Electronic Preprint for Journal of Information Processing Vol.20 No.1

Regular Paper

Parallel Computational Reconfiguration Based on a PGAS Model

Kentaro Hara[1,a)] Kenjiro Taura[1]

Received: March 31, 2011, Accepted: September 12, 2011

Abstract: In order to improve the resource utilization of clusters and supercomputers and thus deliver application results to users faster, it is essential for a job scheduler to expand and shrink parallel computations flexibly. In order to enable the flexible job scheduling, the parallel computations have to be reconfigurable. With this motivation, this paper proposes, implements and evaluates *DMI*, a global-view-based PGAS framework that enables *easy* programming of *reconfigurable* and *high-performance* parallel iterative computations. DMI provides programming interfaces with which a programmer can program the reconfiguration easily with a global-view. Our performance evaluations showed that DMI can efficiently adapt the parallelism of long-running parallel iterative computations, such as a *real-world* finite element method and *large-scale* iterative graph search, to the dynamic increase and decrease of available resources through the reconfiguration.

Keywords: partitioned global address space, finite element method, reconfiguration, scalability, productivity

1. Introduction

1.1 Backgrounds and Goals

In order to improve the resource utilization of clusters and supercomputers and thus deliver application results to users faster, it is essential for a job scheduler to expand and shrink parallel computations flexibly. And in order to enable the flexible job scheduling, the parallel computations have to be reconfigurable. To take an example, let us consider a job scheduling system like TORQUE adopted in most supercomputing centers such as T2K [1] and TSUBAME2 [3] in Japan. In such a torque-like job scheduling system, a user can publish his parallel computation c_1 as a job specifying the number of nodes that he wishes to use, for example 1,000 nodes, and then the job scheduler dispatches c_1 when 1,000 nodes become available. Here note that even if 700 nodes are available at the time when c_1 is published, c_1 has to wait to be dispatched until the 1,000 nodes become available. Obviously, if the job scheduler can dispatch c_1 to the 700 nodes in the beginning and then expand c_1 to the 1,000 nodes when the 1,000 nodes become available, the user will be able to get results much faster. Furthermore, if another user publishes another computation c_2 which requires 800 nodes with higher priority than c_1, then it is preferable to dispatch c_2 immediately to the 800 nodes instead of shrinking the scale of c_1 from the 1,000 nodes to the remaining 200 nodes.

In this way, in order to improve the resource utilization and thus deliver application results to users faster, it is essential for the job scheduler to expand and shrink parallel computations flexibly. Therefore, the parallel computation itself has to be described so that it can expand and shrink its scale freely [7]. It is, however, obviously difficult to program such a reconfigurable parallel computation. With this motivation, this paper proposes, implements and evaluates *DMI* (Distributed Memory Interface), which is a global-view-based PGAS (Partitioned Global Address Space) framework that enables *easy* programming of *reconfigurable* and *high-performance* parallel iterative computations. These parallel iterative computations include many important scientific applications such as a finite element method, a multigrid method, a particle method and iterative graph search, which have high requirement for the dynamic reconfiguration because these applications are often long-running [19].

1.2 Contributions

The contributions of this paper are as follows:

- We point out that in order to support a reconfigurable parallel computation without much performance degradation, a processor virtualization model [14], [15] is inefficient and that it is required to create or kill processes/threads dynamically at every reconfiguration so that one process/thread is always assigned to one core (Section 2).
- We point out that a global-view-based PGAS model is more suitable than a message passing model for easily describing this dynamic increase and decrease of threads (Section 2).
- The access locality of a reconfigurable computation dynamically changes. Thus we propose *selective cache read/write*, with which a programmer can easily adapt data distribution to the actually observed access patterns in the reconfigurable computation *just* by specifying the access locality of each read/write (Section 3).
- We design and implement the programming interfaces that enable easy programming of the parallel reconfigurable it-

[1] School of Information Science and Technology, The University of Tokyo, Bunkyo, Tokyo 113–8656, Japan
[a)] haraken@logos.ic.i.u-tokyo.ac.jp

erative computations based on the global-view-based PGAS model (Section 4).

- Our performance evaluations showed that DMI can efficiently adapt the parallelism of long-running parallel iterative computations, such as a *real-world* finite element method and *large-scale* iterative graph search, to the increase and decrease of available resources through reconfiguration. We also confirmed that DMI has higher programmability than MPI for those *irregular* scientific applications thanks to the global address space (Section 5).

To the best of our knowledge, this is the first work that achieves the reconfiguration based on a PGAS model, although there have been several works based on a message passing model such as MPI [8], [19], [20], [26].

2. Programming Models Suitable for Reconfiguration

2.1 Relationships between Processes and Processors

There are primarily two reconfiguration models depending on whether multiple processes/threads are assigned to one core or not. We refer to the former model as a processor virtualization model [14], [15] and the latter model as a processor non-virtualization model.

First, in a processor virtualization model, for example adopted in Adaptive MPI [14], [15] and MPI checkpoint and restart [24], a programmer just has to program his parallel computation with a sufficient number of processes without considering the reconfiguration. Then a framework transparently reconfigures the computation by mapping these many processes to physically available resources through process migration dynamically. For example, in case that he creates 10,000 processes, if 1,000 cores are available then the framework assigns 10 processes to one core, and eventually if 100 cores become available then the framework assigns 100 processes to one core. The advantage of this model is good programmability because of its transparentness. On the other hand, the disadvantage is poor performance. In order to run his computation efficiently on the unpredicted number of resources, the programmer should create very many processes for good load balancing. However, not only is how many processes the programmer should initially create often unclear in a non-artificial dynamic environment, but also creating too many processes degrades performance dramatically [19]. The first reason for this performance degradation is the overhead of a parallel algorithm. In a parallel scientific computation with a domain decomposition, for example, both computation and communication increase as the degree of the decomposition increases and the number of ghost elements [23], [29] increases. The second reason is the overhead of assigning multiple processes to one core. **Figure 1** shows how performance degrades when n processes are assigned to one core using NAS Parallel Benchmark [12]. Figure 1 indicates that the case of $n = 8$ is 8.5 times in MG and 167.7 times in IS slower than the case of $n = 1$. This degradation is especially critical for parallel iterative computations since their speed is determined by the slowest process at every synchronization at every iteration. For example, the BiCGSafe method used in the finite element method in Section 5.3 includes as much as

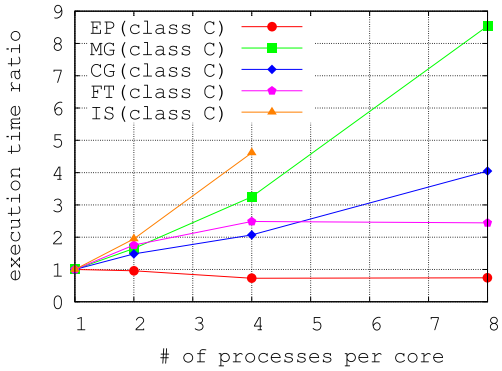


Fig. 1 Performance degradation when n processes are assigned to one core. This experiment used 16 nodes (128 cores) shown in Section 5.1, creating $128n$ processes in total. The execution time of IS at $n = 8$ is 167.7 (out of this graph). All execution times for each benchmark are normalized so that the execution time at $n = 1$ becomes 1.

22 synchronizations per iteration. Note that the execution time of each iteration is quite critical for the total execution time of long-running iterative applications.

Second, in a processor non-virtualization model, for example adopted in SRS [26], DyRecT [8], DRMS [17] and PCM [19], [20], processes are created or killed at every reconfiguration so that one process is always assigned to one core. The advantage of this model is good performance. On the other hand, the disadvantage is poor programmability. In this model since a set of processes changes at the reconfiguration, the data required for continuing a subsequent computation has to be temporarily checkpointed before the reconfiguration and then restored after the reconfiguration among the set of new processes. Here it is a programmer's burden to program explicitly which data should be checkpointed with what kind of data distribution and restored the data with what kind of data distribution. Furthermore, it is non-trivial what programming interfaces enable easy programming of this dynamic process reconfiguration.

With these observations, DMI adopts the processor non-virtualization model for performance reasons and then provides a data communication model and programming interfaces that together enhance the poor programmability of the processor non-virtualization model.

2.2 Data Communication Models

In the processor non-virtualization model a set of processes changes dynamically at the reconfiguration. In this sub-section we compare the programmability for this dynamic process reconfiguration between several representative data communication models: a message passing model, a local-view-based PGAS model [28], [29], a global-view-based PGAS model [16], [23] and a DSM (Distributed Shared Memory) model [5], [18].

First, in the message passing model such as MPI and PVM, a programmer explicitly describes send/receive operations using ranks of processes. In other words, the programmer has to describe who sends what data and who receives what data by explicitly managing data locations (= who owns what data). It is, however, difficult to program the dynamic process reconfiguration in this model since the programmer has to explicitly manage the complicated changes of the data locations at the reconfigura-

tion. Specifically, the reconfiguration requires data re-distribution in order to adapt the data distribution to the change of access locality and in order to hand over the data that a leaving process has owned from the leaving process to another process. In essence, this difficulty comes from the fact that in a *local-view* model like the message passing model it is the programmer that has to manage the complicated data locations. In this sense, the local-view-based PGAS model such as Co-array Fortran [28] also suffers the same problem.

Second, in the global-view-based PGAS model such as UPC [16], Global Arrays [23], X10 [4] and Chapel [6], the programmer can describe data communications as read/write operations from/to a global address space. This global-view-based PGAS model is very suitable to program the dynamic process reconfiguration. This is because the programmer can describe the data communications just by reading/writing from/to an intended memory address, no matter what processes are involved in the computation at the time. Assume, for example, a data x is located at the memory address 0x12340000. Here the programmer can obtain/store the data x anytime just by reading/writing from/to 0x12340000 without considering any reconfiguration since the data x always exists at 0x12340000. Thus it is easy to program the reconfigurable computation. In essence, this programming ease comes from the fact that in a *global-view* model like the global-view-based PGAS model it is not the programmer but a framework that manages the complicated data locations. In this sense, the DSM model such as TreadMarks [5] and CRL [18] also eases the programming of the reconfigurable computations since it also provides global-view [*1]. However, DMI adopts the global-view-based PGAS model instead of the DSM model since the PGAS model provides the programmer with more explicit ways for controlling data distribution to optimize his program straightforwardly. Finally, note that regardless of the reconfiguration, the global-view-based PGAS model delivers *good programmability* based on read/write operations similar to multi-thread programming on a physically shared memory environment [16], [23].

3. Design of a Global Address Space Framework

As discussed in Section 2.2, the reason why a programmer can easily program a reconfigurable parallel computation using a global-view-based PGAS model is that a framework, instead of the programmer, transparently manages the complicated change of data locations over the reconfiguration and abstracts it as a global address space. This section describes the design of the global address space for a *reconfigurable* and *high-performance* parallel computation.

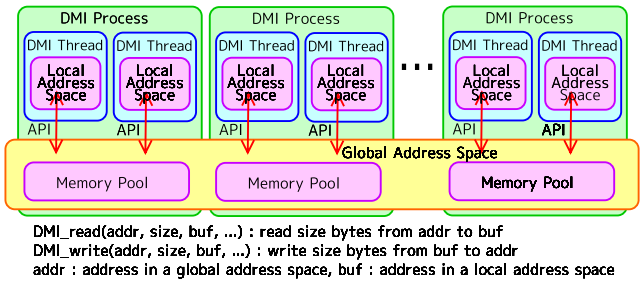


Fig. 2 The system structure of DMI.

3.1 System Overview

DMI is a multi-threaded global-view-based PGAS framework [10], [11] (**Fig. 2**). In DMI, while one node can have multiple processes and one process can have multiple threads, we assume below one process per node and one thread per core for simplicity and performance.

First, DMI provides a global address space with cache coherence. Each process provides some amount of memory called a *memory pool* for DMI and then DMI constructs the global address space over these distributed memory pools by implementing memory management mechanisms such as page tables at user level. Each thread can access all memory pools transparently through reading/writing from/to the global address space. If the accessed region of the global address space does not exist on the memory pool of the process on which the thread runs, a page fault occurs and the page is transferred from the process that owns the page at the time. At this point, since DMI maintains the cache coherence of the global address space, the process can cache the transferred page in its memory pool if necessary, while existing PGAS frameworks based on a put/get operation such as UPC, Global Arrays, X10 and Chapel do not support cache coherence. Second, since the memory pool is shared with multiple threads on the same process, the data sharing between the threads on the same process is realized efficiently through a physical shared memory. In this way a programmer can enjoy hybrid programming transparently without considering the distinction between inter-node parallelism and intra-node parallelism. Third, DMI behaves as a remote swap system by allocating huge global address spaces across the memory pools on multiple nodes. When the memory pool is saturated over repeated remote pagings, DMI sweeps the memory pool on the basis of a page replacement algorithm. Fourth, DMI maintains the cache coherence of the global address space over the dynamic joining and leaving of processes. While mechanisms and coherence protocols to achieve the dynamic joining and leaving of processes are never trivial, we leave the details to another paper [11].

DMI has high portability since DMI is implemented as a static library for C. DMI is implemented in approximately 27,000 lines of C program. DMI provides 83 APIs to support and optimize a broad range of high-performance parallel scientific applications, for example APIs for a memory allocation/deallocation, a memory read/write, an asynchronous memory read/write, a mutual exclusion, a collective synchronization, a user-defined read-modify-

[*1] Both global-view-based PGAS model and DSM model provide a global address space, and there is no exact definition of these two models. In this paper, a PGAS model refers to the programming model by which a programmer can explicitly and strongly control data distribution and can access remote data by get/put operations. In the PGAS model, the programmer needs to describe his program considering each data location and how data is transferred between nodes. On the other hand, a DSM model refers to the programming model by which the programmer can transparently access the global address space by normal read/write memory operations without considering data location or even the difference between remote data and local data. According to this definition and the features of DMI that we mention in Section 3, this paper classifies DMI not in a DSM model but in a global-view-based PGAS model.

write, aggregating multiple discrete reads/writes, expressing irregularly decomposed domains and so on [10], [11].

3.2 A Memory Consistency Model

DMI provides an intuitive memory consistency model, but with which a programmer can explicitly and powerfully optimize the performance of accessing the global address space [11]. DMI separates between the global address space and a local address space as shown in Fig. 2. The local address space of each thread is a normal shared memory allocated/deallocated by mmap()/munmap() and read/written normally. On the other hand, the global address space is allocated/deallocated by DMI_mmap()/DMI_munmap() and read/written by DMI_read()/DMI_write().

The programmer can allocate the global address space with the arbitrary coherence granularity suitable for his computation, as with region-based DSMs [18]. We refer to the coherence granularity as a *page*. Specifically, by calling DMI_mmap(int64_t *page_size*, int64_t *page_num*) the programmer can allocate the global address space with *page_num* pages of *page_size* bytes in size. Thus the programmer can specify any block-cyclic data distribution. For example, when the programmer computes a matrix-matrix multiplication using some block partitioned algorithm, the programmer can allocate the global address spaces with page sizes that agree with the block size of each partitioned submatrix. In this way the programmer can enlarge the unit size of internal data transfers as much as necessary for his computation by adjusting the coherence granularity explicitly. This reduces the number of page faults dramatically and thus improves communication performance, compared to PGAS frameworks and DSM frameworks that utilize the memory protection mechanism of an operating system. Note that when the programmer cannot predict in advance how the number of processes will change with dynamic reconfiguration, it is important to specify some fine coherence granularity in order to avoid false sharing.

The programmer can access the global address space by calling DMI_read(int64_t *addr*, int64_t *size*, void **buf*,...)/DMI_write(int64_t *addr*, int64_t *size*, void **buf*,...), which reads/writes *size* bytes from/to the global address space *addr* to/from a local address space *buf*. DMI guarantees the sequential consistency, an intuitive and easy-to-understand memory consistency model, of DMI_read() and DMI_write(), the address range [*addr*, *addr+size*) of which is within one page. When the programmer calls DMI_read()/DMI_write() across multiple pages, the semantics is such that DMI_read()/DMI_write() is called separately and concurrently for the individual pages. Thus since DMI does not guarantee the atomicity of DMI_read()/DMI_write() across multiple pages, a mutual exclusion is required if necessary. In addition, DMI provides asynchronous DMI_read()/DMI_write(), with which the programmer can explicitly relax the (intuitive but sometimes too strong) sequential consistency depending on the requirement of his computation.

3.3 Access Locality Optimization Over Reconfiguration

In existing PGAS frameworks such as UPC, Global Arrays, X10 and Chapel, data distribution on the global address space is determined at data allocation and the data distribution cannot be changed dynamically. However, when a set of processes is reconfigured, the data that each process frequently accesses changes. Thus it is significant to adapt the data distribution to this dynamic change of *access locality*. The simplest approach for this dynamic data re-distribution is to have a programmer explicitly describe the data re-distribution at the reconfiguration [8], [17], [19], [20], [26]. However, it is complicated to describe the data re-distribution especially for data with irregular structures. With this observation, we propose a *selective cache read/write* [*2], with which the programmer does not have to describe the complicated data re-distribution but just has to specify the access locality of each read/write. Here the access locality of the read/write means how the data should be transferred and cached in the memory pool of the process that issues the read/write. With this selective cache read/write the programmer can explicitly and flexibly optimize the data transfers, including the data re-distribution at the reconfiguration, by controlling this access locality of each read/write.

To understand this selective cache read/write a little knowledge about the cache management of DMI is required. In DMI a process can have a cache of a page in its memory pool. Each page has one *owner* process that always has the cache of the page and has the responsibility for maintaining the cache coherence of the page, while this owner can migrate dynamically. Here a read fault occurs when the process does not have the cache. A write fault occurs when the process is not the owner or the process is the owner but at least one other process has the cache (since the cache coherence has to be maintained with the cache). Therefore the objective of the selective cache read/write is to minimize the performance degradation caused by these read/write faults by explicitly controlling the access locality of each read/write, that is, by explicitly controlling how the accessed page should be transferred and cached when each read/write causes the read/write fault. Specifically, the programmer can specify this access locality as the argument *mode* of DMI_read(int64_t *addr*, int64_t *size*, void **buf*, int *mode*)/DMI_write(int64_t *addr*, int64_t *size*, void **buf*, int *mode*).

We assume below that a process i is going to read/write the page whose owner is a process v. The possible read modes of DMI_read() are as follows (**Fig. 3**):

INVALIDATE mode: The page is transferred from the owner v and then cached in the memory pool of the process i. This cache is invalidated when the page is updated by some process next.

UPDATE mode: The page is transferred from the owner v and then cached in the memory pool of the process i. This cache is kept updated whenever the page is updated.

GET mode: Only the requested part of the page by this read is transferred from the owner v but not cached.

Considering that the owner v has to maintain the cache coherence by invalidating or updating all caches when the page is updated by a write, a programmer should use (1) an INVALIDATE mode if the page will be read in the near future and the perfor-

[*2] We name it "selective" since a programmer can explicitly select cache behaviors.

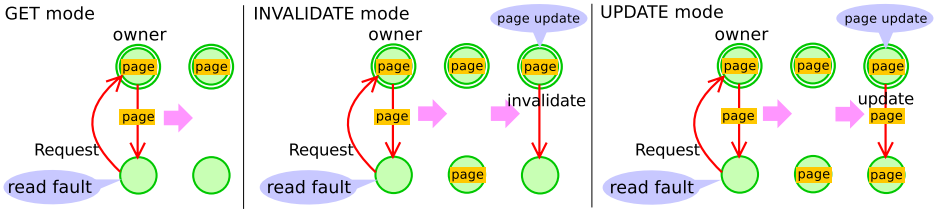


Fig. 3 Selective read.

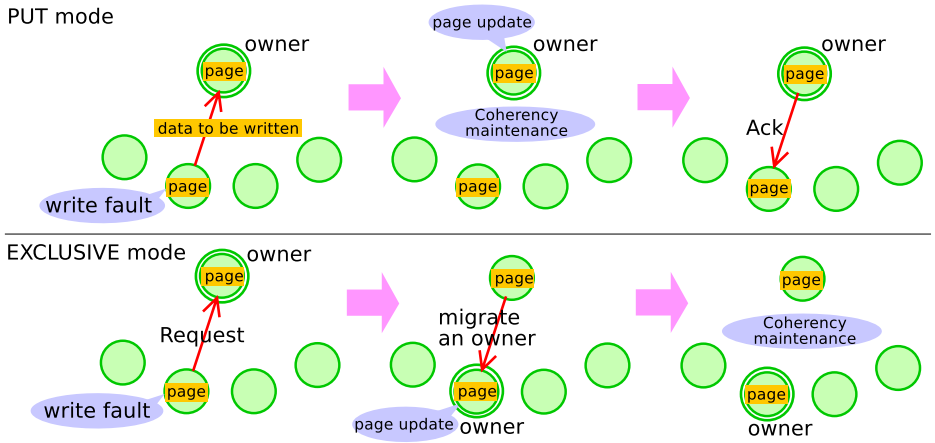


Fig. 4 Selective write.

mance of a write is more critical than that of a read; (2) an UPDATE mode if the page will be read in the near future and the performance of a read is more critical than that of a write; (3) a GET mode if the page will not be read in the near future or only a small part of the page is read.

Next, the possible write modes of DMI_write() are as follows (**Fig. 4**):

EXCLUSIVE mode: After the process i steals the owner authority from the owner v, the page is updated by the process i. Thus the process i becomes a new owner.

PUT mode: The data to be written is sent to the owner v and the page is updated by the owner v. Thus the owner does not change.

Obviously, the owner should not migrate unnecessarily by the EXCLUSIVE mode because the owner migration can cause a heavy page transfer and the overhead of DMI to locate the owner increases as the owner migrates frequently. Thus the programmer should use (1) an EXCLUSIVE mode if only the process has strong write locality for the page; (2) a PUT mode otherwise. In particular, the EXCLUSIVE mode dynamically adapts data distribution to the actually observed access patterns and thus adapts the data distribution to the change of access locality over reconfiguration.

Thus, in DMI the data distribution is adapted to the actually observed access patterns *just* by specifying the access locality of each read/write. This is a productive way of optimizing the dynamically changeable access locality of reconfigurable computations. In addition, regardless of the reconfiguration, this selective cache read/write is a powerful annotation for performance optimization in that the programmer can combine write-invalidate cache, write-update cache and get/put operations very flexibly at every read/write. In essence, the key for optimizing a DMI program is to appropriately control the access granularity by the argument *page_size* of DMI_mmap() and the access locality by the argument *mode* of each DMI_read()/DMI_write().

4. Reconfigurable Programming Interfaces

This section describes the design and implementation of the programming interfaces primarily targeted for SPMD parallel iterative computations with reconfiguration.

4.1 Design

4.1.1 A Basic Idea

Generally, the control flow of the SPMD parallel iterative computation with reconfiguration has the form of **Fig. 5**. Specifically, (1) one thread executes an initialization phase; (2) a set of threads executes the computation for a while; (3) eventually the set of threads changes according to the change of a set of nodes, and the set of new threads continues the computation for a while again; (4) finally the computation completes and one thread executes a finalization phase. Here we refer to the period during which the set of threads does not change as a *itergroup*. The control flow shown in Fig. 5 consists of three itergroups.

In order to express the control flow naturally and flexibly, programming interfaces that meet the following properties are required:

- Each itergroup should be expressed as a normal SPMD program.
- When to finish each itergroup should be programmable. In other words, the condition that determines a reconfiguration timing should be controlled flexibly on the basis of the resource information about joining nodes and leaving nodes.
- At the end of each itergroup, the data required for continuing subsequent itergroups has to be checkpointed to a global address space. At the start of the next itergroup, the data has

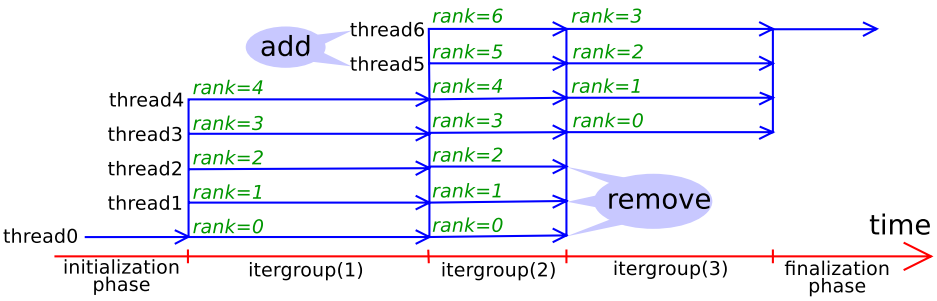


Fig. 5 The control flow of an SPMD parallel iterative computation with reconfiguration.

```
01: void DMI_main(int argc, char **argv) {
02:     struct data_t data; /* the data to be checkpointed and restored */
03:     int init_node_num = atoi(argv[1]); /* the initial number of nodes */
04:     ...; /* an initialization phase of this application */
05:     int64_t addr = DMI_mmap(sizeof(data), 1); /* allocate a global address space */
06:     data.iter = 0; /* a current iteration number */
07:     DMI_write(addr, sizeof(data), &data, PUT); /* write to the global address space */
08:     DMI_rescale(addr, init_node_num);
09:     DMI_munmap(addr); /* deallocate the global address space */
10:     ...; /* a finalization phase of this application */
11: }
12:
13: int DMI_itergroup(int rank, int pnum, int64_t addr) {
14:     struct data_t data;
15:     int iter;
16:     DMI_read(addr, sizeof(data), &data, GET); /* restore */
17:     ...; /* calculate the role of this thread based on rank and pnum */
18:     for(iter = 0; iter < 100 && data.iter < ITER_MAX; iter++, data.iter++) {
19:         ...; /* the body of each iteration */
20:     }
21:     DMI_write(addr, sizeof(data), &data, PUT); /* checkpoint */
22:     return data.iter != ITER_MAX;
23: }
```

Fig. 6 The outline of a reconfigurable DMI program (A simple version).

```
01: void DMI_main(int argc, char **argv) {
02:     struct data_t data; /* the data to be checkpointed and restored */
03:     int init_node_num = atoi(argv[1]); /* the initial number of nodes */
04:     ...; /* an initialization phase of this application */
05:     int64_t addr = DMI_mmap(sizeof(data), 1); /* allocate a global address space */
06:     data.iter = 0; /* a current iteration number */
07:     DMI_write(addr, sizeof(data), &data, PUT); /* write to the global address space */
08:     DMI_rescale(addr, init_node_num);
09:     DMI_munmap(addr); /* deallocate the global address space */
10:     ...; /* a finalization phase of this application */
11: }
12:
13: int DMI_itergroup(int rank, int pnum, int64_t addr) {
14:     struct data_t data;
15:     DMI_read(addr, sizeof(data), &data, GET); /* restore */
16:     ...; /* calculate the role of this thread based on rank and pnum */
17:     while(data.iter < ITER_MAX) {
18:         if(DMI_check_rescale()) {
19:             break;
20:         }
21:         ...; /* the body of each iteration */
22:         data.iter++;
23:     }
24:     DMI_write(addr, sizeof(data), &data, PUT); /* checkpoint */
25:     return data.iter != ITER_MAX;
26: }
27:
28: int DMI_judge_rescale(DMI_node_t *in_nodes, DMI_node_t *out_nodes, DMI_node_t *cur_nodes, int
in_node_num, int out_node_num, int cur_node_num) {
29:     return in_node_num + out_node_num >= 1;
30: }
```

Fig. 7 The outline of a reconfigurable DMI program (A refined version).

to be restored from the global address space.

4.1.2 A Simple Version

With these observations, we propose the programming interfaces shown in **Fig. 6** (which is refined later in **Fig. 7**). In Fig. 6 the initialization and finalization phases are expressed as DMI_main() and each itergroup is expressed as DMI_itergroup().

DMI firstly executes DMI_main() using one thread (line 1). Here a programmer can describe the initialization phase of his application. When the programmer calls DMI_rescale(*addr*, *init_node_num*) (line 8), DMI waits until *init_node_num* nodes become available and then creates one thread per core on these nodes. Next, all these threads created by DMI execute

```
function itergroup_wrapper(thread_id, args_addr):
    while 1 do
        pnum ← $thread_num
        barrier(pnum + 1) /* barrier A */
        my_rank ← $ranks[thread_id]
        ret ← DMI_itergroup(my_rank, pnum, args_addr) /* itergroup */
        if my_rank == 0 and ret == 0 then
            $exit_flag ← 1
        endif
        barrier(pnum + 1) /* barrier B */
        barrier(pnum + 1) /* barrier C */
        if $flags[thread_id] == 1 then
            break
        endif
    endwhile

function DMI_rescale(args_addr):
    $exit_flag ← 0
    foreach thread_id in maximal # of threads do
        $flags[thread_id] ← 0
    endforeach
    pnum ← 0
    RunningThread ← ∅
    while 1 do
        NewNode ← ∅
        DeleteNode ← ∅
        NewThread ← ∅
        DeleteThread ← ∅
        old_pnum ← pnum
        foreach node in the joining nodes do
            NewNode ← NewNode ∪ {node}
            foreach i in # of cores of the node node do
                thread_id ← a unique thread ID
                NewThread ← NewThread ∪ {thread_id}
                RunningThread ← RunningThread ∪ {thread_id}
                nodes[thread_id] ← node
                pnum ← pnum + 1
            endforeach
        endforeach
```

Fig. 8 An algorithm of DMI_rescale() and a wrapper function of DMI_itergroup() (leading to Fig. 9).

DMI_itergroup(int *rank*, int *pnum*, int64_t *addr*) (line 13), where *pnum* is the number of threads involved in this itergroup, *rank* is the rank of this thread (0≤*rank*<*pnum*) and *addr* is the first actual argument of DMI_rescale() (line 8). Thus the programmer can pass arbitrary data from DMI_main() to DMI_itergroup() and from DMI_itergroup() to the next DMI_itergroup() by storing that data into the global address space pointed (directly or indirectly) by the *addr*. The programmer can generally describe each itergroup as DMI_itergroup() as follows: (1) Restore the data checkpointed by the previous itergroup from the global address space (line 16); (2) Execute an SPMD computation based on *rank* and *pnum* (line 19); (3) Checkpoint the necessary data to the global address space (line 21). In Fig. 6, for example, a current iteration number is checkpointed and restored.

In order to improve the responsiveness to the joining and leaving requests of nodes triggered by some external factors, the execution time of each itergroup has to be appropriately short since the nodes can actually join and leave only between itergroups. This can be achieved simply by finishing DMI_itergroup() every appropriate number of iterations, for example every 100 iterations (line 18). Here the programmer can specify whether the itergroup is the final itergroup or not, namely, whether the total computation has completed or not, by the return value of DMI_itergroup() of the thread with rank 0 (line 22). If the return value is 0, DMI_rescale() called from DMI_main() returns (line 8). On the other hand, if the return value is not 0, DMI handles the joining and leaving requests being published at the time, reconfigures a set of nodes and a set of threads transparently and then creates one thread per core on these nodes. Next, all these threads start to execute DMI_itergroup(int *rank*, int *pnum*, int64_t *addr*) with a new *rank* and *pnum*. Thus a new itergroup starts.

4.1.3 A Refined Version

Although the code shown in Fig. 6 works well with reconfiguration, in terms of performance, it is a bit wasteful to once checkpoint and then restore data even when no node is publishing a joining or leaving request. Thus, as shown in Fig. 7, by using DMI_check_rescale() a programmer can finish the itergroup only when the reconfiguration becomes mandatory. Specifically, when the programmer calls DMI_check_rescale() (line 18), DMI executes DMI_judge_rescale(DMI_node_t **in_nodes*, DMI_node_t **out_nodes*, DMI_node_t **cur_nodes*, int *in_node_num*, int *out_node_num*, int *cur_node_num*) defined in his program (line 28) and the return value of the DMI_judge_rescale() becomes the return value of the DMI_check_rescale(). Here the arguments of the DMI_judge_rescale() are a set of joining nodes *in_nodes*, a set of leaving nodes *out_nodes*, a set of currently running nodes *cur_nodes*, the number of joining nodes *in_node_num*, the number of leaving nodes *out_node_num* and the number of currently running nodes *cur_node_num*. Therefore the programmer can flexibly program the condition that finishes the itergroup and thus control the condition for the reconfiguration.

4.2 Implementation

Simply, at the end of a itergroup, a set of nodes and a set of threads can be reconfigured as follows: (1) Retrieve all the threads involved in the itergroup; (2) Handle the joining/leaving

```
    foreach node in the leaving nodes do
      DeleteNode ← DeleteNode ∪ {node}
      foreach thread_id in thread IDs of all the threads on the node node do
        DeleteThread ← DeleteThread ∪ {thread_id}
        RunningThread ← RunningThread \ {thread_id}
        pnum ← pnum − 1
        $flags[thread_id] ← 1
      endforeach
    endforeach
    $thread_num ← pnum
    barrier(old_pnum + 1) /* barrier C */
    rank ← 0
    foreach thread_id in RunningThread do
      $ranks[thread_id] ← rank
      rank ← rank + 1
    endforeach
    foreach node in NewNode do
      handle the joining of the node node
    endforeach
    foreach thread_id in NewThread do
      handle[thread_id] ← thread_create(nodes[thread_id], itergroup_wrapper, thread_id, args_addr) /* cre-
ate a thread on the node nodes[thread_id]. This thread invokes itergroup_wrapper(thread_id, args_addr) */
    endforeach
    barrier(pnum + 1) /* barrier A*/
    foreach thread_id in DeleteThread do
      thread_join(handle[thread_id]) /* retrieve a thread */
      $flags[thread_id] ← 0
    endforeach
    foreach node in DeleteNode do
      handle the leaving of the node node
    endforeach
    barrier(pnum + 1) /* barrier B */
    if $exit_flag == 1 then
      break
    endif
  endwhile
  ...
```

Fig. 9 An algorithm of DMI_rescale() and a wrapper function of DMI_itergroup() (following to Fig. 8).

requests of nodes and determine the set of nodes involved in the next itergroup; (3) Create one thread per core on the nodes and start the next itergroup with the set of new threads. It is, however, wasteful to retrieve all the threads at every reconfiguration. In fact, since it is often the case that a just small, not large, part of the set of nodes changes at the reconfiguration, it is wasteful to once retrieve and again create threads on the nodes that continue to run over the reconfiguration. Therefore we propose an algorithm for reconfiguring the set of nodes and the set of threads without retrieving the threads on the nodes that continue to run over the reconfiguration.

The algorithm is shown in **Figs. 8** and **9**. Figures 8 and 9 describe how DMI_itergroup() is invoked (function itergroup_wrapper()) and what DMI_rescale() does (function DMI_rescale()). In Figs. 8 and 9 the variables prefixed with $ are on a global address space and other variables are local to each thread. Specifically, $*thread_num* indicates the number of threads involved in the current itergroup, $*exit_flag* indicates whether the current itergroup is the final itergroup or not, $*ranks*[*thread_id*] indicates the rank of the thread with thread ID *thread_id*, $*flags*[*thread_id*] indicates whether the thread with thread ID *thread_id* is going to be retrieved or not. Here a thread ID means a unique descriptor for each thread. The thread ID of a thread does not change throughout its lifetime but the rank of the thread changes at every itergroup.

barrier(*pnum*) synchronizes *pnum* threads. In particular, barrier(*pnum*+1) indicates that the *pnum* threads involved in the itergroup and the thread executing DMI_rescale() are synchronized together. In Figs. 8 and 9 three types of barriers are combined sophisticatedly, where barrier A guarantees that the thread executing DMI_rescale() has already set $*ranks*[∗], barrier B guarantees that the thread with rank 0 has already set $*exit_flag*, and barrier C guarantees that the thread executing DMI_rescale() has already set $*thread_num* and $*flags*[∗].

The reason why this algorithm retrieves threads and handles the leaving of nodes not before but after barrier A is that by doing these operations after barrier A, which determines the start timing of each itergroup, these heavy operations can be overlapped with the next itergroup. This overlapping considerably reduces the time from the end of the itergroup to the start of the next itergroup.

5. Evaluations of Performance and Programmability

This section compares the basic performance and the programmability of DMI with those of MPI and also evaluates the performance and the programmability of parallel computational reconfiguration in DMI. The reason why we compare DMI's performance with MPI's performance is that MPI is a de facto standard in high-performance parallel programming and is easy to optimize its performance appropriately.

5.1 Experimental Settings

The experimental platform is the cluster environment composed of 16 nodes interconnected by 10 Gbit Ethernet. Each node contains 2 Intel Xeon E5530 2.40 GHz (4 physical cores but 8 logical cores with hyper-threading) CPUs, 24 GB of memory, running the Linux OS with the 2.6.26-2-amd64 kernel. We used

gcc 4.3.2 with an -O3 option for DMI, and OpenMPI 1.4.2 and mpich2-1.2.1p1 with an -O3 option for MPI. When we executed a DMI program using n ($1 \le n \le 128$) cores, we created 8 threads on $\lfloor n/8 \rfloor$ nodes and the remaining $n - 8 \times \lfloor n/8 \rfloor$ threads on another node. Inside one node, DMI threads communicated through their shared memory. On the other hand, when we executed a MPI program using n ($1 \le n \le 128$) cores, we created 8 processes on $\lfloor n/8 \rfloor$ nodes and the remaining $n - 8 \times \lfloor n/8 \rfloor$ processes on another node.

5.2 A PDE Solver Using a Jacobi Method

This experiment solves a partial differential equation of heat conduction in a 3-dimensional cube using a Jacobi method by dividing the cube into 512^3 elements. In this Jacobi method, the data to be checkpointed and restored are a current iteration number and the values of 512^3 elements. The algorithm of each itergroup is as follows: (1) One thread divides the entire cube into rectangular parallelepiped domains evenly in the direction of the z-axis among all threads, each of which has 514^2 ghost elements [23], [29] in its left and right neighboring domains; (2) Each thread i reads the values of the elements of the domain i from a global address space; (3) Each thread i repeats from (4) to (6) until DMI_check_rescale() tells that reconfiguration becomes mandatory. If mandatory, each thread i finishes this itergroup after checkpointing the current iteration number and the values of the elements of the domain i to the global address space; (4) Each thread i writes the values of the ghost elements for the neighboring domains $i-1$ and $i+1$ to the global address space; (5) All the threads synchronize and then each thread i reads the values of the ghost elements that were written by the neighboring threads $i-1$ and $i+1$ in (4); (6) Each thread i computes the 27-point stencil of the domain i. Here note that as a set of threads changes across itergroups, the domain that each thread works on changes and thus the access locality of each thread changes. Therefore this experiment sets the page size of the global address space for the ghost elements to $514^2 \times$sizeof(double) and uses DMI_write() with an EXCLUSIVE mode in (4) and DMI_read() with a GET mode in (6) in order to adapt data distribution to the change of the access locality dynamically according to the observed access patterns.

First, **Fig. 10** shows the weak scalability of DMI (without any reconfiguration) and MPI to compare the basic performance of DMI with that of MPI. Figure 10 indicates that DMI achieved similar performance to mpich2 and outperformed OpenMPI.

Second, **Fig. 11** shows the execution time of each iteration in DMI when we started the computation on 4 nodes (32 threads on 4 nodes in total), added 12 nodes at the end of (about) the 30-th iteration (128 threads on 16 nodes in total) and then removed 8 nodes at the end of (about) the 60-th iteration (64 threads on 8 nodes in total). Figure 11 shows 3 lines. The line labeled DMI is the result of this experiment. The line labeled DMI(put) is the result when we used DMI_write() with a PUT mode instead of an EXCLUSIVE mode as the DMI_write() in (4). For comparison, the line labeled DMI(pv) is the result based on a processor virtualization model, which we have investigated in detail in our previous work [9]. In this processor virtualization model, we created 128 threads in the beginning. Then DMI mapped these 128 threads to available nodes automatically at each reconfiguration through transparent thread migration. Specifically, 4 threads are mapped on each physical core between the 0-th and the 30-th iteration (since 4 nodes (=32 cores) and 128 threads in total), 1 thread is mapped on each core between the 31-st and the 60-th iteration (since 16 nodes (=128 cores) and 128 threads in total), and 2 threads are mapped on each core between the 61-st and the 90-th iteration (since 8 nodes (=64 cores) and 128 threads in total). However, here we ignore the time for reconfiguration (i.e., the time required for thread migration in the processor virtualization model and the time required for checkpointing and restoring in the processor non-virtualization model), because the mechanism of thread migration is complicated and out of the scope of this paper and because the reconfiguration time is less critical than the execution time of each iteration for the total execution time of long-running iterative applications, considering that reconfiguration does not happen so often in practical. In this way, each point of DMI(pv) in Fig. 11 is the execution time of each iteration and does not include any time requried for thread migration. Also, each point of DMI in Fig. 11 does not include any time required for checkpointing and restoring. Figure 11 indicates that DMI can adapt the parallelism efficiently to the increase or decrease of available resources through the reconfiguration. In contrast, the execution time of DMI(put) did not change even when we increased or decreased nodes. This fact indicates that DMI_write() with the EXCLUSIVE mode is pretty effective for adapting data distribution to the change of access locality over the reconfigura-

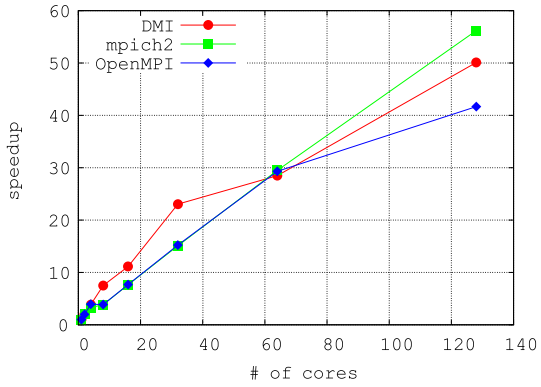


Fig. 10 The scalability of the Jacobi method.

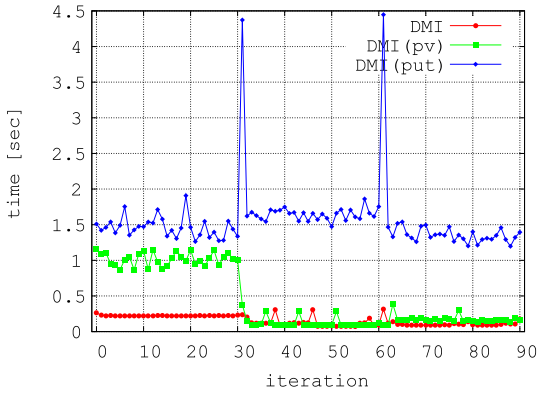


Fig. 11 The execution time of each iteration of the Jacobi method with reconfiguration.

Table 1 The breakdown of the reconfiguration time [seconds].

	Jacobi	FEM	Pagerank
checkpoint(32→128 threads)	0.900	1.18	0.621
restore(32→128 threads)	0.949	7.44	18.0
checkpoint(128→64 threads)	1.29	1.46	0.898
restore(128→64 threads)	0.928	8.92	23.0

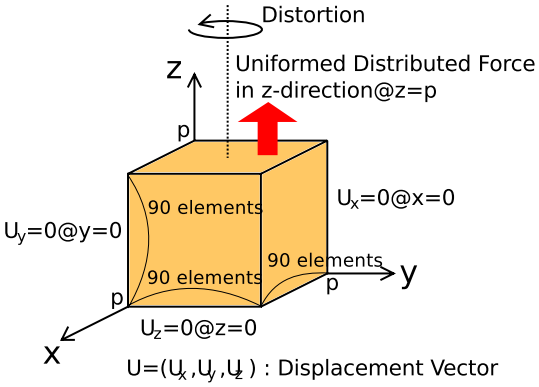


Fig. 12 Stress analysis using the finite element method.

tion. In addition, the fact that the execution time of DMI(pv) was much higher than that of DMI between the 0-th and the 30-th iteration indicates that assigning more than 1 threads to each physical core degrades performance terribly because of the overhead that we mentioned in Section 2.1. Thus, in terms of performance, a processor non-virtualization model is much preferable to a processor virtualization model. Again note that the execution time of each iteration is more critical than the reconfiguration time for the long-running iterative applications.

Third, **Table 1** shows the breakdown of the reconfiguration time. In Table 1, "checkpoint" means the time required for checkpointing data to the global address space before the reconfiguration (in this Jacobi method, the current iteration number and the values of 512^3 elements), and "restore" means the time required for re-calculating the role of each thread among the set of new threads (in this Jacobi method, which domain each thread should work on) and restoring the data from the global address space after the reconfiguration. In fact, 1.03 MB of data is checkpointed/restored to/from the global address space.

Fourth, with respect to programmability, the lines of the MPI program was 147 and the lines of the DMI program without any reconfiguration was 168 (both excluding comment lines and empty lines). We confirmed that adding only 11 lines to this DMI program made it reconfigurable.

5.3 Stress Analysis Using a Finite Element Method

This experiment analyzes the stress of the 3-dimensional cube with the force and the boundary conditions shown in **Fig. 12** using the finite element method with 90^3 elements. These elements are distorted up to 200 degrees around the z-axis based on the Sequential Gauss Algorithm. This finite element method is reduced to the problem of solving the linear simultaneous equation $A\boldsymbol{x} = \boldsymbol{b}$, where A is the sparse matrix representing the connectivity between elements and $\boldsymbol{b}$ is the vector representing the force and the boundary conditions. This is a *real-world* and hard-to-converge problem used in the parallel programming competition in Japan [2] and various engineering methods are essential

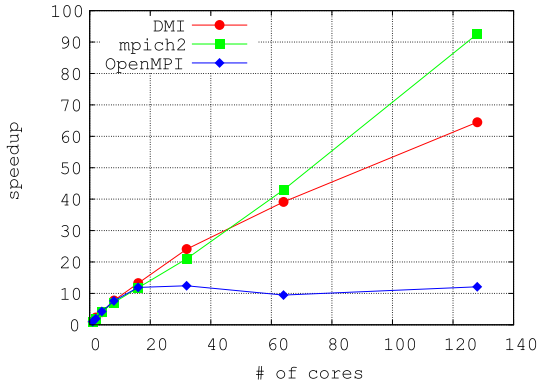


Fig. 13 The scalability of the finite element method.

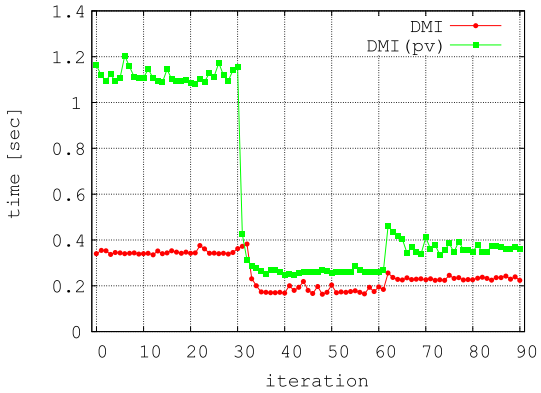


Fig. 14 The execution time of each iteration of the finite element method with reconfiguration.

to solve. While we omit the details, this experiment uses the iterative method called the BiCGSafe method, with an *irregular* domain decomposition considering load balancing, deep domain overlapping using the Restricted Additive Schwarz Method, the RCM ordering of the elements of each domain and preconditioning using the blocked ILU decomposition with fill-ins, which is the champion algorithm of the contest. In this finite element method, the data to be checkpointed and restored are a current iteration number, 13 vectors and 2 variables used in the BiCGSafe method.

First, **Fig. 13** shows the weak scalability of DMI (without any reconfiguration) and MPI. Figure 13 indicates that DMI achieved lower scalability than mpich2. This is because the performance of the collective function for synchronizing all 128 threads, which is called as much as 22 times in each iteration of the BiCGSafe method, becomes worse than that of MPI, as the number of threads increases. Figure 13 also indicates that OpenMPI performed much worse than DMI and mpich2. Here this low performance of OpenMPI was attributed to the slow point-to-point send/receive communication of OpenMPI. Specifically, it took 2.39 seconds in mpich2 but 9.03 seconds in OpenMPI to simply send and receive 65,536 bytes of data 10,000 times between two nodes.

Second, as with Fig. 11, **Fig. 14** shows the execution time of each iteration in DMI when we started the computation on 4 nodes, added 12 nodes at the end of (about) the 30-th iteration and then removed 8 nodes at the end of (about) the 60-th iteration. Figure 14 indicates that DMI can also adapt the parallelism effi-

ciently through the reconfiguration for complicated and irregular scientific applications, and that the processor non-virtualization model outperforms better than the processor virtualization model.

Third, Table 1 shows the breakdown of the reconfiguration time. The breakdown of 7.44 seconds required for "restore" when expanding 32 threads to 128 threads was 3.90 seconds for restoring 13 vectors (1.03 GB in total) from the global address space and generating a preconditioning matrix, and 3.54 seconds for reading the sparse matrix from a file (3.85 GB in total), re-decomposing domains and re-ordering elements based on the set of new threads. The breakdown of 8.92 seconds required for "restore" when shrinking 128 threads to 64 threads was 6.06 seconds for restoring the 13 vectors and generating a preconditioning matrix, and 2.85 seconds for reading the file, re-decomposing the domains and re-ordering the elements. In this experiment since the matrix file existed entirely on the file cache of an operating system, it will take more time if the matrix file exists on a disk.

Fourth, with respect to programmability, the lines of the MPI program was 2,572 and the lines of the DMI program without reconfiguration was 2,368. We also confirmed that adding 187 lines to this DMI program made it reconfigurable. The difference between the 2,572 lines of DMI and the 2,368 lines of MPI was primarily attributed to whether a programmer can program the exchange of the values of the ghost elements in a global-view or not. Since this finite element method requires irregular domain decompositions, in MPI the calculation is very complicated and error-prone of the index of the local buffer from which data should be sent, the process to which the data should be sent, and the index of the local buffer to which the received data should be stored. In contrast, in DMI the programmer can productively program just by reading/writing from/to the global addresses of the ghost elements. In essence, DMI is easier to program than MPI thanks to its global address space.

5.4 Pagerank Calculation

This experiment calculates a pagerank [21] of a large-scale web graph. The experiment generates a web graph similar to the one in the real world with the following properties:

- The total number of vertexes is 128 million.
- The entire graph is composed of 128 sub-graphs, each of which has 1 million vertexes.
- The in-degrees of vertexes are distributed log-normally along the following probability density function [21]:

$$p(d) = \frac{1}{d\sigma\sqrt{2\pi}} e^{-((\ln d-\mu)/\sigma)^2/2}$$

  where $p(d)$ is the number of vertexes with in-degree d, μ and σ are the mean and the standard deviation, respectively, of the corresponding normal distribution.
- Edges are directed and the total number of edges is 447 million. Each vertex has 4 incoming edges in average and the standard deviation of the number of incoming edges is 1.3. For any vertex v_i, the rate of the incoming edges from the vertex in the sub-graph to which the vertex v_i belongs to the incoming edges from the vertex in other sub-graphs is 0.1. Consequently, the total number of edge-cuts between 128 sub-graphs is 44 million.

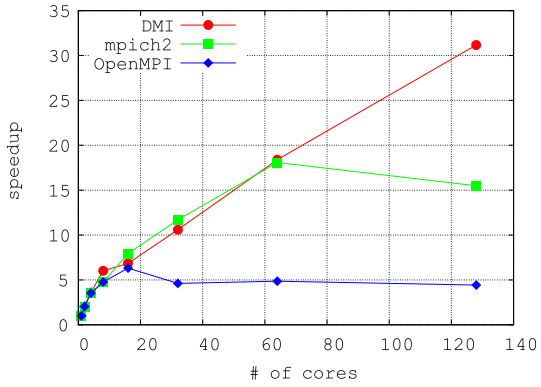


Fig. 15 The scalability of the pagerank calculation.

Let n be the total number of nodes in the web graph, v_i be a vertex (a web page or a person in a social graph), $adj^+(v_i)$ be a set of vertexes to which the vertex v_i links, and $adj^-(v_i)$ be a set of vertexes which have links to the vertex v_i. The pagerank of the vertex v_i is defined as the value of $rank(v_i, t)$ when the following recurrence formula converges [21]:

$$rank(v_i, t) = \begin{cases} 1/n & \text{if } t = 0, \\ 0.15/n + 0.85 \sum_{v_j \in adj^-(v_i)} rank(v_j, t-1)/|adj^+(v_j)| & \text{if } t \geq 1. \end{cases} \quad (1)$$

The experiment implements this iterative algorithm in DMI and MPI. In this pagerank calculation, the data to be checkpointed and restored are a current iteration number and the values of all vertexes.

First, **Fig. 15** shows the weak scalability of DMI (without any reconfiguration) and MPI. Figure 15 indicates that DMI outperformed much mpich2 and OpenMPI. The reason for this low performance of mpich2 and OpenMPI was that all-to-all communication performance of mpich2 and OpenMPI became poorer than DMI as the size of transferred data became larger. In each iteration of this pagerank calculation, in case of using 128 cores, each core sends about 21.5 KB of data to all other 127 cores, literally "all-to-all" dense communications. **Figure 17** shows the potential communication performance of DMI, mpich2 and OpenMPI, that is, the time required for each of 128 cores to send x bytes of data to all other 127 cores simultaneously. In MPI, we used 127 MPI_Send() and 127 MPI_Recv() for each process. In DMI, we used DMI_group_write() and DMI_group_read() for each thread, by which each thread sends data to other 120 threads[*3]. At this point, since DMI_group_write() and DMI_group_read() internally aggregates the data sent to the same node into one message, the number of messages triggered by each thread is reduced to $120/8 = 15$. Each point in Fig. 17 is the average time of 10 runs and the error bar of each point indicates the maxmum time and the minimum time in the 10 runs. Figure 17 indicates that OpenMPI's performance was noisy and in particular the performance of x=21.5 KB was better in the order of DMI, mpich2 and OpenMPI, which accounts for the low performance of mpich2 and OpenMPI in the pagerank calculation. We do not yet inves-

[*3] The number of threads to which the data needs to be sent through inter-node communication is $128 - 8 = 120$.

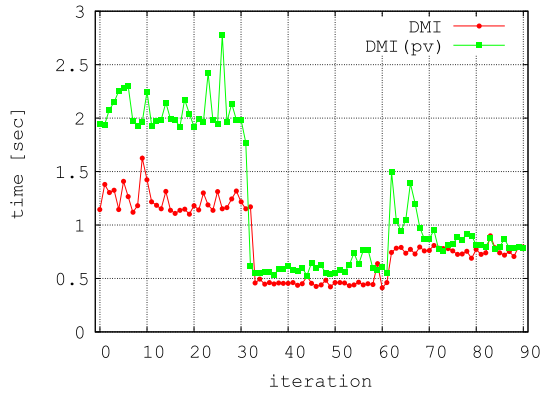


Fig. 16 The execution time of each iteration of the pagerank calculation with reconfiguration.

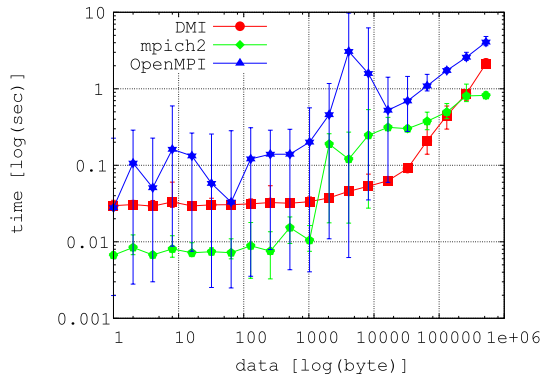


Fig. 17 The time of all-to-all communications using 128 cores (8 cores×16 nodes).

tigate the internal implemention of MPI_Send()/MPI_Recv() of mpich2 and OpenMPI, and thus not yet clarify the reason why DMI outperforms even mpich2 when data size is enough large.

Second, as with Fig. 11, **Fig. 16** shows the execution time of each iteration in DMI when we started the computation on 4 nodes, added 12 nodes at the end of (about) the 30-th iteration and then removed 8 nodes at the end of (about) the 60-th iteration. Figure 16 indicates that DMI can also adapt parallelism efficiently through reconfiguration for complicated and irregular iterative graph applications, and that the processor non-virtualization model outperforms better than the processor virtualization model. We can expect the similar results for many other iterative graph algorithms, such as the shortest path problem and connected components, because they can be formulated similar to the formula (1) and have the similar communication pattern.

Third, Table 1 shows the breakdown of the reconfiguration time. In fact, 0.976 GB of data is checkpointed and restored as the values of all vertexes.

Fourth, with respect to programmability, the lines of the MPI program was 738 and the lines of the DMI program without reconfiguration was 693. We also confirmed that adding only 29 lines to this DMI program made it reconfigurable. The difference between the 738 lines of DMI and the 693 lines of MPI was primarily attributed to whether a programmer can program the exchange of values between sub-graphs in a global-view or not. MPI program requires complicated and error-prone calculations of the indices from which the data should be sent and to which the data should be received, but in DMI the programmer can productively program just by reading/writing from/to the global addresses of each vertex value.

6. Related Work

SRS [26], DyRecT [8] and PCM [19], [20] are reconfigurable frameworks based on a message passing model and a processor non-virtualization model. In these frameworks a programmer can describe a reconfigurable MPI program by inserting into a normal MPI program APIs for registering the data to be checkpointed before a reconfiguration and APIs for restoring the data after the reconfiguration. In SRS, for example, by calling SRS_Register("foo", *a*, *size*, BLOCK,...) in advance, the block distributed data each block of which is a local array *a* of *size* bytes on each process is automatically checkpointed by the name of "foo" before the reconfiguration. Then, by calling SRS_Read("foo", *b*, BLOCK,...) after the reconfiguration, the data checkpointed by the name of "foo" is automatically redistributed by block among a set of new processes and then each block is restored to a local array *b* on each process. In summary, these frameworks re-distribute the data transparently if the programmer specifies the pointer from which the data should be checkpointed, the pointer to which the data should be restored and the type of the data distribution. Here note that checkpointing/restoring the data is expressed as a write/read operation using the "global" name of "foo," that is, a kind of global address space. In other words, we can consider that these frameworks provide the message passing model for the body of a parallel computation but the global address space model for checkpointing/restoring data at the reconfiguration, which implies that these frameworks have already achieved the global address space over the reconfiguration to some extent. However, this global address space is a very simple and limited one that can be used only for redistributing the data with regular structures at the reconfiguration just as the programmer describes, compared to the general and flexible global address space of DMI. In any way, these frameworks are based on the message passing model not a PGAS model and thus the programmer cannot enjoy good programmability of the PGAS model that we confirmed in Sections 5.3 and 5.4.

The framework by Scherer [25] is a reconfigurable framework based on a DSM model. In this framework the programmer can describe a reconfigurable OpenMP program on a distributed environment. This framework extends the cache coherence protocols of TreadMarks [27] so that it can support the global address space with cache coherence over the reconfiguration. Then the framework translates the OpenMP program described by the programmer into a reconfigurable TreadMarks program. However, this framework does not investigate any locality-aware techniques for improving the performance of the reconfigurable computations based on the PGAS model such as selective cache read/write.

Selective cache read/write is novel in that it realizes not only a hybrid cache protocol of an invalidate protocol and an update protocol at each read/write granularity but also get/put operations, which enables flexible optimization of access locality. Most existing PGAS frameworks such as Co-Array Fortran [28], UPC [16], Global Arrays [23], X10 [4] and Chapel [6] support only get/put operations and do not support cache and dynamic change of data

distribution. In contrast, most existing DSM frameworks such as TreadMarks [5], JIAJIA [13] and DSM-Threads [22] support cache but a programmer cannot freely mix the invalidate protocol and the update protocol. Furthermore, page-based DSMs, which manage coherency of a global address space depending on the memory protection mechanism of an operating system, cannot realize get operations in principle. This is because in the page-based DSMs, when a signal handler hooks a SIGSEGV signal caused by a read fault, the signal handler has to set the protection of the page to readable before the signal handler returns. Otherwise, the SIGSEGV will continue to be invoked forever. However, making the protection of the page readable implies that not only the read that is causing the read fault but also all subsequent reads are allowed. Consequently, this is not the get operation but equivalent to cache the page. Thus, since there is no way to allow only the read that is causing the current read fault as long as the page-based DSM depends on the memory protection mechanism, the page-based DSM cannot realize the get operations. DMI solves these problems by managing coherency at user level.

7. Conclusions

This paper proposes how to design and implement a global-view-based PGAS framework that enables easy programming of reconfigurable and high-performance parallel iterative computations. The contributions of this paper are described in Section 1.2. The novelty of this paper is that we designed and implemented a framework for parallel computational reconfiguration based on a global-view-based PGAS model and a processor non-virtualization model, supported by our observations that (1) a processor non-virtualization model is more suitable than a processor virtualization model because the performance of running each iteration is quite critical for the total performance of long-running iterative applications and that (2) the global-view-based PGAS model is more programmable than a message passing model for its potentially good programmability and especially for easy programming of the dynamic increase and decrease of threads because in the global-view-based PGAS model it is not the programmer but the framework that manages the complicated data locations. To the best of our knowledge, this is the first work that achieves reconfiguration based on a PGAS model. Selective cache read/write is also a novel elemental technique for adapting data distribution to the actually observed access patterns in reconfigurable computations. Finally, we evaluated DMI using not simple and regular benchmark applications but *irregular*, *real-world* and *large-scale* applications.

As our future work, we are planning to expand our programming interfaces so that DMI can support not only synchronous and iterative applications but also broader range of applications that allow asynchronous joining and leaving of nodes.

Reference

[1] T2K, available from ⟨http://www.cc.u-tokyo.ac.jp/⟩.

[2] The 2nd Parallel Programming Contest on Cluster Systems, available from ⟨https://www2.cc.u-tokyo.ac.jp/procon2009-2/⟩.

[3] TSUBAME2.0, available from ⟨http://www.gsic.titech.ac.jp/en⟩.

[4] The Cascade High Productivity Language, *9th International Workshop on High-Level Parallel Programming Models and Supportive Environments*, pp.52–60 (Apr. 2004).

[5] Amza, C., Cox, A.L., Dwarkadas, H., Keleher, P., Lu, H., Rajamony, R., Yu, W. and Zwaenepoel, W.: TreadMarks: Shared Memory Computing on Networks of Workstations, *IEEE Computer*, Vol.29, No.2, pp.18–28 (Feb. 1996).

[6] Charles, P., Grothoff, C., Saraswat, V., Donawa, C., Kielstra, A., Ebcioglu, K., von Praun, C. and Sarkar, V.: X10: An Object-oriented Approach to Non-uniform Cluster Computing, *Proc. 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, pp.519–538 (Oct. 2005).

[7] Chaudhary, V. and Jiang, H.: Techniques for Migrating Computations on the Grid, *Engineering the Grid: Status and Perspective*, pp.399–415 (Jan. 2006).

[8] Godard, E., Setia, S. and White, E.L.: DyRecT: Software support for adaptive parallelism on NOWs, *15th IPDPS Workshops on Parallel and Distributed Processing*, pp.1144–1151 (May 2000).

[9] Hara, K., Nakashima, J. and Taura, K.: A PGAS Framework Achieving Thread Migration Unrestricted by the Address Space Size (in Japanese), *IPSJ Transactions on Programming* (2011).

[10] Hara, K. and Taura, K.: A Global Address Space Framework for Irregular Applications (accepted, short paper), *High Performance Distributed Computing* (June 2010).

[11] Hara, K., Taura, K. and Chikayama, T.: DMI: A Large Distributed Shared Memory Interface Supporting Dynamically Joining/Leaving Computational Resources (in Japanese), *IPSJ Trans. on Programming*, Vol.3, No.1, pp.1–40 (2010).

[12] Bailey, D.H., Barszcz, E., Barton, J., Browning, D., Carter, R., Dagum, L., Fatoohi, R.A., Fineberg, S., Frederickson, P., Lasinski, T., Schreiber, R., Simon, H.D., Venkatakrishnan, V. and Weeratunga, S.: THE NAS PARALLEL BENCHMARKS, Technical Report, RNR-94-007 (Mar. 1994).

[13] Hu, W., Shi, W., Tang, Z. and Zhou, Z.: JIAJIA: An SVM System Based on a New Cache Coherence Protocol, Technical Report, Center of High Performance Computing Institute of Computing Technology Chinese Academy of Sciences (Jan. 1998).

[14] Huang, C., Lawlor, O. and Kale, L.V.: Adaptive MPI, *16th International Workshop on Languages and Compilers for Parallel Computing*, pp.306–322 (Oct. 2003).

[15] Huang, C., Zheng, G., Kale, L. and Kumar, S.: Performance Evaluation of Adaptive MPI, *11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp.12–21 (Mar. 2006).

[16] Husbands, P., Iancu, C. and Yelick, K.: A Performance Analysis of the Berkeley UPC Compiler, *Proc. 17th Annual International Conference on Supercomputing*, pp.63–73 (2003).

[17] Naik, V.K., Midkiff, S.P. and Moreira, J.E.: A Checkpointing Strategy for Scalable Recovery on Distributed Parallel Systems, *1997 ACM/IEEE Conference on Supercomputing*, pp.1–19 (Nov. 1997).

[18] Johnson, K.L., Kaashoek, M.F. and Wallach, D.A.: CRL: High-Performance All-Software Distributed Shared Memory, *Proc. 15th Symposium on Operating Systems Principles*, Vol.29, No.5, pp.213–228 (Mar. 1995).

[19] El Maghraoui, K., Desell, T.J., Szymanski, B.K. and Varela, C.A.: Dynamic Malleability in Iterative MPI Applications, *7th IEEE International Symposium on Cluster Computing and the Grid*, pp.591–598 (May 2007).

[20] El Maghraouia, K., Desella, T., Szymanskia, B.K., Terescob, J.D. and Varela, C.A.: Towards a Middleware Framework for Dynamically Reconfigurable Scietific Computing, *Advances in Parallel Computing*, Vol.14, pp.275–301 (2005).

[21] Malewicz, G., Austern, M.H., Bik, A.J.C., Dehnert, J.C., Horn, I., Leiser, N. and Czajkowski, G.: Pregel: A System for Large-scale Graph Processing, *Proc. 2010 International Conference on Management of Data*, pp.135–146 (2010).

[22] Mueller, F.: Distributed Shared-Memory Threads: DSM-Threads, *Workshop on Run-Time Systems for Parallel Programming*, pp.31–40 (Apr. 1997).

[23] Nieplocha, J., Palmer, B., Tipparaju, V., Krishnan, M., Trease, H. and Apra, E.: Advances, Applications and Performance of the Global Arrays Shared Memory Programming Toolkit, *International Journal of High Performance Computing Applications*, Vol.20, No.2, pp.203–231 (2006).

[24] Sankaran, S., Squyres, J.M., Barrett, B., Sahay, V. and Lumsdaine, A.: The Lam/Mpi Checkpoint/Restart Framework: System-Initiated Checkpointing, *International Journal of High Performance Computing Applications*, Vol.19, pp.479–493 (2005).

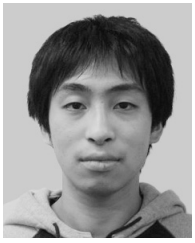
[25] Scherer, A., Lu, H., Gross, T. and Zwaenepoel, W.: Transparent Adaptive Parallelism on NOWs Using OpenMP, Vol.34, No.8, pp.96–106 (Aug. 1999).

[26] Vadhiyar, S.S. and Dongarra, J.J.: SRS: A Framework for Developing Malleable and Migratable Parallel Applications for Distributed Systems, *International Journal of High Performance Applications and Supercomputing*, pp.291–312 (June 2003).

[27] Taura, K., Endo, T., Kaneda, K. and Yonezawa, A.: Phoenix: A Parallel Programming Model for Accommodating Dynamically Joining/Leaving Resources, *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp.216–229 (2003).

[28] Numrich, R.W. and Reid, J.: Co-array Fortran for parallel programming, *ACM SIGPLAN Fortran Forum*, Vol.17, No.2, pp.1–31 (1998).

[29] Yelick, K., Hilfinger, P., Graham, S., Bonachea, D., Su, J., Kamil, A., Datta, K., Colella, P. and Wen, T.: Parallel Languages and Compilers: Perspective from the Titanium Experience, *Journal of High Performance Computing Applications*, Vol.21, No.3, pp.266–290 (2007).



Kentaro Hara was born in 1986. He received his M.E. degree from the Graduate School of Information Science and Technology, the University of Tokyo in 2011. He has been working as a software engineer at Google since 2011.

Kenjiro Taura was born in 1969. He received his Ph.D. degree from the Graduate School of Science, the University of Tokyo in 1997. He researched in the Graduate School of Science at the University of Tokyo as an assistant professor from 1996 to 2001. After that, he has been researching in the Graduate School of Information Science and Technology as a lecturer from 2001 to 2002 and then as an associate professor since 2002.