Process-in-Process: Techniques for Practical Address-Space Sharing

Atsushi Hori[†]
RIKEN
ahori@riken.jp

Min Si[†]
Argonne National Laboratory
msi@anl.gov

Balazs Gerofi
RIKEN
bgerofi@riken.jp

Masamichi Takagi
RIKEN
masamichi.takagi@riken.jp

Jai Dayal
Intel Corp.
jai.dayal@intel.com

Pavan Balaji
Argonne National Laboratory
balaji@anl.gov

Yutaka Ishikawa
RIKEN
yutaka.ishikawa@riken.jp

ABSTRACT

The two most common parallel execution models for many-core CPUs today are multiprocess (e.g., MPI) and multithread (e.g., OpenMP). The multiprocess model allows each process to own a private address space, although processes can explicitly allocate shared-memory regions. The multithreaded model shares all address space by default, although threads can explicitly move data to thread-private storage. In this paper, we present a third model called *process-in-process (PiP)*, where multiple processes are mapped into a single virtual address space. Thus, each process still owns its process-private storage (like the multiprocess model) but can directly access the private storage of other processes in the same virtual address space (like the multithread model).

The idea of address-space sharing between multiple processes itself is not new. What makes PiP unique, however, is that its design is completely in user space, making it a portable and practical approach for large supercomputing systems where porting existing OS-based techniques might be hard. The PiP library is compact and is designed for integrating with other runtime systems such as MPI and OpenMP as a portable low-level support for boosting communication performance in HPC applications. We showcase the uniqueness of the PiP environment through both a variety of parallel runtime optimizations and direct use in a data analysis application. We evaluate PiP on several platforms including two high-ranking supercomputers, and we measure and analyze the performance of PiP by using a variety of micro- and macrokernels, a proxy application, and a data analysis application.

KEYWORDS

Parallel Execution Model, Intranode Communication, MPI, In Situ

1 INTRODUCTION

Multicore and many-core architectures are promising solutions for modern high-performance computing (HPC) systems. Two parallel execution models are widely used in HPC applications on such massively parallel systems: the multiprocess model and the multithread model.

In the multiprocess model, the communication between executing processes on a node is limited by the operating system (OS), such that one OS process cannot directly access the memory regions owned by the other processes. Thus, the interprocess communication usually relies on the message-passing model (e.g., MPI) where additional data copy cannot be avoided. This model is known to be inefficient, however, especially when large numbers of processes on a node are communicating with each other.

Memory-mapping mechanisms have been broadly studied in HPC systems as an approach exploiting the capability of shared memory. POSIX (System V IPC or UNIX) shared memory (shmem) is the most portable and commonly used approach, which allows multiple processes to map the same physical memory region onto the virtual address space (VAS) of each process. However, this mechanism requires every process to participate in the memory region creation. Thus it cannot support any preallocated memory regions or statically allocated data. The other widely used technique is XPMEM [43], which is a Linux kernel module that allows a process to access the memory pages owned by the other processes. XPMEM allows a process to arbitrarily expose any memory region that it owns; other processes can then map the exposed memory region onto their local VAS.

Interprocess communication using these memory-mapping techniques can be achieved by using one of two approaches. In the first approach, a shared-memory buffer is allocated at initialization time, and any data movement between two processes is copied through this buffer. POSIX shmem, for instance, is used with this approach. In the second approach, the process that owns a memory region can expose that memory region to other processes by using system calls. Once the memory region is exposed, other processes can map the exposed memory region into their VAS and then access it directly. XPMEM, for instance, enables this approach. While the latter is more flexible in that it allows the runtime system to map user-managed buffers onto different processes on the node dynamically, a setup overhead is incurred for every memory mapping because it

[†] *Both authors contributed equally to this work.*

has to acquire the memory setup in the OS kernel through expensive system calls. Moreover, the memory-mapping approaches may also suffer from noticeable page fault (PF) overhead during data access because every process maintains a separate page table (PT) and thus a mapped memory page can create as many PT entries as the number of processes that access the page at the time of the first touch, resulting in frequent PFs.

Intercore communication is more convenient and efficient in the multithread model, where multiple threads share the same VAS. Thus, any thread can easily access any data allocated in the VAS. Moreover, accessing the shared-memory page by multiple threads does not trigger frequent PFs because the PT belonging to the VAS is also shared. However, race conditions between threads on shared variables must be carefully handled by application or runtime developers. In fact, the contention overhead between multiple threads still cannot be eliminated in most process-based runtime systems such as MPI [1, 11, 16, 18] because of the MPI semantics limitation.

A third parallel execution model combines the best of both worlds. In this model, each execution entity such as a process or thread, denoted *task* for short, has its own *variable set* to reduce the number of synchronizations, as in the multiprocess model, but shares the same VAS, as in the multithread model. Thus, accessing the data owned by another task becomes simple. Here *variable* is the variable declared in a program and is accessed through its name in a program. When multiple tasks share the same VAS, a privatized variable having the same name on each task generates a separate instance with a different address in the VAS. A privatized variable, however, can also be accessed by other tasks if the address of that variable is known. This model can be implemented by using either a thread-based or an OS-based approach. In the thread-based approach, the privatized variables can be declared to be located in thread-local storage (TLS) supported by a language system to reduce the synchronization overhead, based on support from special preprocessor or compiler systems [9, 32, 40, 46]. In the OS kernel-based approach, on the other hand, a special OS kernel or modification based on an existing kernel is proposed to support the sharing of VAS among processes [7, 35, 36]. Unfortunately, existing solutions rely heavily on either a special programming language system or OS kernel change, thus making their deployment on large HPC systems harder.

This paper presents a novel approach to support the third execution model, named *process-in-process (PiP)*. PiP *allows multiple parallel tasks to execute the same or different programs in a shared VAS environment while maintaining privatized variable sets*. The PiP task does not follow the definition of conventional process or thread. For instance, PiP tasks share the same VAS whereas processes have isolated VAS; two PiP tasks can execute arbitrary programs in parallel in the same VAS, whereas threads must be derived from the same program. Moreover, a process or PiP task always starts its execution from the `main()` function, but a thread starts its execution from an arbitrary function. PiP defines a special task called “root process” that owns the VAS and spawns multiple tasks executing in the same VAS as of the root; hence the model name “process-in-process.”

The fundamental idea of PiP is the unique combination of Position-Independent Executables (PIE) [30] and the `dlmopen()` Glibc function, which ensures that a program binary, its dependencies, and data can be loaded into the same VAS with different locations every time the program is executed. Unlike the existing thread- or OS-based approaches, PiP is a *more portable and practical technique that is implemented completely at the user level, thus being independent of language systems and OS kernels.*

We present the fundamental techniques of PiP and its design for supporting the multiprocess (e.g., MPI) and multithread (e.g. OpenMP) models in scientific programming. PiP aims to resolve the essential performance bottlenecks in the parallel runtime as a portable alternative to the traditional OS process or Pthread low-level support. We showcase the benefits of PiP in three essential HPC scenarios: (1) optimizing the shared-memory communication in MPI runtime via VAS sharing, (2) resolving the MPI multithreading overheads in the hybrid MPI+Threads model based on variable privatization, and (3) utilizing the data-sharing technique with in situ programming. We analyze the performance of PiP through a variety of micro- and macrobenchmarks, a particle transport proxy application, and a molecular dynamics application with in situ analysis. To demonstrate the portability of PiP, we performed the evaluations on four computing environments, including two HPC systems ranked in the top 10 of the Top500 as of November 2017.

2 BACKGROUND AND RELATED WORK

In this section, we give an overview of the available execution models with a focus on the support of memory-sharing and variable privatization. Table 1 summarizes the default support of the multiprocess and multithread models and their variations that enable the combined execution model. The memory-mapping techniques are also included since they enable limited sharing between processes. We define several terms below.

variable: A *variable* has an associated name. Stack (automatic) variables are out of scope here because they ought to be privatized and sharing stack variables is not recommended at all.

data: *Data* has no associated name and can be accessed only via a pointer variable (e.g., `malloc()`ed or `mmap()`ed region).

shared: If a variable is *shared*, then the variable referred to by its name in a program is the same variable having the same address regardless of the number of tasks.

privatized: If a variable is *privatized*, then it has the same number of variable instances as the number of tasks derived from the same program, and each instance belongs to a specific task. Race conditions can be avoided when accessing on a privatized variable.

accessible: The *accessible* variable or data can be accessed by other tasks by specifying its name or by load/store operations via a pointer variable, respectively. A *shared* variable is *accessible*.

2.1 Multiprocess with Memory Mapping

As listed in Table 1, two memory-mapping techniques have been widely used for data sharing in the multiprocess model. POSIX shmem includes System-V IPC and UNIX shared mmap. It is a general term to `mmap()` the memory pages owned by another process. This technique allows newly allocated memory segments (i.e., *data*) to be shared. However, a process cannot access statically allocated *variables* of the other processes. Moreover, the setup cost for the memory mapping is high.

Table 1: Summary of current techniques and PiP

		VAS	Variables	Data	Note
Multi-Proc	(default)	not shared	privatized, inaccessible	inaccessible	
	POSIX shmem			accessible	Only for newly allocated regions
	XPMEM		privatized, accessible	accessible	Linux kernel module
Third Model	SMARTMAP	**shared**	**privatized, accessible**	**accessible**	Kitten OS
	PVAS				Patched Linux kernel
	PiP				**OS and language system independent**
	MPC				Compilers needed
Multi-Thread	(default)	shared	shared	accessible	

XPMEM is the other well-known approach. It was initially developed by SGI and is now available in Linux as a kernel module. It allows processes to map arbitrarily memory regions (i.e., both *data* and *variables*) owned by other processes. These mappings involve costly system calls to the kernel module, however.

Inside the OS kernel, each process has a PT by which physical memory pages and virtual memory pages are associated. Although the memory-mapping approaches allow a process to access the memory region of other processes, it must create a new PT entry to access the region. The creation of new PT entries can result in expensive PFs during data access.

2.2 Multiprocess with VAS Sharing

Two OS-based VAS-sharing techniques have been studied in the multiprocess model. SMARTMAP is a built-in function of the Kitten lightweight OS kernel [7] that exploits the PT structure of the x86 architecture. PVAS provides functionality similar to that of SMARTMAP, but it is implemented as a patched Linux kernel [35, 36].

Different from the memory-mapping approaches, a memory region (both *data* and *variables*) can be seen by the other processes without any additional setup in a shared VAS. Moreover, sharing a VAS means sharing a PT. Thus, the aforementioned PF overhead does not occur in VAS sharing because every memory page creates its PT entry only once. The third benefit of VAS sharing is the reduced memory consumption for PTs. The number of PT entries in the memory-mapping approaches can grow as $O(N^2)$, where N is the number of processes in a node. When N is large on a many-core CPU, the size of PT can be problematic. In contrast, VAS sharing reduces the number of PT entries to $O(N)$.

Although the large page mechanisms in Linux (HugeTLB, Transparent Huge Pages, etc.) can relax the issues of the PF overhead and the total PT sizes, the utilization of large pages is limited in practice. The reason is that (1) more memory may be consumed, (2) memory utilization may be lowered further with NUMA-aware memory allocation, (3) physical memory fragmentation may hinder application performance because of page migration, and (4) users may become stressed at having to expend more effort in programming.

2.3 Multithread with Variable Privatization

In the multithread model, threads share the same VAS by definition. TOMPI [13], TMPI [39], AzequiaMPI [25], and MPC [32] studied ways to optimize MPI intranode communication based on the thread model. Most MPI implementations are based on the process model. When an MPI library is provided on top of the thread model where all *variables* are shared, implementation of process private data becomes the primary concern.

A compiler or preprocessor can convert statically allocated variables, which should be treated as private, that is, to locate variables in TLS. Zheng et al. [46] reported on three implementation techniques for localizing the shared data: (1) compiler transformation enables each thread may have different variable instances, (2) each thread has its own Global Offset Table (GOT) data and switch GOT entries, and (3) the compiler adds the `__thread` storage class specifier to those variables.

MPC is a thread-based language-processing system designed for hybrid MPI and OpenMP programming. It consists of custom compilers, linker, and runtime libraries; thus the maintenance is costly. To support MPI programs, the compiler converts statically allocated variables into TLS variables to make them private, and the MPC runtime creates threads running as MPI processes. Each of the threads may have child threads (e.g., OpenMP threads), and the child threads may contain user-defined TLS variables. A *hierarchical TLS* subsystem is designed to resolve the two kinds of TLS variables that have different access scopes; however, it also introduces additional overhead when accessing the TLS variables [9, 40].

Several issues commonly exist when integrating threads with variable privatization. First, multiple threads created by a process can load and execute only a single program. Thus, this approach cannot handle scenarios that require co-execution of multiple programs (e.g., an in situ analysis program is attached to another simulation program). Second, accessing TLS variables may incur extra overhead depending on the CPU architecture and TLS implementation. Third, the implementations (e.g., MPC) may rely on a special language-processing system; thus supporting another programming language is difficult. More important, a process-based program has to be recompiled in order to privatize variables. If a dependent library is available only in binary form, the library functions used by the program have to be thread-safe.

2.4 Others

The idea for the PiP implementation was inspired by a paper by Morozov and Lukic [27], in which they showed that a PIE program could be loaded into a process's VAS by calling the `dlopen()` Glibc function and could be executed by jumping into the `main()` function. In this way, an in situ program can be executed in the same VAS with a simulation program. Although this approach can load multiple programs in the same VAS, the variables are not privatized. Thus, it cannot support an arbitrary number of tasks running in the same VAS whether or not the loaded programs are the same.

The concept of user-level threads has been studied as the alternative to conventional OS-level threads [22, 29, 34, 41]. It allows the user program to create a large number of lightweight threads and manage the scheduling of these threads at user level. Unlike user-level threads, PiP focuses on the portability of the VAS sharing functionality; it makes no difference in the scheduling of execution units compared with the conventional OS process or thread. That is, PiP tasks are scheduled by the OS kernel and can be synchronized by using, for example, `pthread_mutex` similar to OS-level threads.

3 DESIGN AND IMPLEMENTATION

In this section, we introduce the definition and design of the proposed PiP VAS sharing model. We identify five design goals;

G1 Tasks have the same VAS by sharing the same PT.

G2 Each task has a privatized variable set so that the synchronization overhead found in the multithread execution model can be avoided.

G3 PiP requires no new kernel, no kernel patches, and no kernel module.

G4 PiP must be programming language independent and require no new language processing system.

G5 The upper-level runtime libraries (e.g., MPI) may have multiple tasks derived from one or more different executable files.

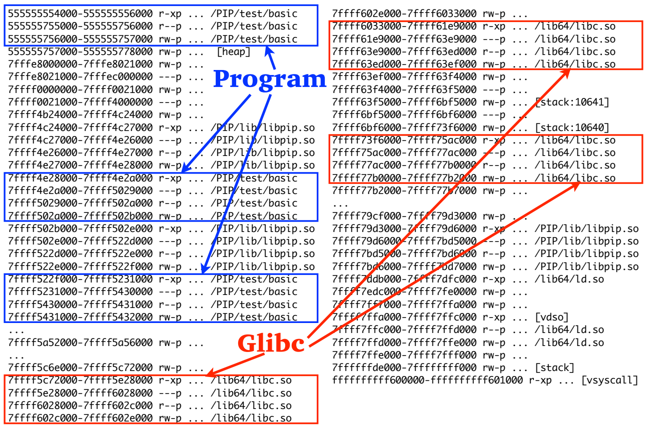
The PiP execution model is simple, having only two types of tasks.

(PiP) root process: A *PiP root process* can spawn PiP task(s) (see below). Spawned PiP tasks are mapped into the same VAS as the root process. The root programs can be regular binaries linked with the PiP library.

PiP task: A spawned task by a root PiP process is called a *PiP task*. PiP tasks having the same PiP root process share the same VAS created by the root process.

3.1 Variable Privatization

Here, *namespace* means the set of variable names and function names. Having a privatized variable set means that the variable named `var` in a PiP task can be accessed by its name in the PiP task. Different tasks derived from the same program have their own variables named `var`. However, each variable is located at a unique address. In this case, *variables are privatized*, so variable privatization means each task has an independent namespace.



```
555555554000-555555556000 r-xp ... /PIP/test/basic
555555755000-555555756000 r--p ... /PIP/test/basic
555555756000-555555757000 rw-p ... /PIP/test/basic
555555757000-555555778000 rw-p ...  [heap]
7fffe8000000-7fffe8021000 rw-p ...
7fffe8021000-7fffec000000 ---p ...
7ffff0000000-7ffff0021000 rw-p ...
7ffff0021000-7ffff4000000 ---p ...
7ffff4b24000-7ffff4c24000 rw-p ...
7ffff4c24000-7ffff4c27000 r-xp ... /PIP/lib/libpip.so
7ffff4c27000-7ffff4e26000 ---p ... /PIP/lib/libpip.so
7ffff4e26000-7ffff4e27000 r--p ... /PIP/lib/libpip.so
7ffff4e27000-7ffff4e28000 rw-p ... /PIP/lib/libpip.so
7ffff4e28000-7ffff4e2a000 r-xp ... /PIP/test/basic
7ffff4e2a000-7ffff5029000 ---p ... /PIP/test/basic
7ffff5029000-7ffff502a000 r--p ... /PIP/test/basic
7ffff502a000-7ffff502b000 rw-p ... /PIP/test/basic
7ffff502b000-7ffff502e000 r-xp ... /PIP/lib/libpip.so
7ffff502e000-7ffff522d000 ---p ... /PIP/lib/libpip.so
7ffff522d000-7ffff522e000 r--p ... /PIP/lib/libpip.so
7ffff522e000-7ffff522f000 rw-p ... /PIP/lib/libpip.so
7ffff522f000-7ffff5231000 r-xp ... /PIP/test/basic
7ffff5231000-7ffff5430000 ---p ... /PIP/test/basic
7ffff5430000-7ffff5431000 r--p ... /PIP/test/basic
7ffff5431000-7ffff5432000 rw-p ... /PIP/test/basic
...
7ffff5a52000-7ffff5a56000 rw-p ...
...
7ffff5c6e000-7ffff5c72000 rw-p ...
7ffff5c72000-7ffff5e28000 r-xp ... /lib64/libc.so
7ffff5e28000-7ffff6028000 ---p ... /lib64/libc.so
7ffff6028000-7ffff602c000 r--p ... /lib64/libc.so
7ffff602c000-7ffff602e000 rw-p ... /lib64/libc.so
7ffff602e000-7ffff6033000 rw-p ...
7ffff6033000-7ffff61e9000 r-xp ... /lib64/libc.so
7ffff61e9000-7ffff63e9000 ---p ... /lib64/libc.so
7ffff63e9000-7ffff63ed000 r--p ... /lib64/libc.so
7ffff63ed000-7ffff63ef000 rw-p ... /lib64/libc.so
7ffff63ef000-7ffff63f4000 rw-p ...
7ffff63f4000-7ffff63f5000 ---p ...
7ffff63f5000-7ffff6bf5000 rw-p ... [stack:10641]
7ffff6bf5000-7ffff6bf6000 ---p ...
7ffff6bf6000-7ffff73f6000 rw-p ... [stack:10640]
7ffff73f6000-7ffff75ac000 r-xp ... /lib64/libc.so
7ffff75ac000-7ffff77ac000 ---p ... /lib64/libc.so
7ffff77ac000-7ffff77b0000 r--p ... /lib64/libc.so
7ffff77b0000-7ffff77b2000 rw-p ... /lib64/libc.so
7ffff77b2000-7ffff77b7000 rw-p ...
...
7ffff79cf000-7ffff79d3000 rw-p ...
7ffff79d3000-7ffff79d6000 r-xp ... /PIP/lib/libpip.so
7ffff79d6000-7ffff7bd5000 ---p ... /PIP/lib/libpip.so
7ffff7bd5000-7ffff7bd6000 r--p ... /PIP/lib/libpip.so
7ffff7bd6000-7ffff7bd7000 rw-p ... /PIP/lib/libpip.so
7ffff7ddb000-7ffff7dfc000 r-xp ... /lib64/ld.so
7ffff7edc000-7ffff7fe0000 rw-p ...
7ffff7ff7000-7ffff7ffa000 rw-p ...
7ffff7ffa000-7ffff7ffc000 r-xp ... [vdso]
7ffff7ffc000-7ffff7ffd000 r--p ... /lib64/ld.so
7ffff7ffd000-7ffff7ffe000 rw-p ... /lib64/ld.so
7ffff7ffe000-7ffff7fff000 rw-p ...
7ffffffde000-7ffffffff000 rw-p ... [stack]
ffffffffff600000-ffffffffff601000 r-xp ... [vsyscall]
```

Figure 1: Example of `/proc/self/maps` (partly omitted)

Fortunately, Glibc supports the `dlmopen()` function, which can create a new *namespace*. By calling `dlmopen()` with the `LM_ID_NEWLM` flag and a filename of a PIE file, the specified PIE program is loaded with the privatized variable set. Thus, **G2** can be achieved. Figure 1 shows an example of `/proc/<PID>/maps` when a PiP root spawns two PiP tasks. In this example, a PiP root and two PiP tasks, three tasks in total, are derived from the same program (`/PIP/test/basic`).

```
7ffff73f6000-7ffff75ac000 r-xp ... /usr/lib64/libc-2.17.so /* TEXT */
7ffff75ac000-7ffff77ac000 ---p ... /usr/lib64/libc-2.17.so /* (gap) */
7ffff77ac000-7ffff77b0000 r--p ... /usr/lib64/libc-2.17.so /* GOT */
7ffff77b0000-7ffff77b2000 rw-p ... /usr/lib64/libc-2.17.so /* DATA & BSS */
```

Figure 2: A set of `libc-2.17.so` segments in Figure 1

Figure 2 shows one set of Glibc segments extracted from Figure 1. The first segment is the TEXT segment where program instructions are stored. The second segment from the top is a gap, and nobody can access this area. The third segment contains GOT, which is essentially an array whose elements are the addresses of external references. The fourth segment at the bottom is DATA & BSS, which contains statically allocated variables defined in Glibc. As shown in Figure 1, there are three tasks and three sets of dynamically shared objects (DSOs). It indicates that each PiP task has its own set of privatized variables for both the program and its dependent libraries.

We note that the current implementation of `dlmopen()` can support only up to 16 namespaces—too small even with the conventional multicore CPUs. This is an implementation issue. We therefore patched Glibc so that the current PiP can create up to 300 PiP tasks. Still, since Glibc is a user-level library, this feature does not ruin the PiP portability. Although we faced some other trivial Glibc implementation issues, we succeeded in working around them in the PiP library.

3.2 Loading Programs into the Same VAS

As described in the preceding section, PiP tasks are derived from PIE programs so that they are loaded into the same VAS with different locations. By using PIE, **G1** and **G5** can be achieved.

Today computer security is a big concern, and address space layout randomization (ASLR) is one of the useful techniques. In ASLR, address mapping of a process is randomized, and therefore programs are compiled and linked as PIE. Furthermore, on many recent operating systems such as Android, iOS, and Mac OSX [42], non-PIE application programs are not accepted for security reasons. Major Linux distributions are following the same road. Thus, PIE has already started to become the de facto binary format.

3.3 Running with an OS Kernel Thread

We next run the loaded program with an OS kernel thread. If this process can be done at the user level and independent from any language-processing system, **G3** and **G4** can be achieved. We design two execution modes in PiP.

Process Mode: In this mode, the OS kernel thread and its stack can be created by calling the Linux `clone()` system call with `CLONE_VM`[1] and without the `CLONE_THREAD`[2] flag setting. With the `CLONE_VM` flag, the `clone()` system call creates an OS kernel thread to run inside the VAS of the caller process. We reset the `CLONE_FS`,[3]

[1] If set, the calling process and the child process run in the same VAS.

[2] If set, a child task is created as a thread.

[3] If set, tasks share the same file system info.

```c
/* --- Core Function --- */
/* 1. Initialize and return my task id, number of tasks, and the address
of an exported region on the root if specified. */
int pip_init (int *id, int *npips, void **root_exp, int opts);

/* 2. Finalize. */
int pip_fin (void);

/* 3. Spawn a PiP task with specified id; if PIP_PIPID_ANY is set,
then the library assigns the id and return the id value. */
int pip_spawn (char *filename, char **argv, char **envv, int coreno,
      int *id, pip_spawnhook_t before, pip_spawnhook_t after,
      void *hookarg);

/* --- Helper Function --- */
/* 4. Get the address of a global variable named nm on task id. */
int pip_get_addr (int id, const char *nm, void **ptr);

/* 5. Initialize the barrier synchronization structure for n participants. */
void pip_barrier_init (pip_barrier_t *barp, int n);

/* 6. Wait on a barrier in a busy-wait way. */
void pip_barrier_wait (pip_barrier_t *barp);
```

Figure 3: PiP API (partly omitted)

CLONE_FILES,[4] and CLONE_SIGHAND[5] flags, so that PiP tasks behave like normal processes regarding PID, termination, file descriptors, and signal handling.

Thread Mode: In this mode, the pthread_create() function is used instead of clone(). PiP tasks behave like threads regarding TID, termination, file descriptors, and signal handling. Again, PiP provides the variable privatization even in this thread execution mode.

The clone() system call is Linux specific, and using this system call means that PiP depends on Linux. If a system does not support the clone() system call, then the pthread_create() function can be used instead. This POSIX function is more widely supported than the clone() system call, and this makes PiP independent from the OS kernel. Thus, **G3** can be achieved.

As described so far, PiP is implemented as a language-independent library, and **G4** is satisfied.

By combining all the above techniques, all design goals are achieved.

3.4 API

Figures 3 summarizes the functions defined in the PiP library. The three core functions are the minimal set of APIs that enables PiP. The pip_spawn() function is called from the PiP root process, which is a normal process, to create a PiP task located in the same VAS as the root. The pip_init() and pip_fin() functions are to initialize and finalize the PiP library. The PiP root process must call them, but the PiP tasks call them only if they use the other PiP functions. The pip_get_addr() function returns the address of a global variable having the specified name owned by the specified PiP task or root. Pthread's barrier and mutex functions can be used to synchronize PiP tasks and root. The pip_barrier_init|wait() functions provide a simple busy-wait based synchronization.

3.5 Portability

PiP depends on the dlmopen() function, PIE, and the clone() system call or pthread_create() function. The most widely used Linux in high-performance computing supports all of them.

Tables 2 and 3 list the machine environments used in this paper. As shown in these tables, PiP can run on various OS kernels and CPU architectures. One may argue that all operating systems of the machines listed in Table 3 are Linux family. According to the Top500 as of November 2017, however, Linux family OSes run on all Top500 machines [38].

Another goal of PiP is language system independence. A typical language system consists of a compiler, linker, runtime libraries, and debugger; it also may include profiling systems. None of these is easy to develop and maintain. PiP is a small library of only about 3,000 lines and thus is easy to maintain. Moreover, PiP is language independent. Indeed, C, C++, and Fortran programs run with PiP. The process execution mode of PiP can run with a thread runtime such as OpenMP. Depending on the OpenMP implementation, OpenMP programs may also run with the thread execution mode of PiP.

3.6 Robustness and Debuggability

Unlike the multiprocess model, the VAS sharing model of PiP removes the protection boundaries of processes in order to improve performance of intercore communication. PiP assumes that all PiP tasks spawned by the same root always behave properly. The memory corruption on a PiP task may cause all tasks in the same VAS to crash, similar to the risk in a multithread program. It is a tradeoff between robustness and performance. The user should share the VAS only when needed.

The complexity of debugging a PiP program is similar to that of a multithread program. A common mistake in multithread programming is that the user does not properly synchronize simultaneous access from multiple threads to a shared variable or data. In PiP, however, the user must explicitly declare every shared variable or data by exchanging the address; thus, the risk of introducing such bugs in a PiP program is small in practice.

4 PIP SHOWCASES

Although users may use PiP directly, we expect that the most common case is to use PiP as an underlying software layer of parallel runtimes; hence, users may not recognize using PiP. The PiP model shows a process aspect (i.e., *variable privatization*) and a thread aspect (i.e., *VAS sharing*). The programming models based on multiprocess or multithread can be implemented by adopting either of the aspects.

In this section, we demonstrate how PiP can benefit the widely used MPI and OpenMP programming models. These examples, however, are not intended for PiP to take over the MPI or OpenMP. PiP merely provides another low-level support of these models for users who are not satisfied with the performance of conventional process-based or thread-based implementation. We also show the benefit of PiP in an emerging in situ analysis application.

4.1 Using PiP in MPI Runtime

MPI is the most widely used parallel programming model on distributed-memory systems. We extended the process launching in the MPICH implementation of MPI (v3.3a2) to use a PiP task as an MPI process. We then chose two internal processing of MPICH to showcase the *process aspect of PiP with efficient memory sharing*.

[4] If set, tasks share the same file descriptor table.

[5] If set, tasks share the same signal handler table.

The presented techniques are generally valid also for other MPI implementations.

Process Launching: MPICH utilizes a Hydra process management system to start parallel jobs [4]. It spawns a helper agent process, called *pmi_proxy*, at each node on the system to handle any process management functionality, such as MPI process spawning and cleanup. In our PiP-supported MPI runtime, the proxy process is the PiP root at each node, and it spawns all MPI processes on that node as PiP tasks by calling the `pip_spawn()` function. Since Hydra is designed to create normal processes, PiP runs only with the process mode in this implementation. The number of lines to change Hydra to spawn PiP tasks is only about 200.

In this prototype implementation, the proxy process (PiP root) and MPI processes (PiP tasks) share the same VAS. Whether or not a user is malicious, an MPI process may destroy the data of the *pmi_proxy* process, resulting in aborting the MPI execution on this node. Fortunately, the hierarchical process structure of Hydra can successfully report this error to the user. If protection of the process manager process is required in an MPI implementation, then the process manager creates another process responsible only for spawning PiP tasks as MPI processes.

Optimizing Intranode Communication: Traditional MPI implementations typically utilize a portable POSIX shmem–based mechanism to transfer messages between processes on the same node. Such approaches, however, do not allow the MPI runtime to directly expose and share user-managed buffers such as those allocated in user applications and passed into MPI calls. Consequently, the MPI runtime has to create additional internal buffers shared between the communicating processes and utilize expensive 2-copy processing to transfer messages [8]. Kernel-assisted memory-mapping techniques such as XPMEM support dynamic exposing and sharing of user buffers, but the expensive setup cost of sharing a buffer cannot be avoided.

PiP's VAS sharing feature can inherently support dynamic sharing of user buffers without any kernel assistance. The *eager* protocol is commonly used for small message transfer where the sender process copies data into an available chunk of the preallocated shared queue on the receiver process. We still keep it in the PiP-based version for better overlap of communication. In medium and large message transfer, the traditional *rendezvous* protocol involves heavy shared buffer creation or memory exposing (i.e., in XPMEM) at the handshake step. Although this overhead can be diluted by caching the exposed memory regions, it can be entirely eliminated if we use PiP. We need to exchange only the address of the user buffer through the handshake; the receiver then moves data within only a *1-copy*. Because the send buffer cannot be used by the user until the copy is completed, the receiver process has to notify the completion after the memory copy is done.

Enhancing Shared-Memory Allocation: MPI-3 introduced the `MPI_Win_allocate_shared()` interface that allocates a shared-memory region for multiple processes on the same node. It enables an alternative to the traditional message-based MPI communication in the shared-memory environment. That is, the user can explicitly locate data in the shared-memory region, and the processes access such data by using direct load/store or MPI RMA operations. Obviously, this model does not require extra data copy that has to be involved in the message-based model (see the aforementioned eager protocol), thus improving performance [21].

In mainstream MPI implementations, the shared-memory region is usually allocated by using POSIX shmem or special memory mapping tools such XPMEM [19, 21]. The data-accessing overhead is identical under both approaches. In our prototype of PiP-aware MPI, we reuse the design for scalable window allocation introduced in [21] but with a simplified allocation step. A root process allocates a memory region (e.g., by calling `malloc()`), and that region is naturally accessible by others.

As a more significant benefit, the PiP-based version also reduces the PF overhead of performance-critical data access because all PiP tasks share the same PT. We demonstrate the performance difference in Section 7.2.

Other Opportunities: The PiP-aware MPICH is only a prototype implementation. PiP's VAS sharing model can be beneficial also for many other internal aspects of the MPI runtime such as the collective communication [24] and the memory-saving optimizations for scaling out the number of MPI processes [20, 31]. In this paper, we focus on the core concept of the PiP model and thus leave deep exploitation of MPI optimizations for future work.

4.2 Using PiP in Hybrid MPI+Threads

The hybrid MPI+Threads model has become the norm for programming on multicore and many-core platforms. Although the VAS sharing capability of threads allows computation to be efficiently executed in parallel on multiple cores of a node, it often suffers from heavy overhead or degraded parallelism in communication when scaling across nodes with an internode communication runtime such as MPI. We describe the critical challenges in the hybrid MPI+Threads model and present a solution based on the *thread aspect of PiP that combines VAS sharing and variable privatization features*.

Limitations in Hybrid MPI+Threads: The MPI standard provides four levels of thread safety. Support of `MPI_THREAD_MULTIPLE` (denoted by multithreading) safety[6] is known to be expensive [5]. The reason is that the message-passing semantics in MPI are designed for processes; that is, MPI ranks are assigned at the process level, and the message matching is based on a set of {rank, tag, communicator}. Consequently, the implementations of MPI usually isolate internal resources (e.g., the message-matching queue) per process. When multiple threads coexist on a process, they have to *exclusively access the resources with expensive lock protection*. Amer et al. [1] and Dang et al. [11] have optimized such critical issues from the locking aspect, but non-negligible overhead still exists compared with the single-threaded case. Although the contention may be reduced by partitioning the range of communicator, sender rank, and tag, such an approach cannot address the complication of wildcard receive.[7] As a result, most MPI implementations still utilize coarse-grained critical sections for simplicity and support of complete semantics.

In addition to substantial contention overhead, the performance of multithreaded MPI communication is *limited by the low utilization*

[6] `MPI_THREAD_MULTIPLE`: Multiple threads can make MPI calls simultaneously.

[7] Wildcard receive: A receive with `MPI_ANY_SOURCE` can match with a message from arbitrary sender rank in the communicator; similar semantics apply to `MPI_ANY_TAG`.

of network resources. For HPC applications that usually issue small to medium messages, concurrent utilization of multiple network hardware contexts (e.g., on InfiniBand and Intel Omni-Path) is especially important for boosting network throughput. However, multiple threads on a process have to post messages to the same MPI stack through the critical sections. This process reduces the number of cores that can concurrently post messages to the network, thus the message throughput of multiple threads can be significantly lower than that of multiple processes.

Some restricted programming models can limit the threads concurrency at access to the communication runtime (e.g., `MPI_THREAD_FUNNELED`, denoted by funneled[8]), thus eliminating the contention overhead; however, such models still cannot address the network utilization issue. Recent work has focused on resource isolation for threads in the communication runtime. For instance, the MPI thread endpoint concept is currently under investigation by the MPI Forum [14]. A PMPI library-based extension of MPI has been implemented by translating threads to proxy processes; however, additional data offload overhead cannot be avoided [37].

Applying PiP to the Multithreading Model: As an alternative solution, we can use PiP tasks as the underlying support for the multithreading runtime, similar to Pthreads. This is based on the notion that PiP allows arbitrary data sharing between tasks similar to threads. It also enables variable privatization similar to processes. Thus, the performance issues of multithreaded MPI disappear.

Here we showcase the widely used OpenMP fork-join model. For simplicity we did not change the OpenMP runtime, but we applied all the necessary changes to our test proxy application for a quick showcase of PiP. Figure 4 compares a simple OpenMP program with its translated execution code based on PiP tasks. Five modifications must be applied: (1) the master task is decided by checking the PiP IDs (line 10), (2) every task queries the address of the shared instance explicitly (e.g., the instance is allocated on the master task. `pip_get_addr()` is used to query the address of a global variable, as shown in line 8), (3) the `barrier` pragma and other implicit barriers are replaced with `pip_barrier_wait()` (lines 12 and 20), (4) the `for` pragma is replaced with explicit workload distribution (lines 14–17) and (5) the "task teams" are managed by creating additional MPI subcommunicator at the application initiation time (omitted in Figure 4). We note that these changes can be implemented more efficiently inside the PiP-aware OpenMP implementation. We leave this task for future work.

We also changed the application code to replace the original tag-based message matching for multiple threads with distinct ranks because every PiP task can own a different MPI rank. No change is required in the MPI library except the process launching mentioned in Section 4.1. Although the PiP-based approach requires several changes, it completely resolves the performance issues of multithreaded MPI. We demonstrate the performance improvement in Sections 7.3.

4.3 Data Analysis Application

Apart from the parallel programming aspects, PiP can directly benefit applications in several ways. For instance, its *VAS sharing ability* can be utilized for in situ programming. In most in situ frameworks, applications use expensive data copies to exchange data with concurrently running analysis routines. Several approaches have been studied to implement these $M{\times}N$ data exchanges [2, 12, 15, 45].

```
 1 int a[N];
 2 int main() {
 3   int i;
 4
 5   #pragma omp parallel
 6   {
 7
 8
 9     #pragma omp master
10     { /* compute master */ }
11
12     #pragma omp barrier
13
14     #pragma omp for
15     for (i=0; i<N; i++)
16     { /* compute a[i] */ }
17
18
19
20     /* implicit barrier */
21   }
22
23 }
```

(a) OpenMP program

```
 1 int a[N];
 2 int main() {
 3   int i, myid, npips, masterid = 0;
 4   pip_init(&myid, &npips, NULL, 0);
 5   /* --- omp parallel --- */
 6   {
 7     int *ap;
 8     pip_get_addr(masterid, "a", &ap);
 9     /* --- omp master --- */
10     if (myid == masterid)
11     { /* compute master */ }
12     pip_barrier_wait(...);
13
14     /* --- omp for --- */
15     int s = N/npips * myid;
16     int e = s + N/p;
17     for (i=s; i<e; i++)
18     { /* compute a[i] */ }
19
20     pip_barrier_wait(...);
21   }
22   pip_fin();
23 }
```

(b) PiP-based program

Figure 4: Example of PiP-based OpenMP program

By porting the in situ framework with PiP, a PiP task is created and kept aside as the in situ process and shares the same VAS with the other application PiP tasks (e.g., the science simulation). Thus, the in situ process can directly access the application's data. One common issue with these frameworks is the assurance of data consistency between the application and the in situ task. One approach is to use locking and signaling mechanisms to indicate when data can be accessed and overwritten. Another option is to give the consumers their own copies of the data. The former can reduce copies and memory consumption, but the latter can allow for higher concurrency because the simulation can run while the analysis is processing the data. For the experiments shown in Section 7.4, we chose the latter approach, but the implementation is flexible enough to allow for either method.

4.4 Summary of PiP Utilization

As showcased in MPI runtime and the data analysis application, the process aspect of PiP becomes beneficial when memory sharing is required (i.e., *variable privatization with VAS sharing*). It outperforms the state-of-the-art memory-mapping techniques because of its low overhead, ease of use, and high portability.

The thread aspect of PiP, on the other hand, enables a unique *VAS sharing environment with inherent variable privatization*. Here we consider two situations: (1) a program does not contain any statically allocated variables or contains some of them but rarely accesses, and (2) a program follows the multiprocess model where static variables are intensively used. PiP has no advantage over the multithread model in the former case; however, the latter can be dramatically optimized by using PiP with only small code reconstruction—something no other existing technique can achieve, as demonstrated in the hybrid MPI+Threads example.

The majority of hybrid MPI+Threads-based applications still follow the MPI funneled safety to work around the performance issues in multithreading safety. However, such an approach can no longer satisfy the network throughput, especially on many-core architectures where performance highly relies on the concurrence

[8] `MPI_THREAD_FUNNELED`: Only the master thread on a process can make MPI calls.

of a large number of low-frequency cores polling the network. Thus, an increasing number of applications are being built by using the multithreading mode [26], and PiP will be the ideal tool to maximize communication performance.

5 EXPERIMENTAL SETTING

We used four experimental platforms to cover several OS kernels and CPU architectures in our evaluation, as listed in Tables 2 and 3. The Linux kernel on the K computer is old, and we gave up trying to install the patched Glibc. The CPU of the K computer supports only eight cores; thus, PiP without the patched Glibc can still utilize all CPU cores.

McKernel is a multikernel that runs Linux with a lightweight kernel side by side on compute nodes [17]. In the experiments with McKernel on Wallaby, McKernel was configured to run on 14 cores out of 16, and the Linux kernel ran on the remaining 2 cores. Since the current McKernel is unable to handle the `clone()` flag combination described in Section 3.3, the PiP programs ran in the thread execution mode.

We report the results of each experiment by averaging 10 executions, unless otherwise stated.

Table 2: Experimental platform hardware information

Name	CPU	# Cores	Clock	Memory	Network
Wallaby	Xeon E5-2650 v2	8×2(×2)	2.6GHz	64 GiB	ConnectX-3
OFP†	Xeon Phi 7250	68(×4)	1.4GHz	96(+16) GiB	Omni-Path
K [44]	SPARC64 VIIIfx	8	2.0GHz	16 GiB	Tofu

Table 3: Experimental platform software information

Name	OS	Glibc	PiP Exec. Mode(s)
Wallaby	Linux (CentOS 7.3)	w/ patch	process and thread
Wallaby	McKernel+CentOS 7.3	w/ patch	thread only
OFP†	Linux (CentOS 7.2)	w/ patch	process and thread
K	XTCOS	w/o patch	process and thread

† Oakforest-PACS (OFP) http://jcahpc.jp/eng/ofp_intro.html. The flat mode was used in the showcase evaluations in Section 7.1 and 7.3 without using MCDRAM (16 GiB). The other evaluations were done with the cache quadrant mode.

6 PIP PERFORMANCE ANALYSIS

We evaluate the characteristics of PiP by using a set of in-house microbenchmarks.

6.1 Setup Overhead

In our first microbenchmark, the root task created and initialized a 2 GiB shared array with integer elements, and then a child task summed members of the array, assuming that the root task sent integer data to the child task via the allocated region. We implemented the XPMEM based and POSIX shmem–based versions. Table 4 shows the times spent in the XPMEM and POSIX shmem functions. PiP also provides the XPMEM APIs so that the XPMEM version can be easily linked to PiP. Most of the XPMEM functions provided by PiP do almost nothing, and the overhead of each function is only 40–80 clock cycles.

Table 4: Overhead of XPMEM and POSIX shmem functions on Wallaby/Linux

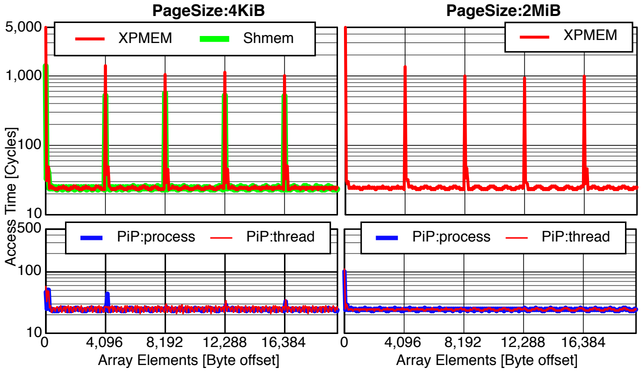
XPMEM	Cycles
`xpmem_make()`	1,585
`xpmem_get()`	15,294
`xpmem_attach()`	2,414
`xpmem_detach()`	19,183
`xpmem_release()`	693

	POSIX Shmem	Cycles
Sender	`shm_open()`	22,294
	`ftruncate()`	4,080
	`mmap()`	5,553
	`close()`	6,017
Receiver	`shm_open()`	13,522
	`mmap()`	16,232
	`close()`	16,746

Note: Measured only once.

6.2 Page Fault Overhead

Figure 5 shows the time series of each access using the same microbenchmark program used in the preceding subsection. Element access was stridden with 64 bytes so that each cache block was accessed only once, to eliminate the cache block effect. The left-hand graphs show spikes with 4 KiB page size. The spike heights of XPMEM are higher than the ones of POSIX shmem; however, the PiP process mode and PiP thread mode show the lowest spike heights. With XPMEM and POSIX, a PF happened every time a new memory page is accessed. The spikes in PiP are the time spent for the translation lookaside buffer (TLB) misses. In PiP, the whole array was touched at the time of initialization by the root task, and all required PT entries were created then.



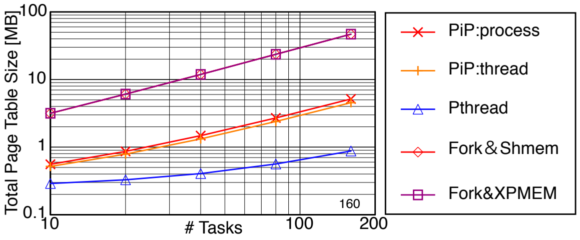
Note: Measured only once. The upper graphs show the time series using POSIX shmem and XPMEM, and the lower graphs show the time series using PiP. Both graphs on the left-hand side show spikes with 4 KiB page, and the graphs on the right-hand side show spikes with 2 MiB HugeTLB.

Figure 5: Time series of array access with 64-byte stride on Wallaby/Linux

The right graphs show the same benchmark but using HugeTLB. POSIX shmem cannot handle the HugeTLB on this Linux kernel. XPMEM does show huge spikes again on the every 4 KiB page boundary. We consulted the XPMEM device driver source code (version 2.6.4) and found that the XPMEM driver can create only 4 KiB PT entries, regardless of the page size of the target region. In PiP, no TLB-miss spikes can be seen this time because of using 2 MiB pages.

6.3 Total Page Table Size

This subsection focuses on the memory consumption of PTs. We compared the PiP process and thread models with Pthread, process fork with POSIX-shmem (the `mmap()`ed region of the parent process was inherited by the child process for simplicity), and process fork with XPMEM.



Note: The results of Fork&Shmem and Fork&XPMEM are overlapped.

Figure 6: Total page table size running on Wallaby/Linux

Figure 6 shows the total size of PTs in a node (y-axis) with varying number of tasks (x-axis). In this microbenchmark, a 128 MiB memory region was shared or made accessible among all tasks. Each task accessed the whole memory region so that all PT entries for the memory region were created. Then we consulted the `/proc/meminfo` file to get the memory size for all PTs in that node. In Fork&Shmem and Fork&XPMEM, each process has its own PT with separate PT entries for this memory region to share. In contrast, PiP and Pthread share the same PT. As shown in this figure, the former cases consume much more memory just for PTs. Table 5 summarizes the number of PT entries required for each technique.

Table 5: Total number of page table entries

	Total Number of Page Table Entries
Pthread	$M + D + \sum S_i$
PiP	$M + \sum D_i + \sum S_i$
Process + POSIX shmem	$(M \times N) + \sum D_i + \sum S_i$
Process + XPMEM	$(M \times N) + \sum D_i + \sum S_i$

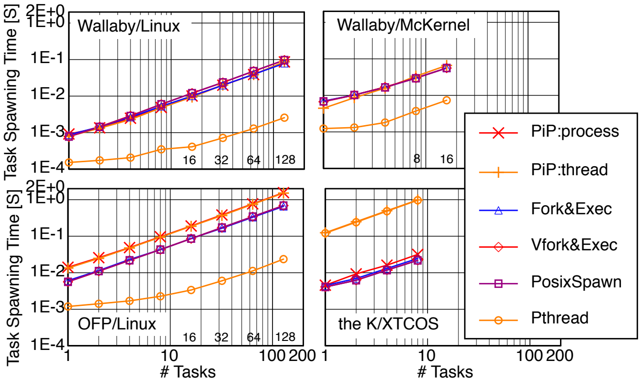
M is the number of PT entries for the shared-memory region(s).
S_i is the number of PT entries for the stack segment of task i.
D_i is the number of PT entries to map shared objects belonging to task i.
N is the number of tasks (processes or threads).

6.4 Spawning Time

Our next microbenchmark measured the time for spawning child tasks. In PiP, all memory mappings were done at the program loading time, and its cost is hidden from the time for accessing it. The purpose of this microbenchmark is to measure this “hidden” cost. Figure 7 compares the time to spawn null tasks by using PiP, Pthread, `fork()&exec()`, `vfork()&exec()`, and `posix_spawn()`. As shown in this figure, the PiP spawning times are mostly the same as those with creating processes, except the OFP case. In most cases, although the program loading is known to be costly, it does not happen frequently, so this overhead is acceptable.

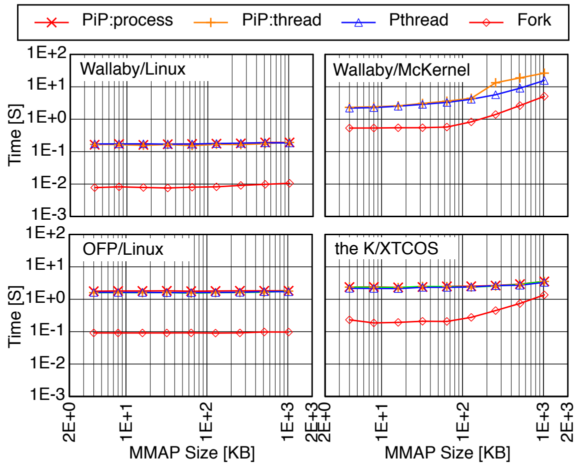
6.5 Performance of `mmap()/munmap()`

In PiP and Pthread, the memory management structures that point to the same PT in the Linux kernel are also shared. These structures must be locked in order to avoid inconsistent states by the race conditions being accessed simultaneously. This situation never happens between processes; and the lock overhead might be a weak point of PiP when the number of memory segments is significant, as shown in Figure 1.



Note: On Wallaby/Linux and Wallaby/McKernel, the results of all approahes are overlapped except Pthread; on OFP/Linux, the results of Fork&Exec, Vfork&Exec, and PosixSpawn are overlapped, and the results of PiP models are overlapped; on K/XTCOS, the results of PiP:thread and Pthread are overlapped.

Figure 7: Task spawning time on four platforms



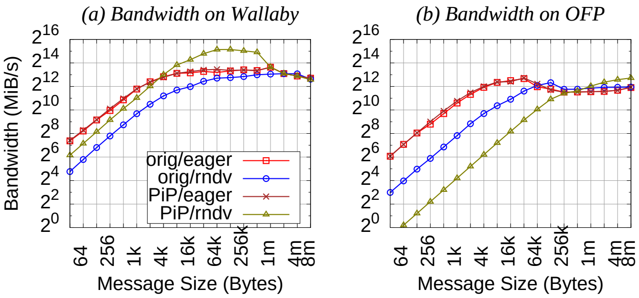
Note: The results of PiP models and Pthread are overlapped in the four graphs.

Figure 8: Performance of `mmap()/munmap()` with ten tasks on four platforms

In this benchmark, memory pages were `mmap()`ed and then `munmap()`ed, repeating 10,000 times. We created ten tasks and measured times until all tasks finished. Each PiP task, Pthread, or forked process was bound to a dedicated CPU core so that it could run without having any context switching (except in the K computer). As shown in Figure 8, PiP performance is similar to that of Pthread, whereas forked processes run much faster.

7 SHOWCASE EVALUATION

We used MPICH v3.3a2 and the PiP-aware version to compare the optimizations in MPI runtime. For the evaluation of hybrid MPI+Threads in Section 7.3, we used the Intel 2017.4.196 package and MPICH v3.3a3 that enables a two-level priority lock optimization for low contention overhead [1]. Every PiP task or thread is bound onto a different physical core in the *compact* shape.



Note: The results of orig/eager and PiP/eager are overlapped.

Figure 9: MPI intranode bandwidth with two processes on Wallaby/Linux and OFP

7.1 MPI Intranode Communication

We first measured the optimized MPI intranode communication, as described in Section 4.1. We disabled the protocol-switching thresholds in order to demonstrate the limit of each approach on the test platforms. We compared the PiP *1-copy* approach for the rendezvous protocol with the original POSIX shmem–based implementation with *2-copy*, denoted by PiP/rndv and orig/rndv, respectively. As a reference, we also included the unchanged eager protocol of each MPI implementation, which performs *2-copy* with less synchronization, denoted as PiP/eager and orig/eager, respectively. We measured the Intel MPI Benchmarks (IMB) 2017 `SendRecv` test with the on-cache mode on a single node.

Figure 9(a) shows the bandwidth on Wallaby/Linux. As expected, the unchanged eager protocol benefits small messages because of less synchronization. PiP/rndv overcomes the eager approaches when the message size increases to 4 KiB because of less data copy. It achieves a peak bandwidth of 35.4 GiB per second at a 128 KiB message size, whereas the eager copy reports up to 12.8 GiB/s (2.8x reduced) and orig/rndv reports only 8.7 GiB/s (4x reduced). When the message increases to 1 MiB, however, the bandwidth starts degrading because of heavy last-level cache and TLB misses and eventually reduces to the same level as the other approaches. Figure 9(b) shows the bandwidth on a single node of OFP. The eager approaches outperform the others for messages smaller than 256 KiB, indicating a larger eager threshold. For large messages, PiP/rndv contributes up to 6.7 GiB/s bandwidth, but orig/rndv achieves a peak of only 3.8 GiB/s.

7.2 MPI Shared-Memory Enhancement

We then evaluated a five-point two-dimensional stencil kernel that uses the `MPI_Win_allocate_shared()`. We compared the original process-based MPICH and the PiP-aware MPICH with enhanced shared-memory allocation (see Section 4.1) by focusing on the number of PFs. The stencil kernel is described in [21] and available online.[9]

Figure 10 shows the number of PFs when the stencil programs ran with a size of $8,192 \times 8,192$ in strong scaling, 1,000 iterations. These results were sampled just before and after the stencil loop. The number of PFs with original MPI is two or more orders of magnitude higher than that of the PiP-aware version. This implies that the frequent intranode data access to newly allocated shared regions may suffer from the PF overhead.

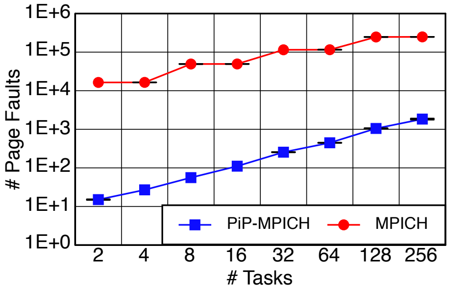
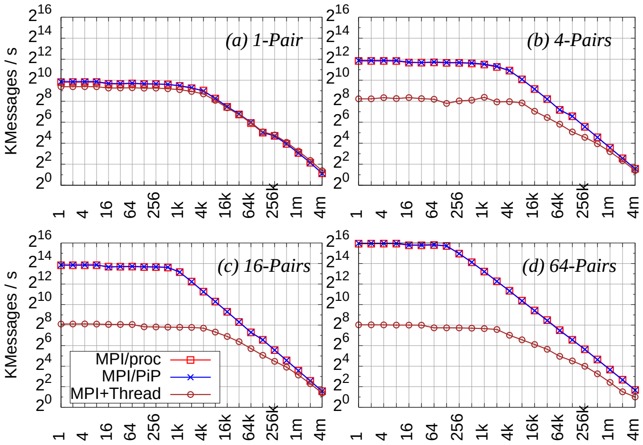


Figure 10: Number of page faults in 5P stencil program using `MPI_Win_allocate_shared()` running on a single OFP node



Note: The results of MPI/proc and MPI/PiP are overlapped in the four graphs.

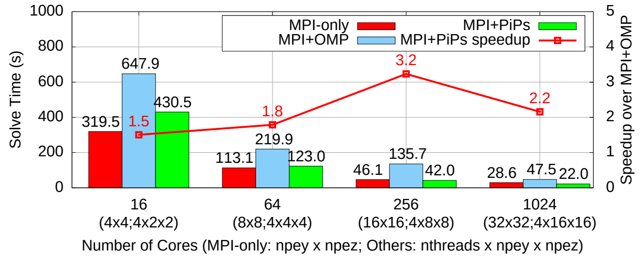
Figure 11: Multipair message rate between two OFP nodes

7.3 Communication in Hybrid MPI+Threads

The third set of experiments focused on the multithreaded MPI communication in the hybrid MPI+Threads programs (see Section 4.2). We measured a message rate microbenchmark and a proxy application.

Multithreaded Message Rate: We first employed the OSU microbenchmark (version 5.1) `osu_mbw_mr` test that measures the message rate with multiple pairs of processes. We compared the PiP-based version (MPI/PiP) with the thread-based version (MPI+Thread) on two nodes of OFP. We also measured the MPI-only model with processes (MPI/proc) as the reference showing the best network utilization without any contention overhead. In MPI/PiP, we simply ran with the PiP-aware MPICH. In MPI+Thread, we changed the benchmark to launch only one process on each node and create multiple threads on each process. The communication happened between each pair of threads located on different nodes. As shown in Figure 11, MPI/PiP delivers improved message rate with increasing numbers of pair connections, similar to the results of MPI/proc. However, MPI+Thread shows degradation. This is because the thread approach fails to concurrently utilize the network by multiple cores but introduces additional contention overhead in the multithreaded MPI.

[9] Example program `stencil_mpi_shmem.c` in Advanced MPI Programming - Tutorial at SC14. http://www.mcs.anl.gov/~thakur/sc14-mpi-tutorial



Note: Uses 64 cores per node, and every PiP task or thread is bound onto different physical core in *compact* shape; report average of 3 runs.

Figure 12: Strong scaling of hybrid MPI+OpenMP SNAP on OFP.

SNAP Particle Transport Proxy Application: SNAP[10] is a proxy application that models the performance of modern discrete ordinates neutral particle transport application. SNAP mimics the computational workload, memory requirements, and communication patterns of the PARTISN neutron transport code [3] with parallelism at the level of the spatial, the angular, and the energy dimensions. The implementation uses the hybrid MPI+OpenMP model where the spatial domain is partitioned across MPI ranks and traversed with sweeps along the angular domain. The energy domain is parallelized by threads. In the OpenMP parallel regions, the threads perform both computation and MPI point-to-point communication following the *multithreading* safety. We updated the SNAP code to be PiP-aware following the description in Section 4.2. We compared the PiP-aware version (MPI+PiPs) with the conventional thread-based approach (MPI+OMP), and measured the MPI-only model as a reference. We note that the domain decomposition in the MPI-only model and the hybrid model are different in the SNAP code, and the comparison of these models is beyond the scope of this paper.

We generated the input files for strong scaling based on the in-package `3d_mms_t2.inp` regression test input, which uses the method of manufactured solutions setting. We increased the spatial domain size, the angular per octant, and the energy group to (nx=128,ny=32,nz=32), $nang$=32, ng=16, respectively. We fixed the problem size and varied the number of MPI ranks along the y and z dimensions (denoted $npey$ and $npez$) for strong scaling and fixed the number of threads per process ($nthreads$=4) in the hybrid approaches.

Figure 12 reports the solve time of 10 time steps. MPI+PiPs always outperforms MPI+OMP. Especially when scaling to a large number of cores, the communication becomes dominant, and thus PiP delivers more benefit, contributing close to 3.2x performance speedup on 256 cores (4 nodes). The reduced speedup on 1,024 cores (i.e., 2.2x) indicates that the contention overhead of multithreaded MPI is optimized by the priority locks. Nevertheless, the network utilization issue still exists and relies on the support from PiP. We note that the MPI-only model shows faster solve time than that of the hybrid approaches on a single node (up to 64 cores). The reason is that it assigned every energy domain to a different process whereas the hybrid approaches divided each domain and distributed to four PiP tasks or threads. Thus, the former may require less intercore data movement. We used only four threads per process in the hybrid approaches because of the limitation of the parallel algorithm in SNAP; however, the performance gap is known to increase when more threads are utilized, as already demonstrated in Figure 11.

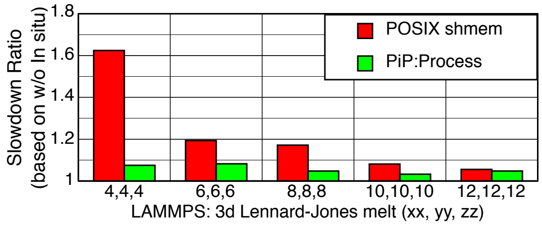


Figure 13: LAMMPS in situ: POSIX shmem vs. PiP running on a single Wallaby/Linux node

7.4 In Situ Analysis of LAMMPS

Our last experiment is a preliminary evaluation of the LAMMPS [23, 33] application attached with an in situ program. The in situ program is a nearest-neighbor program for finding atom pairs having a certain distant range (thought to be chemically bonded). The original generic glue library for combining LAMMPS and the in situ program copies the *Dump* data generated by LAMMPS from the application's space into a POSIX shmem–based shared buffer [10]. The nearest-neighbor program copies the data in the shared-memory buffer to its internal buffer. Thus, the LAMMPS process can continue to compute the next simulation step in parallel with the in situ analysis. The in situ analysis can process the data either by directly accessing the *chunked* data in the shared buffer or by copying the data into a local buffer. Because our analysis program expects all data to be in a linear buffer, we utilized the latter approach in the experiment. In PiP version, the in situ program ran as a PiP task created by LAMMPS. This glue library was modified to utilize VAS sharing of PiP (see Section 4.3). LAMMPS simply passed the address of the *Dump* data, and the nearest-neighbor program copied the data into its internal buffer.

Figure 13 compares the slowdown ratios of using original POSIX shmem and using PiP based on the LAMMPS execution time without having the in situ program. The problem size (x-axis) was varied from $4, 4, 4$ (smallest) to $12, 12, 12$ (largest). In this case, LAMMPS and the in situ program ran on one node, and the single LAMMPS process ran with four OpenMP threads. The PiP-based approach (1-copy) performs more efficiently than does the POSIX shmem–based version (2-copy), resulting in less than 10% slowdown, whereas the latter requires up to 1.6x more overhead. Since the compute time greatly exceeds the data transfer time, however, the benefits of PiP are reduced. This in situ program is computationally intensive, running on the order of $O(P^2)$, where P is the number of atoms, so the slowdown manifests itself early.

8 SUMMARY AND FUTURE WORK

This paper presents a novel technique for supporting VAS sharing with privatized variable sets, in order to have the best of both the multiprocess and multithread execution models. Although this idea is not new, the implementations proposed thus far have depended on either the OS kernel or a programming language system. What

[10] https://github.com/lanl/SNAP

makes our proposed technique, called PiP, unique and practical is that PiP is implemented at the user level, depending only on PIE, the `dlmopen()` function, and the `clone()` system call or `pthread_create()` function. In this paper, we showed that the conventional memory-mapping techniques can suffer from the setup cost for mapping memory pages of the other tasks and PF overhead when accessing the shared region. These overheads can be fully avoided by sharing VAS in PiP.

PiP defines a small set of functions; thus, it is easily integrated into other runtime libraries as a portable low-level support that can be available on various HPC platforms. The PiP model does not take over the conventional multiprocess or multithread execution model. The goal of PiP is to provide an alternative low-level support of these models when necessary. We analyzed the performance characteristics of PiP through a variety of microbenchmarks, and we demonstrated significant performance gain from using PiP in several important HPC scenarios, including use in MPI runtime internal optimizations, integration with the hybrid MPI+OpenMP model, and support of in situ programming for scientific simulations. The evaluations were done in several computing environments including two of the world's top 10 supercomputers. Evaluation results indicate up to 3.2x improved performance in the hybrid particle transport proxy application over 1,024 KNL cores, and a close to 30% reduced slowdown ratio in the LAMMPS application with in situ analysis. All these achievements indicate that PiP is an efficient and practical VAS sharing model for HPC applications that can be applied to large supercomputing systems.

We plan to investigate the utilization of PiP in other communication runtime systems such as OpenSHMEM[6] and XcalableMP[28] as well as with MPI. We will also exploit a comprehensive integration with existing OpenMP implementations.

ACKNOWLEDGMENTS

We acknowledge the contributions made by Matthieu Dreher and Tom Peterka from Argonne National Laboratory through discussions on the utilization of PiP in in situ analysis. We thank the University of Tokyo, the University of Tsukuba, and JCAHPC for letting us access the OFP machine and for their help with getting the experiments done. The material presented in this paper was based upon work supported in part by the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research (SC-21), under contract DE-AC02-06CH11357.

REFERENCES

[1] Abdelhalim Amer, Huiwei Lu, Yanjie Wei, Pavan Balaji, and Satoshi Matsuoka. 2015. MPI+Threads: Runtime Contention and Remedies. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2015)*. ACM, New York, NY, USA, 239–248. http://doi.acm.org/10.1145/2688500.2688522

[2] Utkarsh Ayachit, Brad Whitlock, Matthew Wolf, Burlen Loring, Berk Geveci, David Lonie, and E. Wes Bethel. 2016. The SENSEI Generic in Situ Interface. In *Proceedings of the 2Nd Workshop on In Situ Infrastructures for Enabling Extreme-scale Analysis and Visualization (ISAV '16)*. IEEE Press, Piscataway, NJ, USA, 40–44. https://doi.org/10.1109/ISAV.2016.013

[3] Randal S. Baker. 2013. *PARTISN on Advanced/Heterogeneous Processing Systems*. https://doi.org/10.2172/1063248

[4] Pavan Balaji, Darius Buntinas, David Goodell, William Gropp, Jayesh Krishna, Ewing Lusk, and Rajeev Thakur. 2010. PMI: A Scalable Parallel Process-Management Interface for Extreme-Scale Systems. In *Proceedings of the 17th European MPI Users' Group Meeting Conference on Recent Advances in the Message Passing Interface (EuroMPI'10)*. Springer-Verlag, Berlin, Heidelberg, 31–41. http://dl.acm.org/citation.cfm?id=1894122.1894127

[5] Pavan Balaji, Darius Buntinas, David Goodell, William Gropp, and Rajeev Thakur. 2008. Toward Efficient Support for Multithreaded MPI Communication. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, Alexey Lastovetsky, Tahar Kechadi, and Jack Dongarra (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 120–129.

[6] Ron Brightwell and Kevin Pedretti. 2011. An Intra-Node Implementation of OpenSHMEM Using Virtual Address Space Mapping. In *Fifth Partitioned Global Address Space Conference*. Galveston Island, Texas. http://pgas11.rice.edu/papers/BrightwellPedretti-OpenSHMEM-PGAS11.pdf

[7] Ron Brightwell, Kevin Pedretti, and Trammell Hudson. 2008. SMARTMAP: Operating System Support for Efficient Data Sharing among Processes on a Multi-Core Processor. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing (SC '08)*. IEEE Press, Piscataway, NJ, USA, Article 25, 12 pages. http://dl.acm.org/citation.cfm?id=1413370.1413396

[8] Darius Buntinas and Guillaume Mercier. 2006. Implementation and Shared-Memory Evaluation of MPICH2 over the Nemesis Communication Subsystem. In *Euro PVM/MPI*.

[9] Patrick Carribault, Marc Pérache, and Hervé Jourdren. 2011. *Thread-Local Storage Extension to Support Thread-Based MPI/OpenMP Applications*. Springer Berlin Heidelberg, Berlin, Heidelberg, 80–93. https://doi.org/10.1007/978-3-642-21487-5_7

[10] A. Champsaur, J. Lofstead, J. Dayal, M. Wolf, G. Eisenhauer, P. Widener, and A. Gavrilovska. 2017. SmartBlock: An Approach to Standardizing In Situ Workflow Components. In *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 1301–1308. https://doi.org/10.1109/IPDPSW.2017.149

[11] Hoang-Vu Dang, Sangmin Seo, Abdelhalim Amer, and Pavan Balaji. 2017. Advanced Thread Synchronization for Multithreaded MPI Implementations. In *Proceedings of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid '17)*. IEEE Press, Piscataway, NJ, USA, 314–324. https://doi.org/10.1109/CCGRID.2017.65

[12] J. Dayal, D. Bratcher, G. Eisenhauer, K. Schwan, M. Wolf, X. Zhang, H. Abbasi, S. Klasky, and N. Podhorszki. 2014. Flexpath: Type-Based Publish/Subscribe System for Large-Scale Science Analytics. In *2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. 246–255. https://doi.org/10.1109/CCGrid.2014.104

[13] Erik Demaine. 1997. A Threads-Only MPI Implementation for the Development of Parallel Programs. In *Proceedings of the 11th International Symposium on High Performance Computing Systems*. 153–163.

[14] James Dinan, Pavan Balaji, David Goodell, Douglas Miller, Marc Snir, and Rajeev Thakur. 2013. Enabling MPI Interoperability Through Flexible Communication Endpoints. In *Proceedings of the 20th European MPI Users' Group Meeting (EuroMPI '13)*. ACM, New York, NY, USA, 13–18. https://doi.org/10.1145/2488551.2488553

[15] Ciprian Docan, Manish Parashar, and Scott Klasky. 2010. DataSpaces: An Interaction and Coordination Framework for Coupled Simulation Workflows. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing (HPDC '10)*. ACM, New York, NY, USA, 25–36. https://doi.org/10.1145/1851476.1851481

[16] A. Friedley, G. Bronevetsky, A. Lumsdaine, and T. Hoefler. 2013. Hybrid MPI: Efficient Message Passing for Multi-core Systems. In *IEEE/ACM International Conference on High Performance Computing, Networking, Storage and Analysis (SC13)*. 18:1–18:11.

[17] B. Gerofi, M. Takagi, A. Hori, G. Nakamura, T. Shirasawa, and Y. Ishikawa. 2016. On the Scalability, Performance Isolation and Device Driver Transparency of the IHK/McKernel Hybrid Lightweight Kernel. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 1041–1050. https://doi.org/10.1109/IPDPS.2016.80

[18] Balazs Gerofi, Masamichi Takagi, and Yutaka Ishikawa. 2015. Toward Operating System Support for Scalable Multithreaded Message Passing. In *Proceedings of the 22Nd European MPI Users' Group Meeting (EuroMPI '15)*. ACM, New York, NY, USA, Article 1, 10 pages. https://doi.org/10.1145/2802658.2802661

[19] R. Gerstenberger, M. Besta, and T. Hoefler. 2013. Enabling Highly-Scalable Remote Memory Access Programming with MPI-3 One Sided. In *2013 SC - International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. 1–12. https://doi.org/10.1145/2503210.2503286

[20] David Goodell, William Gropp, Xin Zhao, and Rajeev Thakur. 2011. *Scalable Memory Use in MPI: A Case Study with MPICH2*. Springer Berlin Heidelberg, Berlin, Heidelberg, 140–149. https://doi.org/10.1007/978-3-642-24449-0_17

[21] Torsten Hoefler, James Dinan, Darius Buntinas, Pavan Balaji, Brian W. Barrett, Ron Brightwell, William Gropp, Vivek Kale, and Rajeev Thakur. 2012. Leveraging MPI's One-Sided Communication Interface for Shared-Memory Programming. In *Proceedings of the 19th European Conference on Recent Advances in the Message Passing Interface (EuroMPI'12)*. Springer-Verlag, Berlin, Heidelberg, 132–141. https://doi.org/10.1007/978-3-642-33518-1_18

[22] Yutaka Ishikawa. 1996. MPC++ Approach to Parallel Computing Environment. *SIGAPP Appl. Comput. Rev.* 4, 1 (April 1996), 15–18. https://doi.org/10.1145/240732.240738

[23] Sandia National Laboratories. 2017. LAMMPS Molecular Dynamics Simulator. (2017). http://lammps.sandia.gov/.

[24] Shigang Li, Torsten Hoefler, and Marc Snir. 2013. NUMA-Aware Shared-Memory Collective Communication for MPI. In *Proceedings of the 22Nd International Symposium on High-performance Parallel and Distributed Computing (HPDC '13)*. ACM, New York, NY, USA, 85–96. https://doi.org/10.1145/2462902.2462903

[25] J. C. Díaz Martín, J. A. Rico Gallego, J. M. Álvarez Llorente, and J. F. Perogil Duque. 2009. An MPI-1 Compliant Thread-Based Implementation. In *Proceedings of the 16th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Springer-Verlag, Berlin, Heidelberg, 327–328. https://doi.org/10.1007/978-3-642-03770-2_42

[26] P. J. Mendygral, N. Radcliffe, K. Kandalla, D. Porter, B. J. O'Neill, C. Nolting, P. Edmon, J. M. F. Donnert, and T. W. Jones. 2017. WOMBAT: A Scalable and High-Performance Astrophysical Magnetohydrodynamics Code. *The Astrophysical Journal Supplement Series* 228, 2 (2017), 23. http://stacks.iop.org/0067-0049/228/i=2/a=23

[27] Dmitriy Morozov and Zarija Lukic. 2016. Master of Puppets: Cooperative Multitasking for In Situ Processing. In *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing (HPDC '16)*. ACM, New York, NY, USA, 285–288. https://doi.org/10.1145/2907294.2907301

[28] M. Nakao, J. Lee, T. Boku, and M. Sato. 2012. Productivity and Performance of Global-View Programming with XcalableMP PGAS Language. In *2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*. 402–409. https://doi.org/10.1109/CCGrid.2012.118

[29] Jun Nakashima and Kenjiro Taura. 2014. *MassiveThreads: A Thread Library for High Productivity Languages*. Springer Berlin Heidelberg, Berlin, Heidelberg, 222–238. https://doi.org/10.1007/978-3-662-44471-9_10

[30] Mathias Payer. 2012. *Too Much PIE Is Bad for Performance*. Technical Report 766, ETH Zürich, Switzerland. Department of Computer Science, ETH Zürich. http://dx.doi.org/10.3929/ethz-a-007316742

[31] Marc Pérache, Patrick Carribault, and Hervé Jourdren. 2009. *MPC-MPI: An MPI Implementation Reducing the Overall Memory Consumption*. Springer Berlin Heidelberg, Berlin, Heidelberg, 94–103. https://doi.org/10.1007/978-3-642-03770-2_16

[32] Marc Pérache, Hervé Jourdren, and Raymond Namyst. 2008. MPC: A Unified Parallel Runtime for Clusters of NUMA Machines. In *Proceedings of the 14th International Euro-Par Conference on Parallel Processing (Euro-Par'08)*. Springer-Verlag, Berlin, Heidelberg, 78–88. https://doi.org/10.1007/978-3-540-85451-7_9

[33] Steve Plimpton. 1995. Fast Parallel Algorithms for Short-Range Molecular Dynamics. *J. Comput. Phys.* 117, 1 (1995), 1–19. https://doi.org/10.1006/jcph.1995.1039

[34] S. Seo, A. Amer, P. Balaji, C. Bordage, G. Bosilca, A. Brooks, P. Carns, A. Castello, D. Genet, T. Herault, S. Iwasaki, P. Jindal, L. V. Kale, S. Krishnamoorthy, J. Lifflander, H. Lu, E. Meneses, M. Snir, Y. Sun, K. Taura, and P. Beckman. 2018. Argobots: A Lightweight Low-Level Threading and Tasking Framework. *IEEE Transactions on Parallel and Distributed Systems* 29, 3 (March 2018), 512–526. https://doi.org/10.1109/TPDS.2017.2766062

[35] Akio Shimada, Balazs Gerofi, Atsushi Hori, and Yutaka Ishikawa. 2013. Proposing A New Task Model towards Many-Core Architecture. In *Proceedings of the ACM international workshop on Manycore Embedded Systems 2013 (MES'13)*. ACM, Tel-Aviv, Israel.

[36] Akio Shimada, Atsushi Hori, and Yutaka Ishikawa. 2014. Eliminating Costs for Crossing Process Boundary from MPI Intra-Node Communication. In *Proceedings of the 21st European MPI Users' Group Meeting (EuroMPI/ASIA '14)*. ACM, New York, NY, USA, Article 119, 2 pages. https://doi.org/10.1145/2642769.2642790

[37] S. Sridharan, J. Dinan, and D. D. Kalamkar. 2014. Enabling Efficient Multithreaded MPI Communication through a Library-Based Implementation of MPI Endpoints. In *SC14: International Conference for High Performance Computing, Networking, Storage and Analysis*. 487–498. https://doi.org/10.1109/SC.2014.45

[38] Erich Strohmaier, Jack Dongarra, Horst Simon, and Martin Meuer. 2017. Top500 List - November 2017. (2017). https://www.top500.org/list/2017/11.

[39] Hong Tang, Kai Shen, and Tao Yang. 1999. Compile/Run-time Support for Threaded MPI Execution on Multiprogrammed Shared Memory Machines. *SIGPLAN Not.* 34, 8 (May 1999), 107–118. https://doi.org/10.1145/329366.301114

[40] M. Tchiboukdjian, P. Carribault, and M. Pérache. 2012. Hierarchical Local Storage: Exploiting Flexible User-Data Sharing between MPI Tasks. In *2012 IEEE International Parallel Distributed Processing Symposium (IPDPS)*. 366–377. https://doi.org/10.1109/IPDPS.2012.42

[41] K. B. Wheeler, R. C. Murphy, and D. Thain. 2008. Qthreads: An API for Programming with Millions of Lightweight Threads. In *2008 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 1–8. https://doi.org/10.1109/IPDPS.2008.4536359

[42] Wikipedia. 2016. Address Space Layout Randomization. (2016). https://en.wikipedia.org/wiki/Address_space_layout_randomization.

[43] Michael Woodacre, Derek Robb, Dean Roe, and Karl Feind. 2003. *White Paper: The SGI Altix 3000 Global Shared-Memory Architecture*. Technical Report. SGI.

[44] Mitsuo Yokokawa, Fumiyoshi Shoji, Atsuya Uno, Motoyoshi Kurokawa, and Tadashi Watanabe. 2011. The K Computer: Japanese Next-Generation Supercomputer Development Project. In *Proceedings of the 17th IEEE/ACM international symposium on Low-power electronics and design (ISLPED '11)*. IEEE Press, Piscataway, NJ, USA, 371–372. http://dl.acm.org/citation.cfm?id=2016802.2016889

[45] Fang Zheng, Hongbo Zou, Greg Eisenhauer, Karsten Schwan, Matthew Wolf, Jai Dayal, Tuan-Anh Nguyen, Jianting Cao, Hasan Abbasi, Scott Klasky, et al. 2013. FlexIO: I/O Middleware for Location-Flexible Scientific Data Analytics. In *2013 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 320–331. https://doi.org/10.1109/IPDPS.2013.46

[46] G. Zheng, S. Negara, C. L. Mendes, L. V. Kale, and E. R. Rodrigues. 2011. Automatic Handling of Global Variables for Multi-Threaded MPI Programs. In *2011 IEEE 17th International Conference on Parallel and Distributed Systems (ICPADS)*. 220–227. https://doi.org/10.1109/ICPADS.2011.33