Journal of Information Processing Vol.20 No.1 89–103 (Jan. 2012)

[DOI: 10.2197/ipsjjip.vol.20.89]

Regular Paper

Half-process: A Process Partially Sharing Its Address Space with Other Processes

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

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

Abstract: Threads share a single address space with each other. On the other hand, a process has its own address space. Since whether to share or not to share the address space depends on *each* data structure in the whole program, the choice of "a thread or a process" for the *whole* program is too much "all-or-nothing." With this motivation, this paper proposes a *half-process*, a process partially sharing its address space with other processes. This paper describes the design and the kernel-level implementation of the half-process and discusses the potential applicability of the half-process for multi-thread programming with thread-unsafe libraries, intra-node communications in parallel programming frameworks and transparent kernel-level thread migration. In particular, the thread migration based on the half-process is the first work that achieves transparent kernel-level thread migration by solving the problem of sharing global variables between threads.

Keywords: partitioned global address space, Linux kernel, thread migration, finite element method, reconfiguration, productivity

1. Introduction

Threads share a single address space with each other. When one thread issues memory operations such as mmap()/munmap()/mprotect(), all threads can see the effects immediately. When one thread updates a memory address, all threads can see the update immediately. In addition, all threads can access the same data with the same address. With these convenient semantics, a programmer can easily describe data sharing between the threads. However, in exchange for this convenience, sharing the single address space between the threads often makes it difficult to describe correct parallel programs. The programmer has to take great care of complicated race conditions and non-deterministic behaviors [6]. Needless to say, the programmer cannot use thread-unsafe libraries in his program. To avoid these difficulties, the programmer can instead use not a thread but a process, which has its own address space, but in this case the programmer cannot enjoy the convenient semantics of data sharing based on the single address space. Obviously, whether to share or not to share the address space depends on *each* data structure in the whole program. Therefore, the choice of "a thread or a process" for the *whole* program is too much "all-or-nothing." It is preferable that the programmer can flexibly choose whether (not all but) *each* data structure should be shared or not individually. In this way, the total goal of our work is to develop a mechanism that provides processes with a shared address space semantically equivalent to the single address space between threads. By this mechanism, the programmer can enjoy both advantages of process and thread, depending on how the programmer wishes to share each data structure.

[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

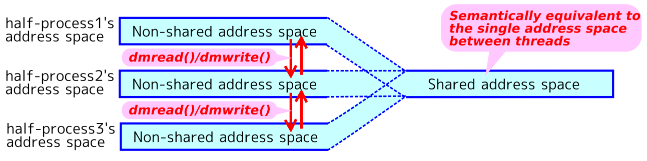

half-process1's address space
Non-shared address space
dmread()/dmwrite()
half-process2's address space
Non-shared address space
dmread()/dmwrite()
half-process3's address space
Non-shared address space
Shared address space
Semantically equivalent to the single address space between threads

Fig. 1 The design of a half-process.

With this motivation, this paper proposes a *half-process*, a process partially sharing its address space with other processes. As shown in **Fig. 1**, each half-process consists of a *non-shared address space* and *a shared address space*. The shared address space is totally shared with other half-processes, *semantically equivalent* to the single address space between threads. While all global variables and the memory allocated by normal mmap() are allocated on the non-shared address space, the programmer can allocate memory on the shared address space *just* by using mmap() with a special option. Although this kind of shared address space can be implemented using mmap() at the user level, the half-process implements the shared address space at the kernel level, which provides programmers with easier way for inter-process communication on the shared address space. In addition, the half-process provides the method for direct memory access between non-shared address spaces for performance and convenience.

The rest of this paper is organized as follows. In Section 2 we discuss the essential difference between the shared address space of the half-process and inter-process shared memory. In Section 3 we describe the potential applicability of the half-process. In Section 4 we describe the design and kernel-level implementation of the half-process. In Section 5 we apply the half-process to transparent kernel-level thread migration and evaluate the performance of the half-process using *real-world* parallel applications such as PageRank calculation and an finite element method.

2. Related Work

The total goal of our work is to achieve a shared address space semantically equivalent to the single address space between threads, so that a programmer can enjoy both advantages of process and thread. As we mentioned in Section 1, the two primary techniques of the half-process are the shared address space between half-processes and the direct memory access between address spaces.

2.1 A Shared Address Space between Processes

Although there are several possible semantics for designing the shared address space, we consider that the shared address space is desirable to be semantically equivalent to the single address space between threads. This is because (1) in the beginning the shared address space is intended to be a space for easy data sharing, (2) one of the easy ways for data sharing is the single address space between threads as we indicated at the head of Section 1, and (3) a lot of programmers have been familiar with the semantics of the single address space between threads.

Here we discuss the essential difference between the shared address space of the half-process and inter-process shared memory. For detailed discussion, let us first clarify the technical semantics of the shared address space between half-processes, which is semantically equivalent to the single address space between threads. The semantics I is that *all meta-operations such as mmap()/munmap()/mprotect() issued by any half-process must be immediately reflected in all half-processes*. The semantics II is that *all the half-processes must be able to access the same data with the same address*. In other words, the semantics I means that when one half-process issues a meta-operation, the result of the meta-operation must be visible to all the half-processes immediately after the meta-operation returns, and the semantics II means that if one half-process can access data x with an address a then all the half-processes must be able to access the data x also with the address a.

In POSIX inter-process shared memory [*1], the programmer creates the inter-process shared memory with shm_open() and sets the size of the inter-process shared memory with ftruncate(). Then by letting multiple processes map the inter-process shared memory into their address spaces with mmap() with MAP_SHARED option, the programmer can use the inter-process shared memory between the processes. The key observation here is that a process p cannot use the inter-process shared memory unless the process p issues mmap() by itself. In other words, if the programmer wishes to make the process p allocate shared data between the multiple processes, the programmer has to make not only the process p but also all the processes issue mmap() somehow. This programming interface is inconvenient. Thus, the inter-process shared memory does not meet the semantics I in that meta-operations issued by any process are not automatically reflected in other processes. Furthermore, there is no guarantee that the programmer can map the inter-process shared memory to the same addresses between all the processes (without any special trick such as negotiating with other processes about which addresses are available on each process). If the mapped addresses are different between the processes, the programmer must manage pointers to the inter-process shared memory by not the mapped address but the offset from the start address of the map. This programming interface dramatically degrades the programmability for complicated data structures such as a graph. Thus, the inter-process shared memory does not meet the semantics II.

One easy way to meet the semantics II using the inter-process shared memory is that the programmer initially maps a sufficient size of the inter-process shared memory with mmap() with MAP_SHARED option and then spawns child processes with fork() [21]. Since the spawned child processes inherit the map of the parent process, all the spawned child processes are guaranteed to map the inter-process shared memory into the same addresses. The semantics I, however, does not still hold. Thus the spawned child processes cannot easily issue mmap() and munmap() after once they are spawned, which means that the once physically-allocated inter-process shared memory must continue to exist until the end of a program, lacking flexibility. For example, assume that the parent process allocates a large size of inter-process shared memory with mmap() with MAP_SHARED option and then spawns child processes. At this point, this mmap() does not consume physical memory. Then, assume that one of the child processes allocates 32 GB of memory (with malloc() or something) from the inter-process shared memory. At this point, 32 GB of physical memory is consumed. Next, assume that the child process is going to deallocate the 32 GB of memory (with free() or something). Here, note that the child process cannot internally issue munmap() to the inter-process shared memory, since there is no easy way to reflect munmap() to other processes (because of the lack of the semantics I). In this way, the child process can just mark that the 32 GB of memory is available so that the memory can be reused in the future, but the child process can never internally deallocate the physical memory. This means that we cannot avoid that the 32 GB of physical memory is continued to be alive, even if the program does not need the 32 GB of memory any more. Incidentally, the system call madvise() with MADV_DONTNEED option enables such physical memory to be released from the process's address space. The system call declares that the process no longer needs specified memory pages. However, this system call is effective only in the process that issues it. In other words, if other processes also have accessed the same shared memory, the physical memory allocated to them is not released. Consequently, even madvise() is not satisfactory. In essence, physically-allocated inter-process shared memory must continue to exist until the end of the program.

In summary, it is difficult to allocate/deallocate and expand/shrink the inter-process shared memory dynamically. In addition, with respect to the semantics I, there is no easy way to reflect the memory protection change by mprotect() to other

[*1] The similar discussion holds for System V inter-process shared memory.

processes. Thus, the semantics of the inter-process shared memory is much less convenient than that of the shared address space between half-processes, which is equivalent to that of the single address space between threads.

2.2 Direct Memory Access between Address Spaces

There has been much work for the direct memory access between address spaces primarily in the context of improving the intra-node communication performance of MPI, such as Nemesis [7], [9], Limic2 [26] and other works [15], [28]. The most popular approach is to create the system call that copies data between address spaces [15], [26], [28]. Specifically, (1) a kernel pins the target address range of a source address space to physical memory, (2) the kernel maps the range to a kernel address space and (3) the kernel copies data from the kernel address space to the target address range of a destination address space. Actually, as discussed in Section 4.3.5, our approach is the same as this approach (and thus has no novelty with respect to the direct memory access). More optimizations are possible by overlapping the pinning and copying [15] or by offloading CPUs using I/OAT DMA engine [7], [15], [28]. SMARTMAP [5], [6] enables the fast direct memory access at the user level by allowing a programmer to address another process's virtual addresses with a simple offset calculation, but it depends on the special lightweight kernel that uses the linear mapping from virtual addresses to physical pages without demand-paging and maps the same executable image to the identical virtual addresses across all processes.

3. Potential Applicability

Before introducing the detailed design and implementation of a half-process, we discuss the potential applicability of the half-process.

The first applicability is multi-thread programming with thread-unsafe or thread-inefficient libraries. Some libraries are thread-unsafe. Even if they are thread-safe, they sometimes perform poorly when used by multiple threads because of too coarse-grained locks or false sharing of data. For example, assume that a programmer is going to describe parallel graph algorithm, which requires complicated handling of pointers, with the support of the thread-unsafe or thread-inefficient libraries. In this case, by using the half-process, the programmer can not only allocate the graph structures on the shared address space and enjoy the convenient semantics of data sharing for his graph algorithm but also use the thread-unsafe or thread-inefficient libraries on the non-shared address space.

The second applicability is flexible hybrid programming [32]. In existing hybrid programming frameworks based on MPI+OpenMP or MPI+pthread, instances [*2] inside one MPI process are implemented as threads because currently the thread is the only way to realize the shared address space. However, since the threads share the shared address space completely, the programmer cannot use the thread-unsafe or thread-inefficient libraries any longer. Note that in the first place, practical hybrid programming itself does not require that all data is shared between the instances inside one MPI process but just require that only necessary data is shared between the instances. At this point, the half-process can be useful to build a more flexible hybrid programming framework that enables the programmer to choose what data to be shared between the instances.

The third applicability is reducing the burden on the framework programmers [*3] of parallel programming frameworks. Ignoring hybrid programming for the moment, each instance of MPI is usually implemented as a process partly because an address space must be independent between the instances in order to give the application programmer a consistent view of global variables. In other words, if each instance is implemented as a thread, whether or not the global variables are shared between the threads depends on whether or not the threads exist on the same process, which cannot be naturally distinguished by the application programmer view. Therefore, each instance of MPI is usually implemented as a process and thus the framework programmers of MPI have challenged how they can improve the performance of intra-node communications between the MPI processes using inter-process shared memory and the direct memory access across the MPI processes [5], [6], [7], [9], [15], [26], [28]. As discussed in Section 2, however, it is difficult to dynamically allocate/deallocate or expand/shrink the inter-process shared memory. Thus, the following way has been used for intra-node communication methods in most existing frameworks [10], [28]:

(1) Initially, a framework allocates the fixed and relatively small size of inter-process shared memory for each pair of MPI processes. Namely, ${}_nC_2$ inter-process shared memories are allocated in total, where n is the number of the MPI processes. Here we denote the inter-process shared memory for process i and process j by $m_{i,j}$.

(2) When the size of the message that the process i sends to the process j is small enough, $m_{i,j}$ is used for this communication. Specifically, the process i copies the message from the process i's address space to $m_{i,j}$, and then the process j copies the message from $m_{i,j}$ to the process j's address space.

(3) When the size of the message that the process i sends to the process j is large, several ways are possible. The first way is that the process i divides the whole message into the small chunks each size of which fits in $m_{i,j}$ and then copies those chunks to the process j's address space using $m_{i,j}$ in a pipelined manner. The second way is that the process i (or the process j) copies the message directly from the process i's address space to the process j's address space by using the direct memory access between processes. The third way is that (1) the process i allocates a new inter-process shared memory m' of the message size, (2) the process i copies the message from the process i's address space to m', (3) the control data that tells the head address of m' is sent to the process j through $m_{i,j}$, and (4) the process j copies the message from m' to the process j's address space.

[*2] In this paper we refer to a process or a thread or a half-process as an *instance*.

[*3] In this paper we refer to the programmer who develops applications by using some kind of programming frameworks like MPI or DMI as an *application programmer*, and refer to the programmer who develops the framework itself as a *framework programmer*.

Thus, if a framework uses the inter-process shared memory, extra memory copy is required or the framework programmer of the framework has to describe complicated data sharing. In addition, efficient implementation of collective communications will require more complicated data sharing between processes. In contrast, if a shared address space between the processes is provided by the half-process, the framework programmer can implement those intra-node communications more simply thanks to the convenient semantics of the shared address space. In essence, by using the half-process, the framework programmer can easily describe efficient intra-node communications based on the shared address space, as well as the application programmer can still describe his program based on the non-shared address space.

The fourth applicability is transparent kernel-level thread migration. We discuss this in detail in Section 5.

4. Half-process: A Process Partially Sharing Its Address Space with Other Processes

4.1 Design

Figure 1 shows the design of a *half-process*. The whole address space of each half-process consists of a *non-shared address space*, the private address space to the half-process, and a *shared address space*, the address space shared with all half-processes. This shared address space is *semantically equivalent* to the single address space between threads and meets the semantics I and II. In other words, if a half-process p allocates a memory on the shared address space, then the half-process p can pass the address of the memory to another half-process q (through some memory on the shared address space), and then the half-process q can access the memory using that address, equivalent to the way that a programmer normally does in multi-threaded programming. Here we refer to the set of half-processes that share their shared address spaces with each other as a *half-process set*. In particular, if a half-process set includes only one half-process, the half-process is just a normal process. By default, the half-process behaves as a normal process using only the non-shared address space, namely, all global variables and the memory allocated by mmap() are allocated on the non-shared address space. In order to allocate data on the shared address space, the programmer can use mmap() with MAP_HALFPROC option. In addition, the programmer can copy data directly between non-shared address spaces of two half-processes in the same half-process set. Specifically, dmread(pid_t `pid`, void *`ptr`, size_t `size`, void *`buf`) copies data directly from the range [`ptr`, `ptr+size`) of the half-process with process id `pid` to the range [`buf`, `buf+size`) of the half-process that issued this dmread(). Similarly, dmwrite(pid_t `pid`, void *`ptr`, size_t `size`, void *`buf`) copies data directly from the range [`buf`, `buf+size`) of the half-process that issued this dmwrite() to the range [`ptr`, `ptr+size`) of the half-process with process id `pid`. These dmread() and dmwrite() are possible only between two processes inside the same half-process set.

In summary, by using half-processes, the programmer can describe intra-node communications easily using the shared address space or directly copying data between the non-shared address space.

The half-process set is defined by the option of clone(). In

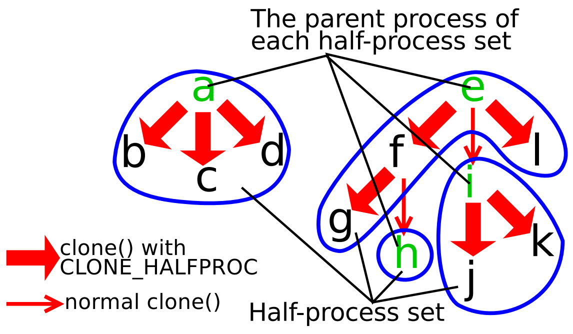

The parent process of each half-process set
a
b
c
d
e
f
g
h
i
j
k
l
clone() with CLONE_HALFPROC
normal clone()
Half-process set

Fig. 2 Examples of half-process sets.

Linux, if the programmer issues clone() with CLONE_VM option, then the spawned child process shares the address space of the parent process and thus the programmer can create a thread, and if the programmer issues clone() without CLONE_VM option, then the spawned child process uses a new address space and thus the programmer can create a process. Similarly, if the programmer issues clone() with CLONE_HALFPROC option, the programmer can create a half-process. More precisely, the half-process spawned by clone() with CLONE_VM option belongs to the half-process set that includes only the half-process (and thus this half-process becomes just a normal process), and the half-process spawned by clone() with CLONE_HALFPROC option belongs to the same half-process set of the parent half-process. **Figure 2** shows examples of the relationships of half-process sets.

With respect to security issues, note that the shared address space can be constructed only among half-processes inside the same half-process set and that dmread()/dmwrite() can be also used only among half-processes inside the same half-process set. Furthermore, each half-process p in a half-process set s can be added to the half-process set s only by forking the half-process p with CLONE_HALFPROC option from any half-process in the half-process set s. This means that the shared address space and dmread()/dmwrite() are allowed only among the half-processes that are explicitly allowed to use the shared address space and dmread()/dmwrite() with each other. In other words, it is impossible for a stranger process to violate the protection of half-processes by abusing the shared address space or dmread()/dmwrite().

4.2 Motivation for Kernel-level Implementation

Here we explain the reason why we implement a half-process not in the user level but in the kernel level. First, dmread() and dmwrite() require the kernel-level implementation because the default Linux kernel does not allow a user-level direct memory access over different address spaces. Second, with respect to the shared address space between half-processes, the user-level implementation based on inter-process shared memory is possible in the following way: (1) we can allocate a sufficient size of the inter-process shared memory with shm_open(), (2) we can meet the semantics I by hooking all meta-operations such as mmap()/munmap()/mprotect() for the shared address space and then somehow let all the half-processes issue the meta-operation synchronously, and (3) we can meet the semantics II by carefully coordinating the address mapped by mmap() so that all the half-processes can map memory into the same address. This user-

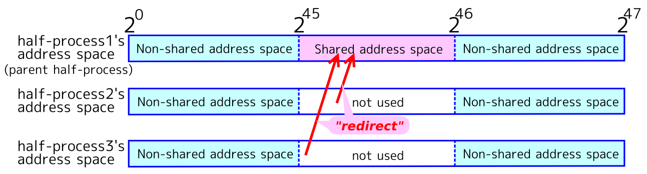


Fig. 3 The address range partitioning for a shared address space and a non-shared address space.

level implementation, however, causes overhead proportional to the number of half-processes for every meta-operation, because all the half-processes need to issue the meta-operation based on synchronizations and negotiations between all the half-processes. We consider that this overhead is critical, especially in case of a large number of half-processes, because these meta-operations are very primitive and used regularly by many applications. Thus we adopt the kernel-level implementation explained below.

4.3 Linux Kernel Modifications

We describe the kernel-level implementation of the half-process based on the Linux 2.6 kernel and the x86_64 architecture.

4.3.1 The Basic Idea

First, we statically fix the address ranges of a non-shared address space and a shared address space. Specifically, since in the X86_64 architecture we can use the range $[0, 2^{47})$ as a user address space range, we use the range $[2^{45}, 2^{46})$ as a shared address space range and the remaining range $[0, 2^{45})$ and $[2^{46}, 2^{47})$ as a non-shared address space range [*4] (**Fig. 3**). Second, we let each half-process use the address space of the half-process for the non-shared address space range, and let each half-process use the *parent half-process*' address space for the shared address space range. Here the parent half-process of a half-process p is defined as the half-process that is the first member of the half-process set to which the half-process p belongs. In other words, there is exactly one parent half-process for each half-process set and the parent half-process is the half-process that firstly spawned another half-process in the half-process set (See Fig. 2). As a result, the shared address space range of the non-parent half-processes is not used at all.

Obviously, the key implementation issue here is how to "redirect" the memory operations (i.e., meta-operations and normal read/write accesses) for the shared address space of each half-process into the memory operations for the shared address space of the parent half-process. To address this issue, we propose the novel implementation techniques, *address space switching* for redirecting meta-operations and *page table redirection* for redirecting normal read/write accesses. By these two redirections, we can meet the semantics I and II, respectively. Note that since the target of these redirections is only the memory operations for the shared address space range, this kernel modification does not affect the behaviors of the normal applications that do not use any functions of the half-process, except that an available address range is limited to $[0, 2^{45})$ and $[2^{46}, 2^{47})$.

[*4] The reason for this "enclave" address partitioning is that since the first range [0, ...) and the last range [..., 2^{47}) are implicitly used for non-shared special data such as a code segment and a machine stack, we have to handle these address ranges as a non-shared address space range.

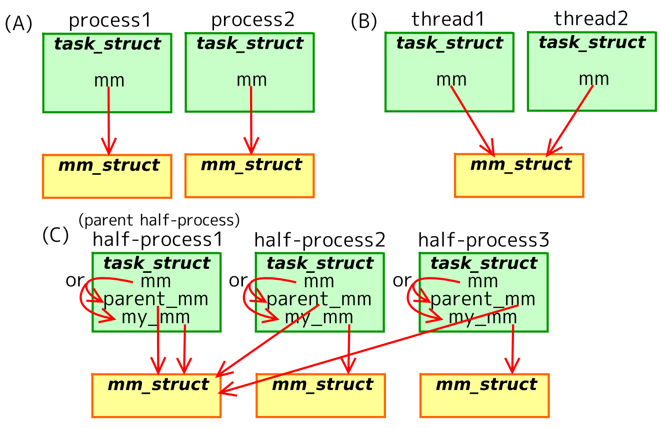


Fig. 4 The relationships between `task_struct` and `mm_struct` for (A) a process, (B) a thread and (C) a half-process.

4.3.2 Address Space Switching

Address space switching is a technique for redirecting meta-operations for the shared address space into the desired address space.

In the default Linux kernel, a process/thread is represented as a `task_struct` structure, and an address space is represented as an `mm_struct` structure. Each `task_struct` has an `mm` member, the pointer to the `mm_struct` that represents the address space of the process/thread represented by the `task_struct` (**Fig. 4** (A)(B)). This `mm_struct` manages meta-operations for its address space, for example which addresses are currently mapped and which kind of memory protection is applied to each address range. Since a half-process originally has to behave as a normal process, each half-process is also represented by a `task_struct` structure with its own `mm_struct`.

The purpose here is to switch the `mm_struct` of the half-process to the `mm_struct` of the parent half-process for all the meta-operations targeted for the shared address space range. The key observation is that the address space to which the meta-operation is applied is determined by the `mm` member of the `task_struct` issuing the meta-operation at the time, hence by switching the `mm` member of the `task_struct` to another `mm_struct` at the head of the meta-operation, we can switch the address space to which the meta-operation is applied. To do this, first, we add two members, `my_mm` and `parent_mm`, to the `task_struct` of the half-process (Fig. 4 (C)). The `my_mm` member points to the `mm_struct` of the half-process, the `parent_mm` member points to the `mm_struct` of the parent half-process, and the `mm` member points to the target `mm_struct` of the meta-operation running at the time. If the half-process is spawned by clone() without CLONE_HALFPROC option, both the `my_mm` member and the `parent_mm` member are initialized to the `mm_struct` of the half-process. If the half-process is spawned by clone() with CLONE_HALFPROC option, the `my_mm` member is initialized to the `mm_struct` of the half-process and the `parent_mm` member is initialized to the `parent_mm` member of the parent half-process [*5] of the clone(). Thus we can express the half-process relationships shown in Fig. 2. Second, we modify the kernel code for meta-operations (mmap(), munmap(), mpro-

[*5] Here "the parent half-process" means not the parent half-process of a half-process set but the parent half-process of the clone().

tect(), msync(), mbind(), page fault handling and so on) as shown in **Fig. 5**. At the head of the meta-operation targeted for the shared address space, we switch the `mm` member to the `parent_mm` member, flush TLBs and load the page table of the parent half-process. At the end of the meta-operation we switch the `mm` member back to the `my_mm` member, flush the TLBs and load the page table of this half-process. No modification to the kernel code for the meta-operation itself is required. By this *simple* switching, the kernel executes the meta-operation as if the address space of the half-process temporarily becomes the address space of the parent half-process.

4.3.3 Page Table Redirection

Page table redirection is a technique for redirecting normal read/write accesses for the shared address space into the desired address space by redirecting page table entries.

The x86_64 architecture realizes a 48 bit address space with 4 level page tables. Each level table has 512 entries. In particular, each 4th level (top level) entry manages $2^{48}/512 = 2^{39}$ addresses. In the default Linux kernel, each `mm_struct` has the `pgd` member that points to the head address of the 4th level table. At every context switch from `task_struct` t to `task_struct` t', if $t\to\mathtt{mm}\neq t'\to\mathtt{mm}$, a CPU flushes TLBs and loads the page table of t' by loading the $t'\to\mathtt{mm}\to\mathtt{pgd}$ into %cr3 register of the CPU.

The purpose here is to redirect the page table entries of the shared address space range $[2^{45}, 2^{46})$ of the half-process into the corresponding page table entries of the parent half-process as shown in **Fig. 6**. In this way all accesses to the shared address space range by any half-process are automatically redirected to the parent half-process by the CPU. However, constructing the structure of this page table redirection is not trivial in Linux because the page table is constructed dynamically on demand in response to a page fault. Here assume that a half-process causes a page fault at the shared address the 4th, 3rd, 2nd and 1st level page table entry, i_4, i_3, i_2 and i_1, respectively. We denote by t the `task_struct` of the half-process and denote by t' the `task_struct` of the parent half-process. First, we check whether the 4th level page table exists or not by checking $t\to\mathtt{mm}\to\mathtt{pgd}$, and if not, we allocate the 4th level page table as $t\to\mathtt{mm}\to\mathtt{pgd}$. Second, we check whether the 3rd level page table exists or not by checking $t\to\mathtt{mm}\to\mathtt{pgd}[i_4]$, and if not, we allocate the 3rd level page table as $t'\to\mathtt{mm}\to\mathtt{pgd}[i_4]$ and update $t\to\mathtt{mm}\to\mathtt{pgd}[i_4]$ with $t'\to\mathtt{mm}\to\mathtt{pgd}[i_4]$. As a result, $t\to\mathtt{mm}\to\mathtt{pgd}[i_4]{=}{=}t'\to\mathtt{mm}\to\mathtt{pgd}[i_4]$ holds. Third, we check whether the 2nd level page table exists or not by checking $t'\to\mathtt{mm}\to\mathtt{pgd}[i_4][i_3]$, and if not, we allocate the 2nd level page table as $t'\to\mathtt{mm}\to\mathtt{pgd}[i_4][i_3]$. Fourth, we check whether the 1st level page table exists or not by checking $t'\to\mathtt{mm}\to\mathtt{pgd}[i_4][i_3][i_2]$, and if not, we allocate the 1st level page table as $t'\to\mathtt{mm}\to\mathtt{pgd}[i_4][i_3][i_2]$. Thus, we have to modify two page tables at the page fault but no TLB flushing is necessary during and after this page table modification because the page related to the modified page table entries has not existed before this modification.

4.3.4 Improvement of Copy-on-write

No data is allocated on the shared address space range of a non-parent half-process. Thus, when clone() is issued, we do not have to trap copy-on-write in the shared address space range and thus can reduce the overhead of clone().

4.3.5 A Direct Memory Access Over Address Spaces

dmread(pid_t `pid`, void *`ptr`, size_t `size`, void *`buf`) is implemented as follows (dmwrite() is similar) [15], [26], [28]. First, we pin the range [`ptr`, `ptr+size`) of the half-process with process id `pid`. Second, we map the pinned range into a kernel address space by kmap. Third, we copy data directly from the mapped kernel address space to the range [`buf`, `buf+size`) of the current half-process.

4.4 Performance Evaluations

4.4.1 Experimental Settings

We used the machine that has two Intel Xeon E5530 (2.4 GHz, 4 physical cores but 8 logical cores with hyper-threading, 4 × 256 KB of L2 cache and 8 MB of L3 cache) CPUs, 24 GB

```
some_meta_operation(uint64_t addr, ...) {
    struct task_struct *current = the task_struct running now
    if addr is in the shared address space then
        current→mm = current→parent_mm
        flush TLBs
        load the page table of current→mm to the CPU
    endif
    ... /* the unmodified kernel code of this meta operation */
    if addr is in the shared address space then
        current→mm = current→my_mm
        flush TLBs
        load the page table of current→mm to the CPU
    endif
}
```

Fig. 5 An algorithm for address space switching.

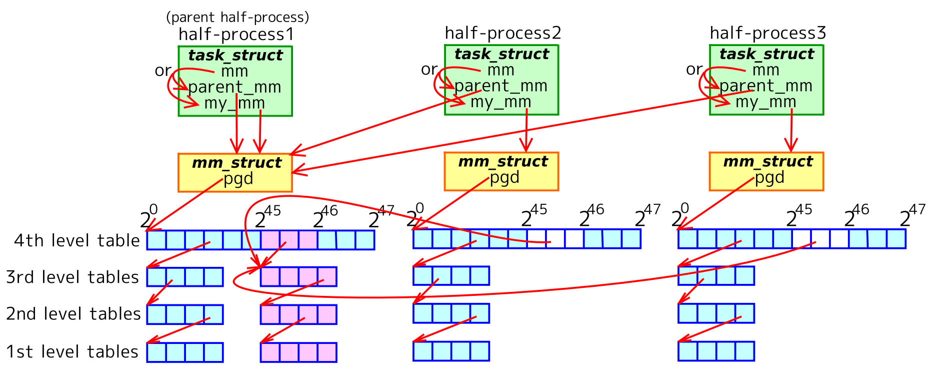


Fig. 6 A mechanism of page table redirection.

of memory, running the Linux OS with the modified 2.6.26-2-amd64 kernel.

4.4.2 Inter-process Data Copy

We evaluate the performance of several strategies [8] for copying data between two half-processes located on the same CPU (**Fig. 7**) or the different CPU (**Fig. 8**). Specifically, (1) one half-process p writes data d to some memory location l, (2) we start a timer, (3) the other half-process p' reads the data d and calculates the check sum of the data d and (4) we stop the timer. In Fig. 7 and Fig. 8, "socket" means that l is the non-shared address space of p and p sends d via a loopback socket to p'. "pipe" means that l is the non-shared address space of p and p sends d via an anonymous pipe to p'. "half-process" means that l is the shared address space between p and p' and p' just reads d from the shared address space. "inter-process" means that l is the POSIX inter-process shared memory between p and p' and p' just reads d from the inter-process shared memory. "dmread" means that l is the non-shared address space of p and p' reads d using dmread(). "doublecopy" means that l is the non-shared address space of p and firstly p writes d to inter-process shared memory and secondly p' reads d from the inter-process shared memory, similar to the intra-node communications in MVAPICH2 and OpenMPI [7], [15].

We can observe that, first, "socket" and "pipe" are much slower than "half-process" for the small size of data and 1.17∼1.96 times slower for 2^{28} bytes of data. Second, "half-process" and "inter-process" perform equally well since they both use shared memory. Third, for 2^{28} bytes of data, "dmread" performs 1.22 times better than "doublecopy" in Fig. 7 and 1.26 times better than "doublecopy" in Fig. 8. Fourth, "dmread" is slower than "doublecopy" in Fig. 7 for less than 2^{16} of data, despite the fact that "dmread" requires only one memory copy but "doublecopy" requires two memory copies. This is considered to be because "dmread" requires kernel context switching but "doublecopy" does not require it. Fifth, the reason why in Fig. 8 "dmread," which requires kernel-level data copy before p' reads and calculates the check sum, performs almost equal to "half-process," which just reads and calculates the check sum, is that both performances are dominated not by the overhead of kernel operations but by the cost of inter-CPU memory accesses.

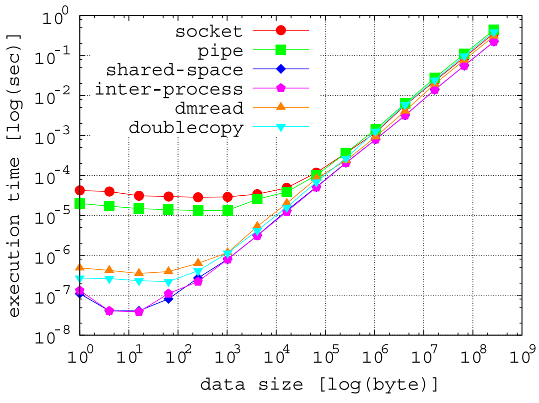


Fig. 7 The performance comparison of several strategies for copying data between half-processes on the same CPU.

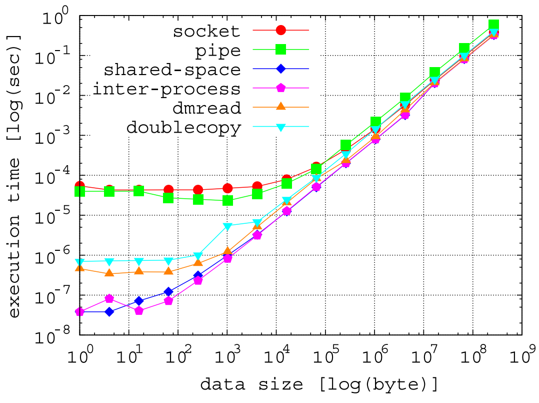


Fig. 8 The performance comparison of several strategies for copying data between half-processes on different CPUs.

4.4.3 The Overhead of Address Space Switching and Page Table Redirection

Next, we evaluate the overhead of address space switching and page table redirection. 10,000,000 mmap()s for the non-shared address space, which requires no address space switching, took 2.531 seconds and 10,000,000 mmap()s for the shared address space took 2.654 seconds. 1,000,000 page faults for the non-shared address space, which requires no page table redirection, took 1.316 seconds and 1,000,000 page faults for the shared address space took 1.327 seconds. Thus, the overhead is pretty small (while frequent TLB flushing may degrade the total performance of realistic applications).

5. Application to Transparent Kernel-level Thread Migration

5.1 Motivation

Thread migration [3], [4], [11], [13], [19], [20], [22], [23], [24], [25], [27], [30] is a significant elemental technique for dynamic load balancing, computational migration for improving data locality and parallel computational reconfiguration. Here, assume the framework whose process structure of each node is shown in **Fig. 9**. There is a receiver thread, which receives messages from another node, there is a controller thread, which schedules thread migration, and there are multiple computational threads, which execute the computation described by the application programmer and are the targets of the thread migration. Although many frameworks have investigated the thread migration in C, the thread migration causes two critical problems.

The first problem is that of address collision because of thread migration. A thread's memory (i.e., stack and heap) can include pointers to the thread's memory itself. Thus in order to continue the thread execution correctly over the thread migration, the thread's memory has to be allocated on the exactly identical addresses between a source node and a destination node. However, without any preparation in advance, there is no guarantee that the addresses used by the thread's memory on the source node are

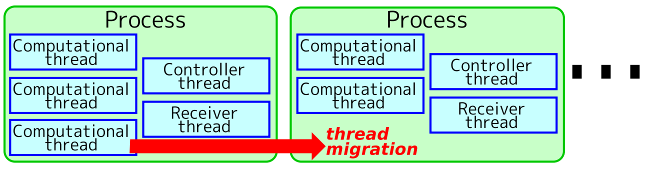


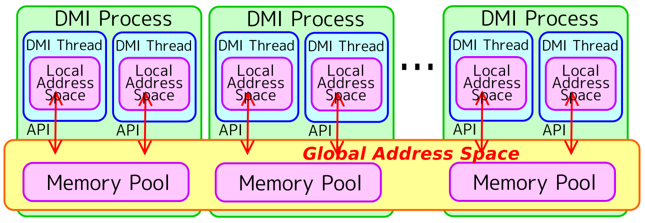
Fig. 9 A model of thread migration frameworks.

available at the destination node. One way to solve this address collision is iso-address [3], [4], [30], which divides the whole address space into many small address spaces and assigns a different small address space to each thread in advance. This guarantees that the addresses used by all threads never overlap and thus the address collision problem never happens at the thread migration. However, this iso-address limits computation scale depending on the whole address space size. In the iso-address $ns = w$ must hold, where n is the number of threads, s is the address space size available for each thread and w is the whole address space size. Thus, even in today's x86_64 architecture where w is 2^{47}, if n is 16384 then s is just 32 GB, and if s is 512 GB then n is just 1,024, which can be practical numbers in the foreseeable future. Another way to solve the address collision problem is to allow the thread's memory to be allocated on different addresses between the source node and the destination node by translating all pointers in the thread's memory correctly at the thread migration [11], [22], [23], [24], [25]. However, complete translation is impossible without restricting the syntax of C because C is not a type-safe language.

The second problem is address space dependence between threads, which is a potential problem of the thread migration. Threads inside one process share a single address space, especially global variables. Thus when one of the threads in the process migrates, it *inevitably* results in inconsistency about the global variables whether the global variables of the process are migrated with the thread or not. Furthermore, an application programmer cannot use the global varibles as he expects, namely as "global variables per thread," because the global variables are actually shared with other threads in the same process. Note that the application programmer cannot even know which threads are in the same process because the framework should migrate the threads transparently. With these backgrounds, most existing frameworks such as Adaptive MPI [19], [20], MigThread [23], [24], [25], PM2 [3], [4] and Arachne [13] simply prohibit the use of the global variables in the first place, which implies that the application programmer cannot even use safely any libc functions that may use the global variables such as printf(), malloc() and many other basic functions. Tern [27] allows the use of the global variables by converting the global variables into TLSes (thread local storages) at the pre-processing phase, but it cannot support the global variables used in already compiled libraries. Thus, no other thread migration framework can solve the problem of sharing global variables.

Considering these facts, in order to achieve *transparent* (i.e., without any inconvenient restriction on the syntax of C and without any annotations by the application programmer) kernel-level thread migration [*6], the address space of each thread must not be shared. In other words, each instance in Fig. 9 must be implemented not a thread but a process. However, replacing the thread in Fig. 9 with a process can be harder work for the framework programmer, as the framework internally requires tighter intra-node communications between the threads. For example, the receiver thread has to transmit received data to a destination thread and

[*6] Here we consider only the migration of the thread's memory. The migration of file descriptors and sockets is out of the scope of this paper.



```
DMI_read(addr, size, buf, ...) : read size bytes from addr to buf
DMI_write(addr, size, buf, ...) : write size bytes from buf to addr
addr : address in a global address space, buf : address in a local address space
```

Fig. 10 The system structure of DMI.

the controller thread has to coordinate all computational threads. Furthermore, in case that this framework provides a PGAS (Partitioned Global Address Space) model for the application programmer and manages shared data cache between the threads (i.e., the remote data once accessed by any thread is cached into the process and subsequent accesses to the data by another thread in the process can use the cached data), more complicated data sharing is required between the threads. If each instance is a thread, the framework programmer can easily program this complicated data sharing by malloc()/free()/read/write with appropriate synchronizations thanks to the convenient semantics of a single address space. On the other hand, if each instance is a process, it is much more difficult for the framework programmer to program the complicated data sharing using inter-process shared memory as discussed in Section 2.

In summary, in the thread migration framework the application programmer requires a non-shared address space but the framework programmer requires a shared address space between instances. This is exactly the case where a half-process finds its value.

5.2 Parallel Computational Reconfiguration

We apply a half-process to *DMI* (Distributed Memory Interface) [17], [18], which is originally a parallel computational reconfiguration framework based on a multi-threaded PGAS model. In DMI, an application programmer only has to create a sufficient number of threads and describe a parallel computation easily on the basis of a global-view-based PGAS model, similar to multi-thread programming on a physically shared memory environment. Then DMI automatically reconfigures (i.e., scales up or down) the computation in response to the dynamic joining or leaving of nodes by transparently migrating the threads on available nodes at the time. The reason why we originally implemented each instance of DMI as a thread is that we needed the convenient semantics of a single address space for implementing sophisticated functions (overviewed below) with complicated data sharing between instances. Thus, previously in DMI the application programmer cannot use global variables and thus cannot use any libraries that may use the global variables for safe thread migration. In this paper we eliminate this inconvenient limitation and realize transparent kernel-level thread migration by replacing a thread with a half-process.

5.2.1 Framework Overview

To begin with, we overview the original multi-threaded DMI (**Fig. 10**). See more details in our publications [17], [18]. First,

```
void DMI_main(int argc, char **argv) {
  ...;
  for(i = 0; i < thread_num; i++) /* create threads */
    DMI_create(&handle[i], arg_addr + i * sizeof(arg_t));
  ...;
  for(i = 0; i < thread_num; i++) { /* retrieve the threads */
    DMI_join(handle[i]);
  ...;
}
uint64_t DMI_thread(uint64_t arg_addr) { /* each thread */
  ...;
  for(iter = 0; /* until convergence */; iter++) {
    DMI_yield();
    ...;
  }
}
```

Fig. 11 Programming interface of DMI.

DMI provides a global address space with cache coherence. Each process provides DMI with some amount of memory called a *memory pool*, and then DMI constructs the global address space over these distributed memory pools by implementing memory management mechanisms such as page tables at the 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. Specifically, the application programmer can explicitly specify the cache coherence granularity and how the transferred page should be cached (no-cache or invalidate-cache or update-cache) at every read/write, and thus optimize the performance of accessing the global address space explicitly and powerfully. 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 accomplished efficiently through physical shared memory. In this way the application 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 can maintain the cache coherence over the dynamic joining and leaving of processes during one computation and thus scales up or down the parallel computation. As shown in **Fig. 11**, the application programmer only has to create a sufficient number of threads using DMI_create() and call DMI_yield() periodically at short intervals in each thread. Then DMI transparently migrates the thread inside the DMI_yield() if necessary, and thus scales up or down the computation in response to the dynamic change of available nodes. In essence, we emphasize that the modification required to make a normal non-reconfigurable DMI program reconfigurable is *just adding DMI_yield()* to the DMI program. We also emphasize that if each instance of DMI is implemented as a half-process, no inconvenient syntax restriction is imposed on the application programmer as discussed in Section 5.1.

DMI is implemented as a static library for C 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 memory allocation/deallocation, memory read/write, asynchronous memory read/write, a mutual exclusion, collective synchronization, user-defined atomic instruction, aggregating multiple discrete reads/writes, expressing irregularly decomposed domains and so on [18].

5.2.2 Replacing a Thread with a Half-process

The first essential modification to the original DMI is replacing pthread_xxxx() with corresponding halfproc_xxxx(). A half-process cannot use glibc's pthread library because the pthread library assumes that global variables and stack variables are shared between instances but the half-process places global variables and stack variables on the non-shared address space. Thus, we implement halfproc_mutex_xxxx() and halfproc_cond_xxxx() not using any global variables and stack variables but using a futex() system call instead. We also implement halfproc_create()/halfproc_join()/halfproc_detach(), which creates/joins/detaches a half-process respectively.

The second essential modification is replacing malloc()/realloc()/free() with halfproc_malloc()/halfproc_realloc()/halfproc_free() because the normal malloc()/realloc()/free() internally issue mmap() without MAP_HALFPROC option. We implement halfproc_malloc()/halfproc_realloc()/halfproc_free() so that they internally issue mmap() with MAP_HALFPROC option.

The third modification is replacing normal memcpy() with dmread()/dmwrite() from/to the non-shared address space to/from the address space of another half-process. For example, assume that a half-process p running on a node n issues DMI_read(`addr`, `size`, `buf`, ...), which reads `size` bytes of data from the global address `addr` to the half-process p's address space `buf`. If the data does not exist on the memory pool of the node n, a read fault occurs and the data is transferred to the node n from the node that owns the data at the time. This data is received by the receiver half-process on the node n (c.f., Fig. 9). Then, the receiver half-process stores this data to the non-shared address space of the half-process p by using dmwrite() instead of normal memcpy() since the receiver half-process cannot normally access the non-shared address space of the half-process p.

The fourth modification is replacing the checkpoint/restart mechanism of a thread with the checkpoint/restart mechanism of a half-process, in which only the memory allocated on the non-shared address space is checkpointed/restarted.

Consequently, we need these several simple modifications but need not modify the basic data structures and the data sharing algorithm used in the original DMI. Below, we refer to the original multi-threaded version of DMI as DMI+thread and the half-process version of DMI as DMI+half-process.

5.3 Performance and Programmability Evaluations

In this sub-section, we evaluate the performance of DMI+half-process using *real-world* parallel applications such as PageRank calculation and an finite element method.

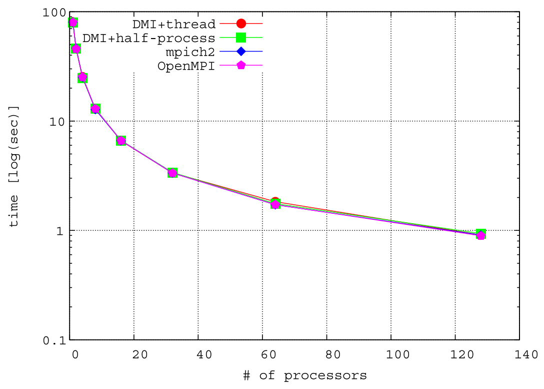


Fig. 12 The execution time of the N-body problem.

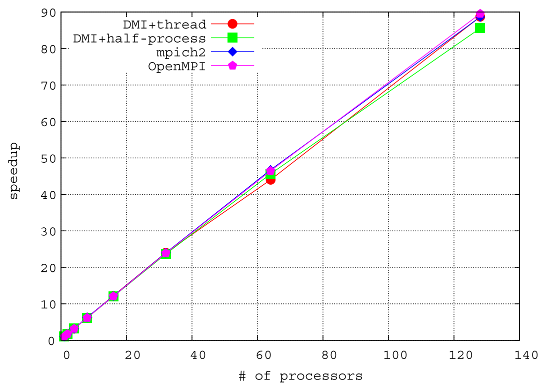


Fig. 13 The weak scalability of the N-body problem.

5.3.1 Experimental Settings

The experimental platform is the cluster environment composed of 16 nodes interconnected by 10 Gbit Ethernet. The architecture of each node is described in Section 4.4.1. 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/MPI program using n ($1 \leq n \leq 128$) cores, we created 8 threads/half-processes/processes on $\lfloor n/8 \rfloor$ nodes and the remaining $n - 8 \times \lfloor n/8 \rfloor$ threads/half-processes/processes on another node.

5.3.2 An N-body Problem

This experiment solves an N-body problem. The experiment divides $24 \times 24 \times 24$ particles evenly in the direction of the z-axis among n processors and assigns $24 \times 24 \times 24/n$ particles to each processor. Each processor repeats below: (1) each processor sends the positions of the particles assigned to the processor to all other processors, namely allgather communication in total, by which all the processors can know the positions of all the particles, and (2) each processor concurrently calculates the interaction between the particles assigned to the processor and all the particles and then updates the positions and velocities of the particles assigned to the processor.

First, in order to compare the basic performance of DMI+half-process with other frameworks, **Fig. 12** shows the execution time of mpich2, OpenMPI, DMI+thread (without any reconfigration) and DMI+half-process (without any reconfigration). **Figure 13** shows their weak scalability. 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. Figures 12 and 13 indicate that DMI+half-process performed equally well to DMI+thread, in other words, DMI+half-process caused little overhead. Thus, the half-process enables not only convenient but also high-performance intra-node communications. Figures 12 and 13 also indicate that DMI+half-process performed equally well to mpich2 and OpenMPI.

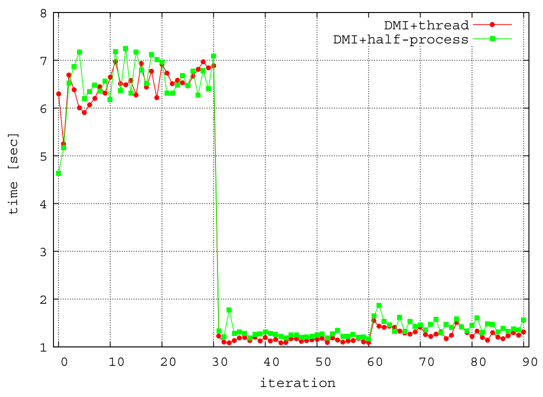


Fig. 14 The execution time of each iteration of the N-body problem with reconfiguration.

Table 1 Each reconfiguration time and transferred data size at each reconfiguration.

	N-body	PageRank	FEM
time(DMI+thread, 32→128 cores) [sec]	5.84	8.66	10.1
time(DMI+thread, 128→64 cores) [sec]	1.02	5.42	5.80
transferred data [GB]	0.671	13.7	15.2
time(DMI+half-process, 32→128 cores) [sec]	8.12	33.4	19.8
time(DMI+half-process, 128→64 cores) [sec]	3.39	8.85	7.46
transferred data [GB]	0.940	14.2	14.4

Second, **Fig. 14** shows the execution time of each iteration in DMI+thread and DMI+half-process, where we initially spawned 128 threads, changed available nodes dynamically, and let DMI+thread and DMI+half-process reconfigure their computational scale through thread migration. Specifically, (1) we started the computation on 4 nodes (32 cores in total, 4 threads per core), (2) added 12 nodes (128 cores in total, 1 thread per core) at the end of the 30-th iteration, and then (3) removed 8 nodes (64 cores in total, 2 thread per core) at the end of the 60-th iteration. Actually, it is impossible to migrate a thread in DMI+thread, if the thread is using global variables or any address in the stack and heap memory of the thread is already used in the destination process. To address these issues, first, we described the program without any global variables. Second, we introduced special memory allocation/deallocation APIs (DMI_thread_mmap()/DMI_thread_munmap()) that guarantee to allocate/deallocate the stack and heap memory so that its memory address never conflicts with the stack and heap memory of all other threads in all nodes [17]. When the thread migrates in DMI+thread, we migrate only the memory allocated by DMI_thread_mmap() into the destination process.

Figure 14 indicates that DMI+half-process performed equally well to DMI+thread, which implies that DMI+half-process caused little overhead, and that both DMI+thread and DMI+half-process can *adapt the application parallelism efficiently* to the increase and decrease of available nodes through reconfiguration.

Third, **Table 1** shows each reconfiguration time in Fig. 14

Table 2 The number of lines of code change required to convert a non-reconfigurable DMI program to a reconfigurable DMI program.

	N-body	PageRank	FEM
DMI+thread	5	18	62
DMI+half-process	1	1	1

(Nbody), Fig. 18 (PageRank) and Fig. 22 (FEM). Table 1 also shows the total size of transferred data through thread migration at each reconfiguration. The total size of transferred data was equal for both cases of 32→128 cores and 128→64 cores, since 120 threads were migrated for both cases [*7]. The difference of the transferred data size between DMI+thread and DMI+half-process comes from the difference of what data is checkpointed in their checkpointing algorithms. For example, global variables are checkpointed in DMI+half-process but are not checkpointed in DMI+thread (since in the first place DMI+thread assumes that no global variables are used in a program). The reason why DMI+half-process took more reconfiguration time than DMI+thread in Table 1, which cannot be explained just by the difference of the transferred data size, will be shown in the next subsection.

Fourth, to compare the programmability of DMI+half-process with that of DMI+thread, **Table 2** shows the number of lines of code change that is required to convert a non-reconfigurable normal DMI program to a reconfigurable DMI program. In DMI+thread, we needed to add DMI_yield() (one line) and convert four mmap()/munmap()s to DMI_thread_mmap()/DMI_thread_munmap() (four lines). Furthermore, we needed to take great care so that all the execution states of each thread exist on the memory allocated by DMI_thread_mmap() at the point of DMI_yield(), so that all the execution states migrate at the thread migration. Specifically, we needed to care that no global variables are used directly or indirectly (through glibc libraries) and the memory of each thread does not contain any pointers to another thread's memory [*8]. On the other hand, in DMI+half-process, we needed to add *only one line of code* (DMI_yield()) at the head of each iteration. Also, no coding care about global variables was necessary.

5.3.3 PageRank Calculation

This experiment calculates a PageRank [29] 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 vertices is 128 million.
- The entire graph is composed of 128 sub-graphs, each of which has 1 million vertices.
- The in-degrees of vertices are distributed log-normally along the following probability density function [29]:

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

  where $p(d)$ is the number of vertices 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.

[*7] Naturally, the number of threads migrated at reconfiguration depends on thread scheduling policy but the scheduling policy is out of the scope of this paper

[*8] For fair comparison of the number of lines of code change, we describe the original non-reconfigurable DMI program so that it uses no global variables and any pointers to other threads' memory. In other words, the number of lines of code change shown in Table 2 does not reflect the "care" that we have to take in DMI+thread.

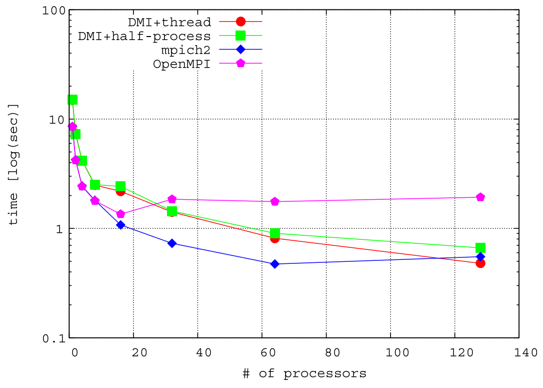


Fig. 15 The execution time of the PageRank calculation.

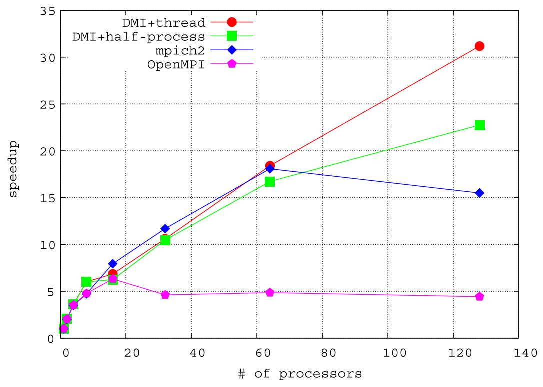


Fig. 16 The weak scalability of the PageRank calculation.

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 vertices to which the vertex v_i links, and $adj^-(v_i)$ be a set of vertices 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 [29]:

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

The experiment implements this iterative algorithm in DMI and MPI.

First, **Fig. 15** shows the execution time of mpich2, OpenMPI, DMI+thread (without any reconfiguration) and DMI+half-process (without any reconfiguration), and **Fig. 16** shows their weak scalability. Figures 15 and 16 indicate that DMI+half-process performed worse than DMI+thread when 128 cores are used. The detailed performance profiling revealed that this overhead primarily comes from the following two reasons.

The first reason is the overhead of dmwrite() issued by the receiver half-process of DMI. As we mentioned in Section 5.2.2, DMI+half-process has to issue dmread()/dmwrite() instead of memcpy() when a half-process reads/writes another half-process's address space. In particular, DMI has only one receiver half-process for one node, which receives messages from another node and handles the messages one by one. Here the receiver half-process has to use dmwrite() when the receiver half-process writes received data to a computational half-process's address space. For example, assume that a read fault happens at a computational half-process of a node *n*. The read fault is notified to an owner node of the page, and the owner node transfers the latest page to the receiver half-process of the node *n*. Then the receiver half-process writes the received latest page to the computational half-process's address space. At this point, the receiver half-process needs to use dmwrite(). The reader may think it strange why the receiver half-process needs to use dmwrite() instead of memcpy() on the shared address space. This is because the target address space is not the shared address space but the computational half-process's address space. Specifically, when DMI_read(`addr`, `size`, `buf`, ...) is issued by the computational half-process, if the `buf` belongs to the shared address space, then the receiver half-process can indeed use normal memcpy() to copy the received latest page to the `buf`. However, if the `buf` belongs to the computational half-process's address space, then the receiver half-process needs to use dmwrite(). Note that "whether an application programmer should place `buf` on the shared address space or the computational half-process's address space" is determined not by "whether the application programmer wants to let the receiver half-process use dmwrite() or memcpy()" but by "whether the `buf` should be migrated or not when the half-process migrates to another node." In the first place, in this experiment we are not thinking of the situation that the application programmer places `buf` on the shared address space, because (1) most buffers allocated on each half-process must be specific to the half-process and thus should be migrated at reconfiguration and because (2) we are assuming that we allow to add only one line of code (DMI_yield()) for making the program reconfigurable [*9]. Getting back to the topic of the overhead, in this way, the receiver half-process needs to use dmwrite(). And the memory copy of dmwrite() is much slower than the memory copy of memcpy(), since context switch to the kernel is associated with dmwrite() in order to achieve direct memory access between different address spaces. In addition, the number of dmwrite()s increases, as the number of messages that the receiver half-process has to handle increases, in other words, as the number of computational half-processes (i.e., cores) increases. This is the reason why the overhead of DMI+half-process became larger, as the number of cores increased in Fig. 15 and Fig. 16. It is considered that we can reduce this overhead by increasing the number of receiver half-processes and thus reducing the bottle-neck of handling received messages. Also, the primary reason why DMI+half-process took more reconfiguration time than DMI+thread in Table 1 can be explained by the overhead of dmwrite()s issued by the single receiver half-process. At each reconfiguration, the receiver half-process needs to copy much received data to each computational half-process's address space.

[*9] Placing `buf` on the shared address space requires more changes in the program.

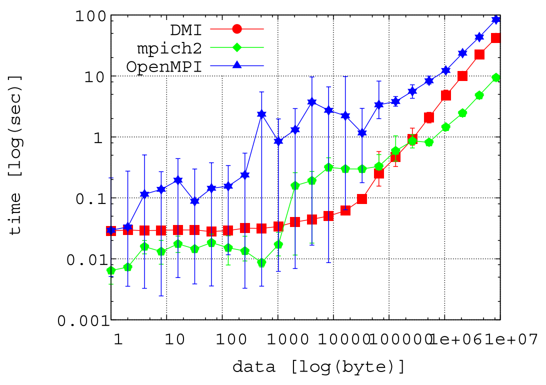


Fig. 17 The time of all-to-all communications using 128 cores (8 cores × 16 nodes).

The second reason is that we implemented halfproc_malloc()/halfproc_realloc()/halfproc_free() with Kernighan & Richie's malloc algorithm [31], the performance of which is poor in a multi-threaded program. It is considered that we can solve this problem by implementing more sophisticated malloc algorithm for halfproc_malloc()/halfproc_realloc()/halfproc_free(), such as Tcmalloc [1] and Hoard [12]. More detailed quantitative discussions about this overhead of DMI+half-process are described in another paper [16].

Figures 15 and 16 also indicate that DMI+thread and DMI+half-process outperformed much mpich2 and OpenMPI. The reason for this low performance of mpich2 and OpenMPI is that the communication performance of mpich2 and OpenMPI becomes poor as socket connections become dense. Specifically, in each iteration of this PageRank calculation, in case of using 128 cores, each core sends approximately 21.5 KB of data to all other 127 cores, literally an "all-to-all" dense communication. **Figure 17** shows the potential communication performance of DMI+thread, 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. Each point in Fig. 17 is the average time of 10 runs and the error bar of each point shows the maximum time and the minimum time in the 10 runs. Figure 17 indicates that OpenMPI's performance was pretty noisy and in particular the performance of x = 21.5 KB was better in the order of DMI+thread, mpich2 and OpenMPI, which accounts for the low performance of mpich2 and OpenMPI in the PageRank calculation.

Second, **Fig. 18** shows the execution time of each iteration in DMI+thread and DMI+half-process, where we initially spawned 128 threads and changed available nodes dynamically in the same manner as explained in Section 5.3.2. Figure 18 indicates that both DMI+thread and DMI+half-process can adapt the application parallelism efficiently to the increase and decrease of the available nodes through reconfiguration. However, the performance of DMI+half-process is worse than that of DMI+thread because of the overhead of DMI+half-process that we mentioned before.

Third, with respect to programmability, we needed to change 18 lines of code in DMI+thread, taking care that all the execu-

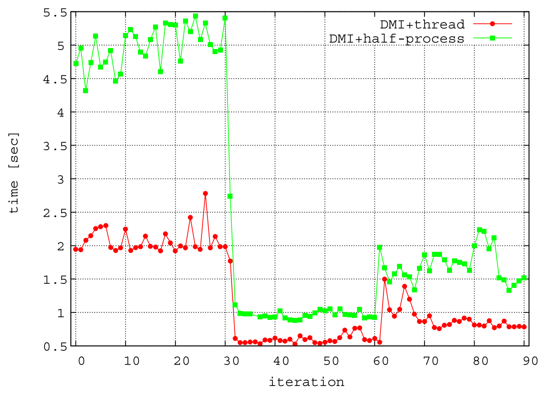


Fig. 18 The execution time of each iteration of the PageRank calculation with reconfiguration.

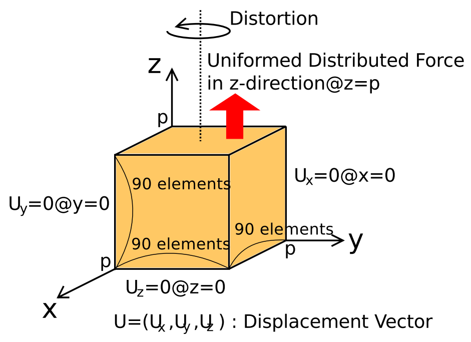

Distortion
Uniformed Distributed Force in z-direction@z=p
z
p
90 elements
90 elements
90 elements
U_x=0@x=0
U_y=0@y=0
U_z=0@z=0
p
p
y
x
U=(U_x,U_y,U_z) : Displacement Vector

Fig. 19 Stress analysis using the finite element method.

tion states exist on the memory allocated by DMI_thread_mmap() at the point of DMI_yield(). On the other hand, in DMI+half-process, only one line of code (DMI_yield()) at the head of each iteration was required.

5.3.4 An Finite Element Method

This experiment analyzes the stress of the 3-dimensional cube with the force and the boundary conditions shown in **Fig. 19** 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 the 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 to solve. Omitting details, we use 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 the preconditioning using the blocked ILU decomposition with fill-ins, which is the champion algorithm of the competition.

First, **Fig. 20** shows the execution time of mpich2, OpenMPI, DMI+thread (without any reconfiguration) and DMI+half-process (without any reconfiguration), and **Fig. 21** shows their weak scalability. Figures 20 and 21 indicate that DMI+half-process performed worse than DMI+thread because of the overhead of DMI+half-process. Figures 20 and 21 also indicate that even DMI+thread achieved lower scalability than mpich2. This

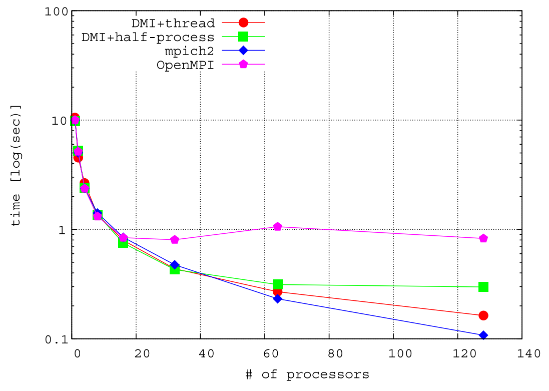


Fig. 20 The execution time of the finite element method.

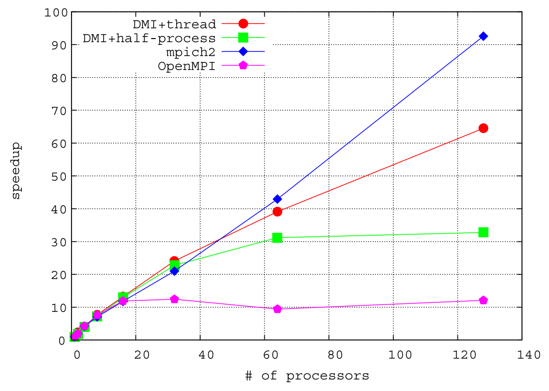


Fig. 21 The weak scalability of the finite element method.

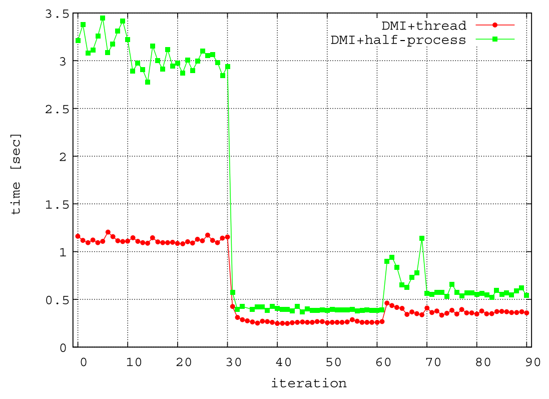


Fig. 22 The execution time of each iteration of the finite element method with reconfiguration.

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. Furthermore, Figs. 20 and 21 indicate that OpenMPI performed pretty poor compared to other frameworks. 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, **Fig. 22** shows the execution time of each iteration in DMI+thread and DMI+half-process, where we initially spawned 128 threads and changed available nodes dynamically in the same manner as explained in Section 5.3.2.

Third, with respect to programmability, we needed to change

62 lines of code in DMI+thread with the coding care. On the other hand, in DMI+half-process, only one line of code (DMI_yield()) at the head of each iteration was required.

5.3.5 Discussion

As we discussed, *one of* the potential applicabilities of a half-process is transparent kernel-level thread migration, solving the problem of sharing global variables between migrating kernel-level threads in existing frameworks. In this section, we applied the kernel-level thread migration to the programming framework that achieves computational reconfiguration just by creating a sufficient number of half-processes. As a result, DMI+half-process indeed delivers good programmability for reconfiguration. However, as far as the performance of such a reconfigurable framework is concerned, considering the fact that multiple half-processes can run on one core when the scale of a computation shrinks and that the overhead of the context switching between half-processes (this is almost equivalent to that between normal processes) is larger than that between kernel-level threads and much larger than that between user-level threads [14], a user-level thread with the support of transparent migration is considered to be a better choice than a half-process for the reconfigurable framework.

6. Conclusions

The choice of "a thread or a process" is too much "all-or-nothing." This paper proposed a half-process, which has both the non-shared address space privatized to the half-process and the shared address space between half-processes. Unlike inter-process shared memory, the half-process realizes the shared address space semantically equivalent to the single address space between threads. We discussed the potential applicability of the half-process for multi-thread programming with thread-unsafe libraries, intra-node communications in parallel programming frameworks and transparent kernel-level thread migration. In particular, we applied the half-process to the transparent kernel-level thread migration based on a PGAS programming framework named DMI and confirmed that *just* by adding DMI_yield() to a normal non-reconfigurable DMI program makes the DMI program reconfigurable without no inconvenient syntax restriction about using global variables. Also we evaluated the performance of the half-process using *real-world* parallel applications such as PageRank calculation and an finite element method. To the best of our knowledge, the design and the kernel-level implementation of the half-process are novel. Also, this is the first work that solves the problem of sharing global variables between migrating kernel-level threads and thus achieves transparent kernel-level thread migration.

Reference

[1] TCMalloc (online), available from ⟨http://goog-perftools.sourceforge.net/doc/tcmalloc.html⟩.

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

[3] Antoniu, G., Bouge, L. and Namyst, R.: An Efficient and Transparent Thread Migration Scheme in the PM2 Runtime System, *Proc. IPPS/SPDP'99 Workshops Held in Conjunction with the 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing*, pp.496–510 (1999).

[4] Antoniu, G. and Perez, C.: Using Preemptive Thread Migration to Load-Balance Data-Parallel Applications, *Proc. 5th International Euro-Par Conference on Parallel Processing*, pp.117–124 (1999).

[5] Brightwell, R. and Pedretti, K.: Optimizing Multi-core MPI Collectives with SMARTMAP, *2009 International Conference on Parallel Processing Workshops*, pp.370–377 (Sep. 2009).

[6] Brightwell, R., Pedretti, K. and Hudson, T.: SMARTMAP: Operating system support for efficient data sharing among processes on a multi-core processor, *Proc. 2008 ACM/IEEE Conference on Supercomputing*, pp.1–12 (Nov. 2008).

[7] Buntinas, D., Goglin, B., Goodell, D., Mercier, G. and Moreaud, S.: Cache-Efficient, Intranode, Large-Message MPI Communication with MPICH2-Nemesis, *2009 International Conference on Parallel Processing*, pp.462–469 (Sep. 2009).

[8] Buntinas, D., Mercier, G. and Gropp, W.: Data Transfers between Processes in an SMP System: Performance Study and Application to MPI, *2006 International Conference on Parallel Processing*, pp.487–496 (Aug. 2006).

[9] Buntinas, D., Mercier, G. and Gropp, W.: Design and Evaluation of Nemesis, a Scalable, Low-Latency, Message-Passing Communication Subsystem, *6th IEEE International Symposium on Cluster Computing and the Grid*, pp.521–530 (May 2006).

[10] Chai, L., Hartono, A. and Panda, D.K.: Designing High Performance and Scalable MPI Intra-node Communication Support for Clusters, *2006 IEEE International Conference on In Cluster Computing*, pp.1–10 (Sep. 2006).

[11] Cronk, D., Haines, M. and Mehrotra, P.: Thread Migration in the Presence of Pointers, *Proc. 30th Hawaii International Conference on System Sciences: Software Technology and Architecture*, Vol.1, pp.292–302 (1997).

[12] Berger, E.D., Mckinley, K.S., Blumofe, R.D. and Wilson, P.R.: Hoard: A Scalable Memory Allocator for Multithreaded Applications, *Proc. 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp.117–128 (Dec. 2000).

[13] Dimitrov, B. and Reg, V.: Arachne: A portable threads system supporting migrant threads on heterogeneous network farms, *IEEE Trans. on Parallel and Distributed Systems*, Vol.9, pp.459–469 (May 1998).

[14] Anderson, T.E., Bershad, B.N., Lazowska, E.D. and Levy, H.M.: Scheduler Activations: Effective Kernel Support for the User-level Management of Parallelism, *ACM Trans. on Computer Systems*, Vol.10, No.1 (Feb. 1992).

[15] Goglin, B.: High Throughput Intra-Node MPI Communication with Open-MX, *2009 Parallel, Distributed and Network-based Processing*, pp.173–180 (Feb. 2009).

[16] Hara, K.: A PGAS Programming Framework for Reconfigurable and High-Performance Parallel Computations, *Master thesis, the University of Tokyo* (Feb. 2011).

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

[18] 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*, 3 (2010).

[19] 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).

[20] 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).

[21] Duell, J.: Pthreads or Processes: Which is Better for Implementing Global Address Space Languages?

[22] Jiang, H. and Chaudhary, V.: Compile/Run-Time Support for Thread Migration, *Proc. 16th International Parallel and Distributed Processing Symposium*, pp.58–66 (2002).

[23] Jiang, H. and Chaudhary, V.: MigThread: Thread Migration in DSM Systems, *International Conference on Parallel Processing*, p.581 (2002).

[24] Jiang, H. and Chaudhary, V.: On Improving Thread Migration: Safety and Performance, *Proc. 9th International Conference on High Performance Computing*, pp.474–484 (2002).

[25] Jiang, H. and Chaudhary, V.: Thread Migration/Checkpointing for Type-Unsafe C Programs, *International Conference on High Performance Computing*, Vol.2913, pp.469–479 (Nov. 2003).

[26] Jin, H.-W., Sur, S., Chai, L. and Panda, D.K.: Lightweight kernel-level primitives for high-performance MPI intra-node communication over multi-core systems, *2007 IEEE International Conference on Cluster Computing*, pp.446–451 (Sep. 2007).

[27] Ke, J. and Speight, E.: Tern: Thread Migration in an MPI Runtime

Environment, Technical Report, Cornell (Nov. 2001).

[28] Lai, P., Sur, S. and Panda, D.K.: Designing Truly One-sided MPI-2 RMA Intra-node Communication on Multi-core Systems, *International Computing Conference*, Vol.25, pp.3–14 (May 2010).

[29] 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).

[30] Milton, S.: Thread Migration in Distributed Memory Multicomputers, Technical Report, Australia National University (1998).

[31] Richie, D.M. and Kernighan, B.W.: *The C Programming Language*, Prentice Hall (1990).

[32] Rabenseifner, R., Hager, G. and Jost, G.: Hybrid MPI/OpenMP Parallel Programming on Clusters of Multi-Core SMP Nodes, *2009 Parallel, Distributed and Network-based Processing*, pp.427–436, (Feb. 2009).

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.