDSM, including ArgoDSM. Consequently, it also shares with PDSM the disadvantage that the cost model is complex and difficult to understand [18]. Further, UPC provides an operator to check whether the pointer points to an object that the node holds. This is equivalent to the on-home API not adopted in this study, which was discussed in Section 4.4.

UPC++ is a port of the concept of UPC to the form of a C++ library. Because it is a library, it does not provide language-level abstractions, but it provides abstractions based on the rich language features of C++. However UPC++ is not a mere library implementation of UPC; its design policy is different. In UPC++, pointers to shared objects are distinguished from ordinary pointers, and access by pointers requires the use of dedicated APIs. Therefore, UPC++ abandons the implicit communication feature of UPC, and the APIs are designed such that, all operations related to communication are explicit. Consequently, UPC++ can achieve higher performance than UPC, because it can reduce overhead more explicitly, but at the expense of ease of programming [19].

The extended ArgoDSM proposed in this study can be regarded as a kind of library implementation of UPC. It differs from UPC++ in its direction, however, in that, it focuses on realizing implicit communication without a dedicated compiler through an implementation technique based on page protection, without sacrificing ease of programming.

6.2 Software Distributed Shared Memory

Distributed shared memory (DSM) is the implementation of shared memory on top of distributed-memory environments, and

[*3] http://www.openshmem.org/

particularly those implemented by software are called software DSM. Software DSM was widely studied in the 1990s [2], [4], [6], [8], [15], [26], [31], mainly in the fields of architecture and operating systems, after the appearance of IVY [22], the first implementation. The most widely studied implementation of software DSM at that time was page-based DSM (PDSM), which manages memory based on page protection functionality.

Then, by end of the 90s, PGAS became popular, particularly in HPC field, and in the 2000s, software DSM research went downhill. The situation where PGAS is more popular than software DSM has continued to the present. However, PGAS is merely a form of software DSM. Certainly, region-based DSM libraries [15] are technically difficult to distinguish from RMA libraries with caches [5], [12].

Today's high-performance network devices are equipped with a feature called RDMA, which bypasses the OS kernel and transfers data with zero-copy. ArgoDSM [16], which is the subject of this study, is a modern PDSM designed on the premise of RDMA. However, the coherence protocol at its core is a classical method proposed in the 1990s [21], [31]. In this sense, ArgoDSM is an old and new PDSM.

Because ArgoDSM is a simple prototype implementation, it is easy to add various extensions. Certainly, Anevlavis [3] prepares multiple home assignment policies and compares their performance in a collection of benchmarks [*4] for HPC that includes Himeno and NPB CG. However, rather than implementing the home allocation functionality as an API that can be called at runtime, ArgoDSM has been extended to specify the policies as runtime parameters.

There are several PDSM implementation techniques that have been studied in the past that are worth revisiting, as the case with the coherence protocol employed by ArgoDSM. For example, shifting the home based on the workload [8], using different coherence protocols with different characteristics [1], and using different communication methods [14]. The design that allows users to select consistency models and protocols through APIs can be seen in Munin [6]. However, designing easy-to-use APIs for these features and implementing efficient integration is not obvious, and designing APIs based on computational patterns, as in NanosDSM [9] designed to cooperate with an OpenMP runtime, will be crucial.

7. Conclusion

In this study, we propose three cost-aware programming measures on top of the state-of-the-art PDSM system ArgoDSM: informing PDSM of changes in access patterns to shared regions, inspecting the data to be placed in shared regions, and performing writes with an awareness of the original owner of the shared region. We extend ArgoDSM for this purpose. We experimentally confirmed that the proposed measures reduce the number of coherent actions, significantly improving the situation where the parallel performance is below the sequential performance, and make it possible to benefit from speedup from distributed-memory computers under the high-level abstraction provided by PDSM.

Future work may be directed toward integrating methods that have been confirmed to be effective in existing studies on PDSM, by giving them appropriate APIs, as discussed in Section 6. In particular, we expect that extending the allocation API such that the home assignment of distributed arrays can be adapted to the workload would greatly improve the performance by increasing the utility of the on-home API. Certainly, in preliminary experiments that mimic the ideal home assignment of arrays, we were able benefit the good locality of Himeno and scale up to 14.4 times the sequential performance on 64 nodes. For problems that do not scale well, such as NPB CG, it is promising to introduce an API that can create distributed data structures that leverage the knowledge of the application side [24]. With such extensions, we expect that performance comparable to UPC, which is based on a dedicated compiler, can be achieved in the form of C++ runtime library.

[*4] https://github.com/ioanev/argodsm-benchmarks

References

[1] Adve, S.V., Cox, A.L., Dwarkadas, S., Rajamony, R. and Zwaenepoel, W.: A Comparison of Entry Consistency and Lazy Release Consistency Implementations, *Proc. 2nd International Symposium on High-Performance Computer Architecture*, *HPCA '96*, pp.26–37 (online), DOI: 10.1109/HPCA.1996.501171 (1996).

[2] Amza, C., Cox, A.L., Dwarkadas, S., Keleher, P.J., Lu, H., Rajamony, R., Yu, W. and Zwaenepoel, W.: ThreadMarks: Shared Memory Computing on Networks of Workstations, *IEEE Computer*, Vol.29, No.2, pp.18–28 (online), DOI: 10.1109/2.485843 (1996).

[3] Anevlavis, I.: A Study of Page-Based Memory Allocation Policies for the Argo Distributed Shared Memory System, Master's Thesis, Uppsala University, Department of Information Technology (2021).

[4] Bershad, B.N., Zekauskas, M.J. and Sawdon, W.A.: The Midway distributed shared memory system, *Digest of Papers. Compcon Spring*, pp.528–537, IEEE (online), DOI: 10.1109/CMPCON.1993.289730 (1993).

[5] Cai, Q., Guo, W., Zhang, H., Agrawal, D., Chen, G., Ooi, B.C., Tan, K.-L., Teo, Y.M. and Wang, S.: Efficient Distributed Memory Management with RDMA and Caching, *Proc. VLDB Endowment*, Vol.11, No.11, pp.1604–1617 (online), DOI: 10.14778/3236187.3236209 (2018).

[6] Carter, J.B., Bennett, J.K. and Zwaenepoel, W.: Implementation and Performance of Munin, *Proc. 13th ACM Symposium on Operating Systems Principles*, *SOSP '91*, pp.152–164, ACM (online), DOI: 10.1145/121132.121159 (1991).

[7] Chamberlain, B.L.: Chapel, *Programming Models for Parallel Computing*, Balaji, P. (Ed.), MIT Press, chapter 6, pp.129–159 (2015).

[8] Chung, J., Seong, B.H., Park, K.H. and Park, D.: Moving Home-Based Lazy Release Consistency for Shared Virtual Memory Systems, *Proc. 1999 International Conference on Parallel Processing*, *ICPP '99*, pp.282–290, IEEE (online), DOI: 10.1109/ICPP.1999.797414 (1999).

[9] Costa, J.J., Cortes, T., Martorell, X., Ayguade, E. and Labarta, J.: Running OpenMP applications efficiently on an everything-shared SDSM, *Journal of Parallel and Distributed Computing*, Vol.66, No.5, pp.647–658 (online), DOI: 10.1016/j.jpdc.2005.06.018 (2006).

[10] Ferguson, M.P. and Buettner, D.: Caching Puts and Gets in a PGAS Language Runtime, *Proc. 2015 9th International Conference on Partitioned Global Address Space Programming Models*, pp.13–24, IEEE (online), DOI: 10.1109/PGAS.2015.10 (2015).

[11] Gharachorloo, K., Lenoski, D., Laudon, J., Gibbons, P.B., Gupta, A. and Hennessy, J.L.: Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors, *Proc. 17th Annual International Symposium on Computer Architecture*, *ISCA '90*, pp.15–26, ACM (online), DOI: 10.1145/325164.325102 (1990).

[12] Girolamo, S.D., Vella, F. and Hoefler, T.: Transparent Caching for RMA Systems, *Proc. 2017 IEEE International Parallel and Distributed Processing Symposium*, pp.1018–1027 (online), DOI: 10.1109/IPDPS.2017.92 (2017).

[13] Hoefler, T., Dinan, J., Thakur, R., Barrett, B., Balaji, P., Gropp, W. and Underwood, K.: Remote Memory Access Programming in MPI-3, *ACM Trans. Parallel Computing*, Vol.2, No.2, pp.9:1–9:26 (online), DOI: 10.1145/2780584 (2015).

[14] Iosevich, V. and Schuster, A.: Software Distributed Shared Memory: A VIA-based implementation and comparison of sequential consistency with home-based lazy release consistency, *Software - Practice and Experience*, Vol.35, No.8, pp.755–786 (online), DOI: 10.1002/spe.656 (2005).
[15] Johnson, K.L., Kaashoek, M.F. and Wallach, D.A.: CRL: High-Performance All-Software Distributed Shared Memory, *Proc. 15th ACM Symposium on Operating Systems Principles*, *SOSP '95*, pp.213–226, ACM (online), DOI: 10.1145/224056.224073 (1995).
[16] Kaxiras, S., Klaftenegger, D., Norgren, M., Ros, A. and Sagonas, K.: Turning Centralized Coherence and Distributed Critical-Section Execution on their Head: A New Approach for Scalable Distributed Shared Memory, *Proc. 24th International Symposium on High-Performance Parallel and Distributed Computing*, *HPDC '15*, pp.3–14, ACM (online), DOI: 10.1145/2749246.2749250 (2015).
[17] Kayraklioglu, E., Ferguson, M.P. and El-Ghazawi, T.: LAPPS: Locality-Aware Productive Prefetching Support for PGAS, *ACM Trans. Architecture and Code Optimization*, Vol.15, No.3, pp.1544–3566 (online), DOI: 10.1145/3233299 (2018).
[18] Lagraviŕe, J., Langguth, J., Prugger, M., Einkemmer, L., Ha, P.H. and Cai, X.: Performance Optimization and Modeling of Fine-Grained Irregular Communication in UPC, *Scientific Programming*, Vol.2019, No.6825728, pp.1–20 (online), DOI: 10.1155/2019/6825728 (2019).
[19] Lagraviŕe, J., Langguth, J., Prugger, M., Ha, P.H. and Cai, X.: A Newcomer In The PGAS World – UPC++ vs UPC: A Comparative Study (2021), available from ⟨https://arxiv.org/abs/2102.03614⟩.
[20] Löff, J., Griebler, D., Mencagli, G., Araujo, G., Torquati, M., Danelutto, M. and Fernandes, L.G.: The NAS Parallel Benchmarks for evaluating C++ parallel programming frameworks on shared-memory architectures, *Future Generation Computer Systems*, Vol.125, pp.743–757 (online), DOI: 10.1016/j.future.2021.07.021 (2021).
[21] Lebeck, A.R. and Wood, D.A.: Dynamic Self-Invalidation: Reducing Coherence Overhead in Shared-Memory Multiprocessors, *Proc. 22nd Annual International Symposium on Computer Architecture*, *ISCA '95*, pp.48–59 (online), DOI: 10.1145/223982.223995 (1995).
[22] Li, K. and Hudak, P.: Memory Coherence in Shared Virtual Memory Systems, *ACM Trans. Computing Systems*, Vol.7, No.4, pp.321–359 (online), DOI: 10.1145/75104.75105 (1989).
[23] Nelson, J., Holt, B., Myers, B., Briggs, P., Ceze, L., Kahan, S. and Oskin, M.: Latency-Tolerant Software Distributed Shared Memory, *Proc. 2015 USENIX Annual Technical Conference*, *ATC '15*, pp.291–305, USENIX (2015) (online), available from ⟨https://www.usenix.org/conference/atc15/technical-session/presentation/nelson⟩.
[24] Ruan, Z., Schwarzkopf, M., Aguilera, M.K. and Belay, A.: AIFM: High-Performance, Application-Integrated Far Memory, *Proc. 14th USENIX Symposium on Operating Systems Design and Implementation*, *OSDI '20*, pp.315–332, USENIX (2020) (online), available from ⟨https://www.usenix.org/conference/osdi20/presentation/ruan⟩.
[25] Schuchart, J.K.: *Global Task Data Dependencies in the Partitioned Global Address Space*, PhD Thesis, University of Stuttgart (2021).
[26] Stets, R., Dwarkadas, S., Hardavellas, N., Hunt, G.C., Kontothanassis, L.I., Parthasarathy, S. and Scott, M.L.: Cashmere-2L: Software Coherent Shared Memory on a Clustered Remote-Write Network, *Proc. 16th ACM Symposium on Operating System Principles*, *SOSP '97*, pp.170–183, ACM (online), DOI: 10.1145/268998.266675 (1997).
[27] Tardieu, O., Herta, B., Cunningham, D., Grove, D., Kambadur, P., Saraswat, V.A., Shinnar, A., Takeuchi, M., Vaziri, M. and Zhang, W.: X10 and APGAS at Petascale, *ACM Trans. Parallel Computing*, Vol.2, No.4, pp.25:1–25:32 (online), DOI: 10.1145/2894746 (2016).
[28] UPC Consortium: UPC Language and Library Specifications, v1.3, Technical Report LBNL-6623E, Lawrence Berkeley National Lab Tech Report (2013).
[29] Wael, M.D., Marr, S., Fraine, B.D., Cutsem, T.V. and Meuter, W.D.: Partitioned Global Address Space Languages, *ACM Computing Surveys*, Vol.47, No.4, pp.62:1–62:27 (online), DOI: 10.1145/2716320 (2015).
[30] Zheng, Y., Kamil, A., Driscoll, M.B., Shan, H. and Yelick, K.: UPC++: A PGAS Extension for C++, *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, *IPDPS '14*, pp.1105–1114, IEEE (online), DOI: 10.1109/IPDPS.2014.115 (2014).
[31] Zhou, Y., Iftode, L. and Li, K.: Performance Evaluation of Two Home-Based Lazy Release Consistency Protocols for Shared Virtual Memory Systems, *Proc. 2nd USENIX Symposium on Operating Systems Design and Implementation*, *OSDI '96*, pp.75–88, USENIX (1996) (online), available from ⟨https://www.usenix.org/legacy/publications/library/proceedings/osdi96/zhou.html⟩.

Appendix

A.1 Experimental Reproduction of Reference [16]

NPB CG was also used in the original study [16] of ArgoDSM, where CG class C on top of ArgoDSM achieved approximately 50-fold performance over serial execution. We conducted an experiment to reproduce that result.

A.1.1 Experimental Setup

We used the publicly available version of ArgoDSM[*2] (commit 4a7789a) and the publicly available NPB [*5] (commit 542c92f) for ArgoDSM. We measured the time taken for the main computation phase for CG class C problem (from NAS Parallel Benchmark) by following the built-in setting.

The benchmark program uses intra-node multithreading with OpenMP and inter-node communication based on ArgoDSM. We scaled the number of nodes with one thread per node (conforming to the experiments in Section 5) and 15 threads per node (conforming to the experiments in Ref. [16]). As the baselines, we also measured serial execution without OpenMP and ArgoDSM and 15-threaded execution without ArgoDSM.

The experimental environment was Oakbride-CX (Table 3) as in the experiments in Section 5.

We used `g++` 7.5.0 to compile both ArgoDSM and the benchmark programs with the `-O3` optimization option, where the OpenMP version was 201511. We used Open MPI 4.0.5 for inter-node communication.

A.1.2 Experimental Results

Table A·1 summarizes the execution time with 1–64 nodes and the relative speed over the corresponding single-node execution.

In the 15-thread-per-node case, the single-node execution was the fastest and multi-node executions did not scale up speed at all, which is different from the results of the original study. Notably, each node had higher performance than the one used in the original study, which was equipped with AMD Opteron™ 6220 (3.0–3.6 GHz), which could affect scalability negatively.

Then, in the single-thread-per-node case, multi-node executions scaled up speed up to 4.53 folds with 16 nodes, which conform roughly to the results of the relative speed over single-node execution reported in the original study. However, the situation that two-node execution is equivalent to or a little faster than single-node execution reported therein has not been reproduced.

Lastly, we compare the results with those in Section 5.2. Because the benchmark program roughly corresponded to the privatized one of our benchmark program, we compared S_1 in Table A·1 with the "privatize" plot in Fig. 3 (b). Both became smallest at two nodes. S_1 became largest at 16 nodes, while the "privatize" plot became largest at 32 nodes, which were slightly better than the 16 nodes. The "privatize" plot scaled up to 2.04, which was less than the maximum of S_1, 4.56. We attribute this slowdown to two major factors. One is that S_1 did not count the initial-

[*5] https://github.com/ioanev/npb

Table A·1 Execution time T_k in second and relative speed S_k for k threads per node. The baseline of S_1 is serial execution without OpenMP and ArgoDSM; that of S_{15} is 15-threaded execution without ArgoDSM.

#nodes	T_{15}	S_{15}	T_1	S_1
1	17.15	1.03	214.86	1.01
2	255.72	0.07	254.52	0.85
4	61.36	0.29	148.23	1.46
8	49.70	0.35	78.84	2.75
16	46.29	0.38	47.43	4.56
32	53.05	0.33	51.33	4.22
64	111.98	0.16	102.47	2.11

ization phase, where much communication occurred, but only the parallell computation phase, while the "privatize" plot counted both. The other is that S_1 was for the class C problem, while the "privatize" plot was for the smaller class B problem. Considering these factors, we conclude that the difference in speedup between them fitted in a reasonable range.

A.1.3 Threats to Validity

In this reproduction experiment, we used the publicly available version of ArgoDSM, and one of the benchmark programs by the authors' group of the original study. However, there is still room for completely reproducing the experiments in the original study.

One major point is that "the prototype implementation" of ArgoDSM used in the original study is different from the publicly available version. For example, the prototype implementation ran with up to 128 nodes, whereas the published implementation can run with up to 64 nodes. Based on the answers [*6] of one of the authors, "some of the more hardcoded features [of the prototype implementation] may have changed how some applications run." However, the details of the differences between them were unclear to even the answering author.

Another major point is that the current implementation of NPB CG for ArgoDSM can be different from the one used in the original study. The current implementation was based on NPB-CPP [*7][20], which did not exist at the publication date of ArgoDSM. Even if their programs in themselves were almost the same, their parameter settings would affect the performance significantly. The performance differences between this reproduction and the original study should be attributed to at least parameters regarding memory alignment and data stored in globally shared memory. However, even the answering author "cannot remember exactly how the applications use their memory and if alignment matter that much."[*6]

In summary, it is practically difficult to completely reproduce the experiments in the original study. This reproduction is our best effort for NPB CG.

[*6] Private communication on 2021-12-21
[*7] https://github.com/GMAP/NPB-CPP

Takato Hideshima is a graduate student in the Graduate School of Information Science and Technology at the University of Tokyo. He received his B.E. from the University of Tokyo in 2021. He is interested in memory management for distributed computing in C/C++.

Shigeyuki Sato is an Assistant Professor in the Graduate School of Information Science and Technology at the University of Tokyo. He received his Ph.D. from the University of Electro-Communications in 2015. His research interest is in compilers and parallel programming, especially, automatic parallelization, program synthesis, high-level optimizations, domain-specific languages, parallel patterns, and tree/graph processing. He is also a member of ACM and JSSST.

Kenjiro Taura is a Professor in the Department of Information and Communication Engineering at the University of Tokyo. He received his B.S., M.S., and Ph.D. from the University of Tokyo in 1992, 1994, and 1997, respectively. His major research interests spread parallel and distributed computing, system software, and programming languages. He is also a member of ACM, IEEE, and USENIX.