Daps: A Dynamic Asynchronous Progress Stealing Model for MPI Communication

1st Kaiming Ouyang
Computer Science & Engineering
University of California, Riverside
Riverside, USA
kouya001@ucr.edu

2nd Min Si
Mathematics and Computer Science
Argonne National Laboratory
Lemont, USA
msi@anl.gov

3th Astushi Hori
National Institute of Informatics
Tokyo, Japan
ahori@nii.ac.jp

4rd Zizhong Chen
Computer Science & Engineering
University of California, Riverside
Riverside, USA
chen@cs.ucr.edu

5th Pavan Balaji
Facebook
Menlo Park, USA
pavanbalaji@gmail.com

***Abstract*—MPI provides nonblocking point-to-point and one-sided communication models to help applications achieve communication and computation overlap. These models provide the opportunity for MPI to offload data transfer to low level network hardware while the user process is computing. In practice, however, MPI implementations have to often handle complex data transfer in software due to limited capability of network hardware. Therefore, additional asynchronous progress is necessary to ensure prompt progress of these software-handled communication. Traditional mechanisms either spawn an additional background thread on each MPI process or launch a fixed number of helper processes on each node. Both mechanisms may degrade performance in user computation due to statically occupied CPU resources. The user has to fine-tune the progress resource deployment to gain overall performance. For complex multiphase applications, unfortunately, severe performance degradation may occur due to dynamically changing communication characteristics and thus changed progress requirement. This paper proposes a novel Dynamic Asynchronous Progress Stealing model, called Daps, to completely address the asynchronous progress complication. Daps is implemented inside the MPI runtime. It dynamically leverages idle MPI processes to steal communication progress tasks from other busy computing processes located on the same node. The basic concept of Daps is straightforward; however, various implementation challenges have to be resolved due to the unique requirements of interprocess data and code sharing. We present our design that ensures high performance while maintaining strict program correctness. We compare Daps with state-of-the-art asynchronous progress approaches by utilizing both microbenchmarks and HPC proxy applications.**

***Index Terms*—Asynchronous Progress, Progress Stealing, PiP, MPI**

I. Introduction

MPI [1] is widely used in high-performance computing (HPC) applications running on distributed-memory systems. To optimize communication overhead that may become expensive especially in large-scale executions, application developers often overlap communication and computation by leveraging nonblocking MPI communication functions such as nonblocking send/receive and one-sided (also known as RMA) operations. Although these functions provide the semantics to decouple the issuing and completion steps of a communication (e.g., a nonblocking send/receive can be completed by a separate call to `MPI_Wait`), the underlying MPI implementation may not be able to continue processing the data transfer asynchronously because of limitations from network hardware support or complex MPI-level protocols. For instance, an `MPI_Accumulate` with a noncontiguous data array has to be emulated in MPI software (also known as *active messages*) because such a complex atomic operation still cannot be handled by the network hardware of most HPC interconnects. A send/receive with a large message often involves an additional handshake to ensure zero-copy optimization [2].

Traditionally, the user program has to make frequent MPI calls to ensure prompt processing of these internal package exchanges, causing overcomplicated user code. Alternatively, a number of asynchronous progress mechanisms have been developed. For instance, asynchronous threading [3]–[5] is the most commonly supported asynchronous progress mechanism where a background thread is spawned on each MPI process to actively poll the progress for that process. Each background thread is usually bound onto dedicated idle CPU cores or share the same core of the MPI process. Although it helps communication, severe performance degradation may occur in user computation beeause of reduced computing resource or overhead from core contention. Casper [6], [7] introduces a process-based asynchronous progress mechanism to address the issues of the thread model. It allows the user to keep aside a few CPU cores and launch a ghost process on each core. The ghost process can help advance communication for other MPI processes on each node. Nevertheless, both models all fall into the *static asynchronous progress model* where the communication progress resource (i.e., CPU cores) has to be statically set when executing a program. Because of such a limitation, the user has to determine the optimal resource configuration (i.e., number of dedicated cores and core binding strategy in the thread model; number of ghost processes in Casper) through repeated experiments, causing an extra burden on the user that is tedious and time wasting. What is worse, scientific applications often involve multiple stages and each stage may contain different computation and communication workloads. Thus, different progress resource configuration is preferred for each stage. Unfortunately, the static progress model cannot dynamically adjust a dedicated progress resource at runtime. Hence, the user has to make a tradeoff among multiple stages suffering from suboptimal overall performance.

In this paper we present a novel dynamic asynchronous progress-stealing (Daps) model that eliminates the drawbacks of the traditional static progress model. The core notion of Daps is to *dynamically determine idle MPI processes at runtime and utilize them to perform MPI internal progress tasks for the other busy computing processes on the node*. Thus, we call this model "progress stealing." Daps internally manages the dynamic stealing and thus does not require the user to make a decision about the appropriate resource configuration of asynchronous progress. More important, Daps can adapt to complex multistage applications and deliver the optimal overall performance. The reason is that Daps utilizes only an idle MPI process to perform progress tasks and thus does not statically occupy any computing resources.

We note that although both work stealing [8]–[11] and MPI asynchronous progress [3]–[7] have been heavily studied in the HPC field, to our best knowledge Daps is the first work that combines them together and delivers improved performance.

We implement Daps inside MPICH by leveraging the Process-in-Process (PiP) memory-sharing technique [12] for interprocess task stealing. Interprocess task stealing in multiprocess parallelism (e.g., the MPI model) brings in various implementation challenges compared with traditional task-stealing techniques in multithread parallelism, mainly because of different low-level data- and code-sharing mechanisms between processes and threads. Unlike our previous work CAB-MPI [11], which can steal only the basic memory copy and MPI reduce operation tasks between processes, Daps steals an arbitrary progress task defined in MPI that may involve complex code context and interaction with external libraries and low-level network drivers. In this paper we make a thorough analysis of all prerequisites and challenges of Daps and present efficient solutions for the MPI communication environment.

We demonstrate the benefit of Daps by applying it to the widely used MPI nonblocking communication routines including point-to-point `MPI_Isend/Irecv` and RMA operations (e.g., `MPI_Put/Get/Accumulate`). We compare Daps with the state-of-the-art static asynchronous progress approaches in both microbenchmarks and computational kernels on an Intel Omni-Path cluster. The results demonstrate that the Daps model is truly efficient and adaptive for both single-stage and multistage MPI applications.

II. Background

In this section, we briefly introduce the MPI progress concept.

A. MPI Progress

An ideal MPI implementation is to directly offload all communication requests (e.g., a send or a put) to the low-level network or translate to shared memory load/store. In reality, however, MPI often has to require additional packets exchanges to process complex data movement or for performance optimization. For instance, an RMA operation defines the datatype of both origin and target buffers on the local process. When the target datatype is noncontiguous, a typical implementation is to let the local process send both the target datatype metadata and the data to the remote process via an internal active message. An MPI call on the remote process can internally process this incoming active message (i.e., unpack the data into the window buffer) and send back an acknowledgment when transmission completes.[1] In the point-to-point model, for example, a receiver-first nonblocking receive often posts only the request to the MPI internal queue. The receiver process can handle the request matching with an incoming message and perform the actual data transfer at a later MPI call (e.g., in `MPI_Wait`). If the message is large, the typical rendezvous protocol often sends only the message metadata in the first round handshake and lets the sender or receiver perform direct RDMA for actual data transfer. An acknowledgment is required to notify the other side in order to return the buffer to the user. Similar protocols are used also in intranode communication (e.g., a handshake following with a POSIX-shared-memory-based pipeline copy or XPMEM-based single copy). For simplicity, in this paper "active message" refers to all these multipacket protocols.

To properly handle various internal active messages, MPI implementations often define a generic progress routine (also known as a progress engine) that receives any incoming internal packet and dispatches to the corresponding callback function to process the packet. The progress engine consists of a network routine and a shared-memory routine. As showcased in the above examples, the callback function can involve different code logic and often interacts with external libraries (e.g., memory allocation via Glibc) as well as low-level network drivers (e.g., sending an acknowledgment).

B. Asynchronous Progress

In order to ensure prompt processing of the internal active messages, MPI has to ensure that the progress routine is frequently triggered. When the user process is busy in computation outside MPI, however, it cannot make MPI calls until the computation completes. Consequently, an arbitrary long delay may occur in the communication [13], [14]. Asynchronous progress is the mechanism to ensure that the MPI progress

[1] An implementation may choose to process a noncontiguous RMA operation by issuing multiple network RDMA operations each carrying a chunk of the data. However, such an approach is expensive especially for sparse data layout. Therefore, the active-message-based implementation is still the norm.

can be asynchronously triggered even when the user process is outside MPI, thus achieving communication and computation overlap. Three mechanisms have been well studied in the community: thread-based [3]–[5], process-based [6], [15], and system interrupt-based [16]–[20]. Because of the the lack of portability and significant overhead of the interrupt-based mechanism, the former two are more commonly used on mainstream HPC systems. Thus, our work omits the comparison with the interrupt-based mechanism. We detail the design of the above state-of-the-art asynchronous progress mechanisms in Section VII.

III. Limitation of Static Asynchronous Progress

In both thread-based and process-based asynchronous progress mechanisms, the user has to statically configure the amount of progress resources, often a few "likely idle" CPU cores that can be occupied from a multicore or many-core node. The user has to make repeated experiments to understand the application characteristics in order to determine the optimal number. The thread-based mechanism usually allows the user to choose either dedicating 50% of CPU cores for the background threads or oversubscribing cores.[2] The process-based Casper mechanism is more flexible in that it can specify an arbitrary number of cores used for asynchronous progress. Nevertheless, once the setting is specified, when s execution starts, that number can no longer change during runtime. Hence, both mechanisms follow a static asynchronous progress model.

Scientific applications often consists of multiple solvers that form a multistage execution. At each stage, the communication and computation characteristics can be completely different. Consequently, the number of "likely idle" cores varies. For instance, in the quantum chemistry application suite NWChem [21], the "gold-standard" CCSD(T) task contains four stages: self-consistent field (SCF), four-index transformation (4-index), CCSD iteration, and the noniterative (T) portion. As reported in [7], [20], the (T) portion is extremely computationally expensive whereas the others are more communication intensive.

Clearly, the static model cannot provide optimal performance for multistage applications. To demonstrate the performance impact, we extended the block sparse matrix multiplication (BSPMM) proxy application of NWChem to mimic a two-stage execution (see details of two-stage BSPMM in Section VI-B1). We compare the execution time of original MPI without asynchronous progress (Baseline) and static asynchronous progress mechanisms (Thread and Casper) with varying numbers of cores dedicated to the progress thread or process (see the definition of the experimental platform and baseline MPI in Section VI). The remaining cores run user processes. Figure 1 shows the breakdown of the computation time and the communication time of each stage. Clearly, neither Thread nor Casper can deliver the best-performing result (shown as Ideal). The reason is that the first stage needs only two cores (Thread-2 and Casper-2) to make communication progress and prefers to use the majority of cores to speed up the computation. Dedicating more cores at stage 1 only slows down the computation (T_{comp1}). On the other hand, the second stage suffers from a communication bottleneck when using only a few cores for communication progress. Thus, using more progress cores helps (see T_{comm2}). Unfortunately, the state-of-the-art static model cannot adjust the number of progress cores for the second stage. Even the best-performing Casper-2 shows a 30% slowdown if we compare with the ideal time.

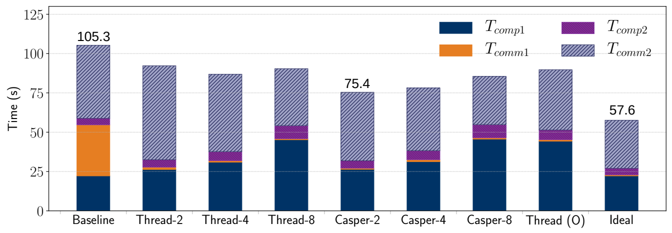


Fig. 1: Two-stage BSPMM execution time on 4 Broadwell nodes each with 36 cores. The first stage ($T_{comp1} + T_{comm1}$) is computation intensive, and the second stage ($T_{comp2} + T_{comm2}$) is communication intensive. The fifure compares the original MPI (Baseline), thread-based asynchronous progress with core oversubscription (denoted by Thread (O)), and thread-based asynchronous progress and Casper with varying numbers of cores dedicated to progress (denoted by Thread-N and Casper-N where N=$2, 4, 8$; the number of user processes are $34, 32, 28$, respectively). The Ideal time is estimated by combining the best T_{comp1} and T_{comm1} from Baseline and Casper-2, and T_{comp2} and T_{comm2} from Baseline and Casper-8, respectively.

IV. Dynamic Asynchronous Progress Design

To address the critical limitation of the static mechanisms, we design Daps that enables fully dynamic MPI asynchronous progress. Daps is a novel *progress-stealing* model that explores the work-stealing scheme to dynamically balance progress tasks in the multiprocess space. The foundation of Daps is the flexible data- and code-sharing capability provided by the underlying interprocess memory-sharing techniques. In this section, we introduce the basic concept of Daps together with a comprehensive analysis of the data- and code-sharing prerequisites that ensure the design correctness.

A. Basic Definition

In a work-stealing scheme, the common objects are task, task worker, and task owner. We define similar concepts in Daps.

1) Progress Task: Section II-A described the MPI progress concept. A progress task consists of the network progress step and the shared-memory (shm) progress step that handle incoming active messages from network and shared memory,

[2]To ensure fairness, we modified the MPI implementation to internally bind the background threads to user-specified number of dedicated cores.

respectively. We separately define network and shm progress tasks for each process because of their different overheads. We detail the progress-overhead-aware design in Section IV-C. Inside a progress task, it primarily polls the low-level progress (e.g., via a call to Libfabric `fi_cq_read` function on the Omni-Path platform) to receive an incoming packet and then triggers the corresponding handling function (callback) defined in MPI. A callback may consist of various internal steps such as memory copy (e.g., move network transferred data from a temporary contiguous buffer to the destination noncontiguous buffer), computation of a reduce operation, issuing of a network packet (e.g., for acknowledgment), and memory allocation for any temporary buffer. Some of the internal steps cause implementation challenges in the interprocess-stealing scheme. We give a systemic diagnosis and propose solutions in Section V.

2) Progress Worker: Any MPI process that is idly waiting in an MPI blocking call (e.g., `MPI_Wait` in a point-to-point communication, or `MPI_Win_flush` in RMA) can become a progress worker. But if the process has to handle any pending internal incoming or outgoing messages once having polled the progress routines, it becomes busy and thus cannot continue stealing. If the process is in an MPI nonblocking call (e.g., `MPI_Test`), we define it as a busy process because it is expected to return to the user program immediately.

3) Progress Owner: The owner is the process that originally receives the incoming data.

Putting the above objects together, a *progress stealing* is the procedure where a progress worker checks and executes the progress task of the selected progress owner. Only the owner that is likely busy in the user computation will be selected.

B. Prerequisite Analysis

A progress task in Daps may involve arbitrary code logic as defined in the preceding section. A correct progress stealing requires the worker process to access the task code and data belonging to the owner process and execute the task code the same as that executed by the owner. Code sharing is a concept commonly explored in the multithread model. In the multiprocess model (e.g., MPI), however, only data sharing has been studied so far [11], [12], [22], [23]. To the best of our knowledge, *Daps is the first work that explores code sharing for multiprocess programs*.

We define three prerequisites for a correct progress stealing:

- Data Sharing: All global, static, and private data of the progress owner must be successfully accessed by the progress workers.
- Code Sharing: The progress worker must be able to access any code blocks loaded into the address space of the owner.
- Shared Code Execution: The progress worker must execute the shared code instructions and deliver exactly the same resulting state as that performed by the owner itself.

To this end, we investigate five widely studied interprocess memory-sharing techniques in the HPC domain: POSIX SHM [24], Cross-Memory-Attach (CMA) [25], KNEM [26], XPMEM [27], and Process-in-Process (PiP) [12]. We analyze their capability following the above prerequisites.

POSIX SHM, CMA, and KNEM are designed for optimizing interprocess data transmission on a node. POSIX SHM allows the user to allocate a shared buffer for two processes. CMA and KNEM allow a process to directly read or write a buffer allocated by the other process via a kernel-assisted approach. None of these techniques can share code.

XPMEM allows a process to expose a section of its virtual address space (VAS) so that the other process can map the exposed section to its own VAS. Both code and data sections can be exposed and mapped.

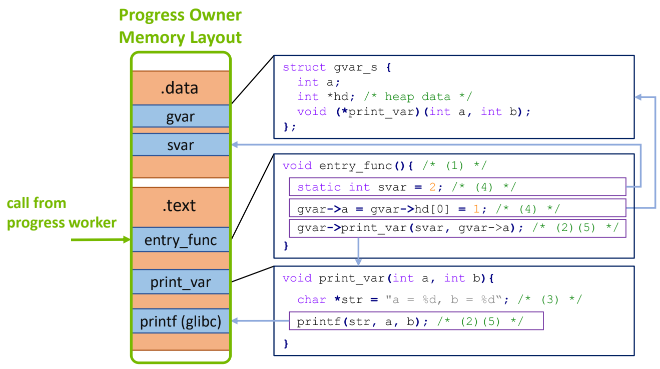
PiP spawns multiple PiP tasks into the same VAS. Unlike a thread, all statically allocated variables are privatized and behave like a process. Similar to a thread, a PiP task can directly access the data and code of another task. Previous work [11], [12] has demonstrated the usage of PiP tasks as MPI processes and optimized the interprocess data transmission via data sharing. We clearly distinguish these works with Daps in Section VII.

We further assess the shared code execution capability of XPMEM and PiP. The key questions are (1) whether the shared functions can be correctly executed, (2) whether the function parameters can be correctly passed, (3) whether the stack variable can be correctly allocated and referenced in the shared function, (4) whether the global, static, and heap data can be correctly referenced, and (5) whether any internal function of the shared function can be executed. For instance, Figure 2 showcases the expected correct behaviors during a progress stealing. The progress worker is expected to access the shared `entry_func()` function defined in the owner's text segment. Once the worker calls into the function, it is expected to reference the static variable `svar` and the global variable `gvar` allocated in the owner's data segment. The shared function may invoke other function or function pointer (`gvar->print_var()`) or external library function (`printf()` from Glibc), which should be correctly accessed by the worker. Moreover, the stack variables (`a`, `b`, and `str` in `print_var()`) must be correctly allocated, value assigned, and referenced.

XPMEM can expose the entire VAS of a process and map it to the other process's VAS. Thus, all code and data can be theoretically shared at MPI initialization. [3] However, the mapped address range is usually different from that on the owner process. This causes severe correctness fault in our stealing scheme. For example, a variable or a function is usually referenced by its address in the assembly code. The referenced address is undefined or defined for other data on the worker's VAS. Consequently, undefined behavior may occur when the worker executes the mapped code.

Unlike XPMEM, PiP can correctly support shared code execution thanks to the shared VAS scheme. The addresses of any global, static, or heap variable and function are consistent

[3] Such a coarse-grained memory mapping may cause other drawbacks such as heavy page mapping overhead. Nevertheless, it is beyond the scope of this paper, and we omit the discussion.



```
struct gvar_s {
  int a;
  int *hd; /* heap data */
  void (*print_var)(int a, int b);
};
```

```
void entry_func(){ /* (1) */
  static int svar = 2; /* (4) */
  gvar->a = gvar->hd[0] = 1; /* (4) */
  gvar->print_var(svar, gvar->a); /* (2)(5) */
}
```

```
void print_var(int a, int b){
  char *str = "a = %d, b = %d"; /* (3) */
  printf(str, a, b); /* (2)(5) */
}
```

Fig. 2: Expected behaviors when a progress worker calls into `entry_func()` shared by the owner; the corresponding assessment question ID is marked behind each line.

on all processes located in the same PiP VAS. Thus, they can be correctly referenced. A stack variable is automatically allocated and stored on the executing process's stack (i.e., the worker's stack in a progress stealing) during its lifetime.

So far, our analysis indicates that PiP is the most promising low-level memory sharing technique to support the interprocess progress stealing. Hence, we implement Daps based on PiP-aware MPICH [12], and the baseline in the remaining sections refers to PiP-aware MPICH (extended from bb595ca0 of MPICH main branch).

C. Progress-Stealing Algorithm

To ensure efficient stealing, the stealing algorithm of Daps is locality aware, progress overhead aware, and MPI context aware. We describe each aspect in detail.

1) Locality Aware: Similar to previous work-stealing studies [11], [28], [29], locality awareness helps the stealing procedure reduce the cache miss penalty when the data is accessed by both the worker and the owner. The same principle can be applied to our case, where a progress worker may touch the user communication buffer (e.g., moving data from a temporary buffer to the user destination buffer) during stealing, and the destination buffer is then accessed by the owner after communication completes (e.g., using the data to compute in the user program). On modern multicore and many-core architectures, we expect that NUMA brings in the most significant performance impact. Thus, our algorithm prioritizes local NUMA progress stealing. We note, however, that when no local progress task is available, the worker may still handle the remote task located on a remote NUMA node because the delay caused by lack of asynchronous progress is usually more significant than the cache miss penalty. Our previous work [11] discussed locality-aware stealing in detail. We follow a similar approach in Daps and thus omit the locality impact analysis.

2) Progress Overhead Aware: Stealing a shm progress task is expected to be more expensive than stealing a network progress task. The reqson is that the shm progress task often involves blocking memory copy (e.g., copy data from the user source buffer to the destination buffer after a handshake in the rendezvous protocol). The blocking copy can be significantly expensive in a large data transmission. In a network progress task, on the other hand, even if a progress task involves user data transmission, the transmission is offloaded to network hardware so that the progress worker can return. The completion of the network transmission may be checked asynchronously in a separate progress task. Taking into account the progress overhead difference, we prioritize network progress stealing. When the owner exposes both the network progress task and the shm progress task, the worker always first picks the network one. The shm task is picked only when the network task had nothing to do (i.e., no incoming active message from the network). This design allows us to maximize the stealing throughput.

3) MPI Context Aware: A progress task can be empty when no incoming active message arrives on the owner. Arbitrarily stealing the progress of the other process can cause severe contention overhead because the progress routines is often protected in a critical section in most MPI implementations. Therefore, we define two MPI-context-aware rules to reduce invalid stealing. *Rule 1*: If the progress owner can poll the progress soon (e.g., the owner is in a blocking MPI call) no worker can concurrently steal from the owner. It also ensures good data locality. *Rule 2*: Only the progress task with a likelihood of receiving an incoming active message can be stolen.

We implement the rules by defining three atomic flags on each process. The first flag is `in_progress`. It is set to true when the owner itself is capable of making progress (*Rule 1*) or a worker is already handling the progress task for the owner. The flag is reset once the progress task completes. The other two flags are `net_avail` and `shm_avail`, which indicate the likelihood of receiving an incoming active message in the network progress and shm progress, respectively. We estimate the likelihood based on the MPI context. Specifically, if the process posts a nonblocking receive that involves any active-message-based handshake (e.g., for a rendezvous protocol), we enable `net_avail` or `shm_avail` based on the source rank's location. For the receive with `MPI_ANY_SOURCE`, we have to enable both flags because we do not know whether network or shm will receive the message. A similar approach can be used for collectives. For RMA, however, we can estimate only based on the window creation. We enable the flags on a process whenever it creates a window with a nonzero buffer since this is an indication of remote access. We cannot make a more fine-grained estimation similar to point-to-point because the origin process in the RMA model specifies both synchronization (e.g., `MPI_Win_lock`) and communication (e.g., `MPI_Put`). The target side is unaware of such incoming RMA accesses.

V. Implementation Challenges

The progress-stealing scheme is promising; however, we faced two challenges when implementing the novel interprocess stealing. These challenges are caused mainly by the traditional operating system process-based implementation in low-level libraries such as network user space drivers and Glibc. It assumes that the internal structure of a process is never accessed by the other process. We expect that software stack codesign is becoming the norm in HPC. Thus, the multithread-like weak process model may be also adapted in low-level libraries to fully benefit from the performance gain [11], [12]. Nevertheless, in this paper we systemically diagnose the issues with the current software stack and present solutions.

A. *Network Interaction*

A special interaction between the MPI progress task and the external library is that a task can issue a network data transmission (e.g., when issuing an acknowledgment or issuing a RDMA read/write after a handshake in the rendezvous protocol). On modern HPC interconnects (e.g., InfiniBand and Intel Omni-Path (OPA)), a network data transmission is cooperatively handled by the low-level network drivers in both the user space and kernel space to synchronize with the network adapter. Different network architectures may have different designs. Because of space limitation, we focus on the OPA platform in this paper with the open-source Libfabric [30] and Intel Performance Scaled Messaging 2 (Psm2) [31] libraries as the low-level user space drivers. The OPA-stack analysis also can be insightful for adaptation on other RDMA architectures.

A typical communication approach between the user-space driver and the corresponding kernel module is system calls with a preopened file descriptor. For instance, Psm2 contains the send direct memory access (SDMA) engine for large data transmission [31], [32]. It preallocates a number of SDMA slot queues to offload RDMA data transmission to the network adapter. When issuing a RDMA write, for example, Psm2 stores the metadata (e.g., data address and size) of the transmission into an SDMA header descriptor and passes down to the kernel module via a system call `writev`. Then, the kernel module can properly issue an RDMA transmission via the corresponding SDMA queue by referencing based on the file descriptor. The problem that occurs during a progress stealing is that each process initializes a different file descriptor and shares it with the corresponding kernel thread. If a worker steals the progress task and issues an RDMA for the owner, it uses the owner's endpoint and thus the owner's file descriptor. Such a file descriptor is invalid for the kernel thread of the worker, consequently causing undefined behavior.

One can enable stealing-awareness in the low-level libraries in several ways. Here we use a simple yet efficient approach. We modified Psm2 to bypass the kernel notification if the current process is a progress worker. It allows the worker to asynchronously handle the user space progress (i.e., most software instructions in MPI, Libfabric, and Psm2) but leave the final kernel notification to either the Psm2 background thread [4] or the progress owner itself, whoever arrives first.

[4]A low-frequency progress thread to avoid memory overflow in case the user cannot consume the received data. It can make progress neither for MPI nor for Libfabric.

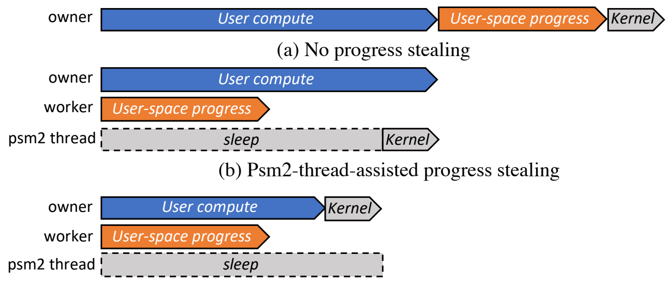


Fig. 3: Progress-stealing timeline with Psm2 network driver.

One concern may be that the delayed kernel notification may degrade performance. We analyzed the possible scenarios with and without progress stealing and compare the timeline in Figure 3. Figure 3(a) shows the timeline without progress stealing. The owner has to handle the user space progress and kernel notification after it finishes the computation. Figures 3(b) and (c) demonstrate two possible cases when stealing is enabled. In the extreme scenario, the owner may finish its computation and be ready to progress when a worker just stole its task. Then, the owner has to wait until the stealing completes. The execution time of such a progress task becomes $(T\_u + T\_k)$, where T_u indicates the time to process the user-space progress and T_k is the kernel notification overhead. It is the same as that without Daps (e.g., Figure 3 (a) with a zero computation). Whenever the user computation is not zero, Daps improves performance.

B. *Thread Local Storage*

The Thread Local Storage (TLS) mechanism allows each thread to allocate distinct instance for a static or global variable. A TLS variable is declared by using the `__thread` attribute. The implementation of TLS is architecture specific. However, we notice that the PiP-based progress-stealing implementation may not be compatible with TLS. Specifically, we define that a correct stealing must ensure that the global and static variables of a progress owner can be correctly accessed by the worker during stealing (see prerequisites definition in Section IV-B). The same rule applies to the TLS variable. Unfortunately, accessing a remote TLS variable in our current implementation may cause an incorrect result or even segmentation fault.

We note that the goal of this paper is to enable interprocess progress stealing in the MPI-specific context but not to implement a generic interprocess work stealing. Thus, we seek only MPI-specific solutions. A systemic analysis of the TLS issue in generic interprocess stealing is left for future work.

To this end, we survey the usage of TLS in all libraries that may be called in an MPI progress task. Table I includes the libraries used in the MPI stacks for all major HPC interconnects: the Intel OPA network (libfabric, libpsm2), Mellanox InfiniBand (libucx, libibverbs, librdmacm), and Cray interconnects (libfabric, libugni[5]).

[5]We check the close-source libugni via `readelf`.

Lib (Version)	Funcs/Vars	Purpose
libfabric (1.10.1)	(1) gnix_debug_pid (2) gnix_debug_tid (3) cntr_test_tid	(1), (2) gnix prov debug var (3) gnix prov test var
libpsm2 (0201)	-	-
libugni (6.0.14)	-	-
libnl-3 (3.2.25)	-	-
libucx (1.9)	(1) ucs_profile_thread_expand_locations (2) ucs_profile_record (3) ucs_profile_dump	(1), (2), (3) profiling and debug func
libibverbs (33.1)	-	-
librdmacm (33.1)	(1) socket (2) fds_alloc (3) rs_fds_alloc	(1) socket hook func (2) socket fd buffer alloc func (3) rsocket fd buffer alloc func
Glibc (2.17)	**(1) malloc/free/calloc/realloc/posix_memalign** (2) errno/h_errno (3) __resp (4) _dlerror_run (5) strsignal	**(1) manage heap memory func** (2) function call return state var (3) DNS resolver var (4) load dynamic binary func (5) describe signal func

TABLE I: Survey of TLS variable usage in the MPI progress stack. We summarize the functionality of each TLS variable and its purpose. We highlight the variables that can impact on the correctness of Daps.

We notice that the usage of TLS is not common in network low-level libraries. Some TLS variables are used only for debugging (i.e., in libfabric and libucx). In librdmacm, TLS variables are defined only for the socket or socket-over-RDMA protocol, which is not used in any progress task of MPI. In Glibc, the variables `errno` and `h_errno` report the error number whenever a Glibc function call fails. In all call paths of the progress task, we confirm that these variables are not used. Moreover, `__resp`, `_dlerror_run`, and `strsignal` are irrelevant to MPI progress tasks as well.

The Glibc memory allocation functions bring more concern in our case. An MPI callback function commonly may internally allocate a temporary buffer and free it when the communication finishes. Thus, the worker may perform memory allocation during stealing, which accesses to the Glibc internal TLS-variable-based allocator owned by the owner process. To ensure correctness, we choose a simplified solution that patches Glibc to create two global memory allocators for each process; the more comprehensive solution by using a PiP-based user-level thread [33] will be our future work. The default TLS-variable-based allocator is still used by the process itself. When a worker calls `malloc` in the namespace of another process (i.e., the owner), we use the second global-variable-based allocator, which can be correctly referenced in our implementation. We note that the second allocator on a process can be used by at most one worker at a time; thus thread-safety is unnecessary. A worker-allocated buffer can be correctly freed by the worker, the owner, or another worker because Glibc stores the allocator base address as the metadata of each Glibc-allocated buffer.

VI. Experimental Setup

We perform all experiments on the Argonne Bebop cluster[6]. Each node contains two Intel Xeon E5-2695v4 processors with 36 cores in total, and each NUMA node attaches 64 GB DDR4 memory locally which amounts to 128 GB memory on a node. The nodes are connected via the Intel OPA interconnect. Hyper-threading is disabled on all nodes. We use PiP-aware MPICH (extended from commit bb595ca0 of MPICH main branch) as the baseline implementation which includes single-copy based optimization for intranode messages [12]. For fairness, we apply the thread-based asynchronous progress (denoted by Thread), Casper (version 1.0b2), and Daps onto the same baseline. All source codes are compiled by the gcc/gfortran compiler (version 4.8.5). The low-level libraries include Libfabric (version 1.10.1), Psm2 (version 0201), and Glibc (version 2.17). The Psm2 and Glibc are patched as described in Section V.

For the static approaches, we vary the number of dedicated cores in each experiment, denoted by Thread-N and Casper-N where N indicates the number of dedicated cores. Thus, the number of remaining user processes reduces accordingly (e.g., N=1 leaves 35 cores as user processes on a node). For multi-stage experiments in Sections VI-B1 and VI-C, we also include the dynamic Casper extension [7], denoted by Casper(D). We compare only the user-guided strategy in our experiments as it is the best result of Casper(D). Following the configuration suggested in [7], we always set 2 dedicated cores in Casper(D) to minimize computation degradation and disable the progress redirection in the communication-heavy stage. We omit the thread version with core oversubscription in all experiments because of its known high overhead on Hyper-threading disabled machine.

A. Asynchronous Progress Capability

We first design a set of microbenchmarks to ensure the asynchronous progress capability of all mechanisms in internode RMA and point-to-point communication. We employ two communication processes each is launched on a separate node. In our three RMA tests (including Get, Put, and Accumulate), process-1 performs a 200ms `sleep` to mimic user computation followed with a `barrier`, and process-0 issues two RMA operations each with contiguous 16MB data to process-1 followed with a call to `flush`. In the point-to-point test, process-

[6]https://www.lcrc.anl.gov/systems/resources/bebop/

0 issues two `isend`, whereas process-1 posts two `irecv` followed with a 200ms `sleep`. Both sides complete with a call to `waitall`. Each isend-irecv pair transfers 2MB data with the vector derived datatype (`basic_datatype`=`char`, `blk_len`=1, `stride`=64) which is internally handled by the rendezvous protocol in our baseline. For the Thread and Casper options, one CPU core is dedicated to the background thread or ghost process. For Daps, we launch one more user process on each node which is only idly waiting in a `barrier` thus it becomes a progress worker. We note that all RMA operations in Libfabric/Psm2 are emulated by internal active messages, thus benefiting from asynchronous progress.

Figure 4 measures the overall runtime time. With baseline, process-1 cannot promptly handle the incoming active message while computing outside MPI due to lack of asynchronous progress. Thus, communication from process-0 cannot be overlapped with the computation on process-1. All the asynchronous progress options, including Daps, can provide prompt progress for the incoming message on process-1, thus communication overhead can be perfectly hidden. One exception is that Casper does not support asynchronous progress in the noncontiguous point-to-point test, thus showing the same result as that of baseline in Figure 4d.

B. Static Progress v.s. Dynamic Progress

We then compare the adaptability of static and dynamic progress models by utilizing two computational kernels.

1) Custom Two-Stage BSPMM: Block sparse matrix multiplication (BSPMM) is the proxy application of computational chemistry software suite NWchem [21]. It represents the core *get-compute-update* computing pattern for a 3D sparse matrix multiplication C=$A*B$. Each process uses the one-sided `get` to obtain subblocks of the global 3D matrices A and B, and then performs `DGEMM` locally with the local subblocks, and finally updates results to the global matrix C via an `accumulate`. Inspired by the benchmarks used in [7], we extended BSPMM to contain two stages. The first stage (stage-1) is computation-intensive where each process performs 130 DGEMM tasks each with a local problem size M=N=K=1024. Each process issues 512 `get` and `accumulate` operations to all the other processes in an all-to-all fashion. Each operation carries a 50^3 3D subarray as the target datatype. The second stage (stage-2) is communication-intensive where each process performs only 50 local `DGEMM` tasks with unchanged problem size, but increases the number of issued `get` and `accumulates` to 8640. In both stages, the origin data layout of each operation is a contiguous array with 125000 (=50^3) count of double elements. To focus on the progress of the key get-accumulate operations, we omit the `fetch_and_op` based global counter update in the original BSPMM which is used for subblock scheduling.

Figure 5 reports the overall performance on 8 nodes of the Bebop cluster. It is clear that Daps outperforms all the static approaches including Casper(D). It reduces 48% overall execution time compared to the baseline. The static approaches can successfully reduce the time of stage-1 but suffer from the heavy communication cost of stage-2. The achieved best improvement compared to baseline is only 37%, delivered by Casper-8.

To further analyze the internal behaviors of each stage, we zoom in the communication portion (sum of all `gets` and `accumulates`) and the computation portion (sum of DGEMM tasks) of each stage. In stage-1 (see Figures 5b), the communication with baseline is heavily delayed due to lack of asynchronous progress. The Casper approach delivers the best improvement in the communication portion, because all active messages are promptly handled by the dedicated cores. Using dedicated cores, unfortunately, also degrades the speed of the heavy computation portion. In contrast, Daps does not degrade the user computation and also largely reduces the communication delay. We note that the communication improvement from Daps is not as good as Casper, because the heavy computation makes all processes busy at most time thus providing only a few temporary progress workers in Daps.

Figure 5c details the internal overheads of the communication-dominant stage-2. Casper has to occupy as many as 8 cores from the user processes to balance the heavy communication progress workload, largely reducing the available computing resources. Again, Daps does not require any dedicated core thus does not affect user computation. Our MPI-context-aware design can further minimize invalid stealing, ensuring efficient communication progress without any side effect.

2) Five-Point 2D Stencil: Stencil is a widely used kernel in various scientific applications for solving partial differential equations and so on [34]. A stencil kernel consists of an iterative computation stage where each iteration involves a local extremely expensive computation (5-point stencil with double elements in our case) following with a halo exchange that updates the edge data with the neighbors. We implement the halo exchange by using nonblocking `isend/irecv` with `waitall` and setup the processes in a two-dimensional Cartesian topology, following the common approach in domain applications.

Figure 6 breaks down the stencil update time (T_{comp}) and halo exchange cost (T_{comm}) by comparing baseline, Thread, Casper, and Daps on 8 Bebop nodes. The 2D matrix is with $4096*4096$ dimension size. The left and right direction isend-irecv exchanges a vector type noncontiguous data whereas the top and bottom direction carries contiguous double elements.

As shown in the baseline result, the stencil kernel is dominated by computation. Neither Casper nor Thread can help performance. This is because Casper does not support noncontiguous datatype, thus it is disabled and shown as Casper-0. Thread has to occupy only a few cores to minimize the degradation in the computation portion. But using such a few number of cores cannot efficiently handle all communication requests, thus resulting in visible communication bottleneck. As a result, the Thread option even performs worse than the baseline. Unlike the static model, Daps can dynamically detect the spare time of each process rather than static core occupancy, thus enabling performance improvement. It achieves a

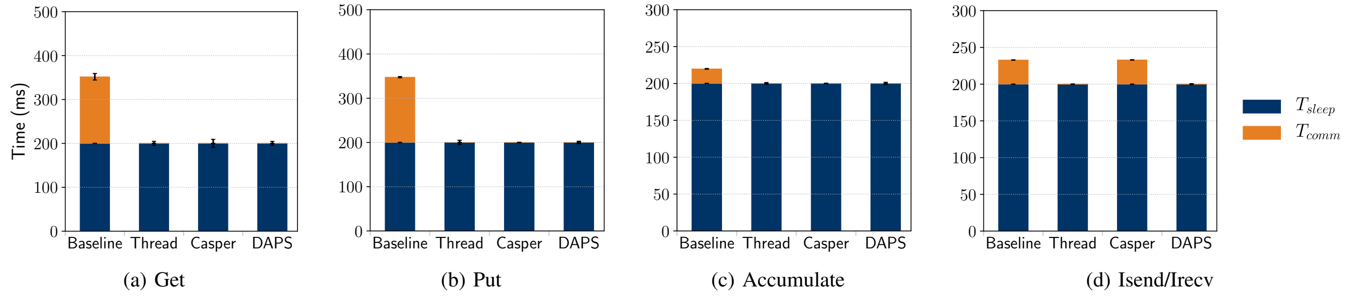


Fig. 4: Evaluating asynchronous progress capability with Get, Put, Accumulate, and Isend/Irecv with two internode process.

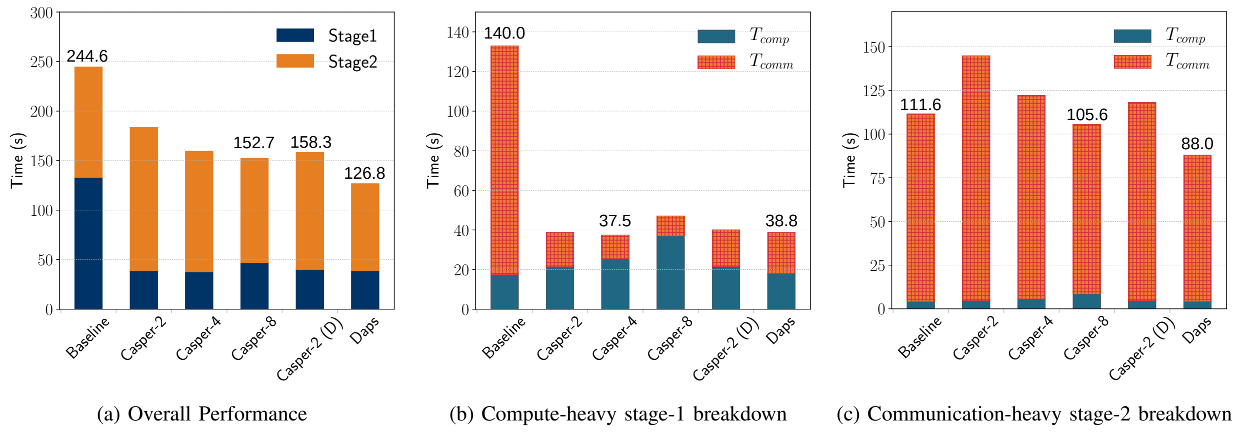


Fig. 5: Comparing Daps with static progress using the custom Two-Stage 3D BSPMM on 8 Bebop nodes (Thread results cannot be shown due to an MPICH bug, will add upon fix).

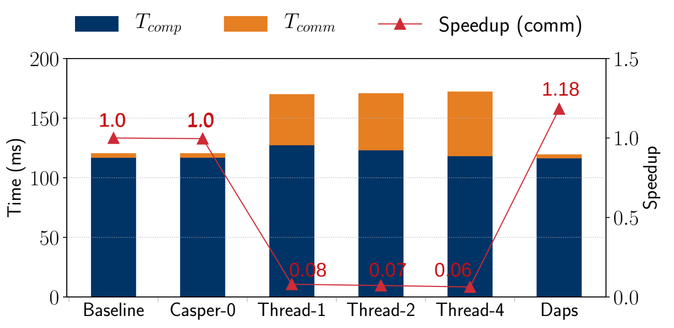


Fig. 6: Comparing Daps with static progress using five-point 2D stencil with a total problem size of $4096*4096$ on 32 Bebop nodes. Showing computation and communication breakdown and the speedup of the communication portion compared to baseline.

1.18x speedup in the communication portion compared to the baseline.

C. Scalability

By utilizing the same custom Two-Stage BSPMM as described in Section VI-B1, we evaluate the scalability of Daps in weak scaling over varying number of computing nodes. We empirically pick the best configuration for the static approaches at each test. Specifically, we use 2 dedicated cores for tests running on 2-6 nodes, use 8 dedicated cores for tests on 8 or more nodes and always use 2 dedicated cores for Casper (D). The results are reported in Figure 7. We observe that Daps can always achieve the best performance even in comparison to the best configuration of the static mechanisms in most of cases. It delivers up to 50% improvement compared to the baseline and 20% compared to Casper(D) on 16 nodes (576 processes). The performance gap between Casper and Daps consistently increases with increasing number of processes. This is because the amount of communication gradually increases, thus dynamically contributing more progress workers for Daps.

VII. Related Work

We separate the related work into two aspects: asynchronous progress and work stealing, as described below.

A. Asynchronous Progress

Jiang et al. [3] propose a thread-based asynchronous progress method on the InfiniBand clusters for one-side communication. Pritchard et al. [4] utilize CoreSpec capability provided by Cray XE Systems to dedicate one or more cores to background threads for asynchronous progress. Casper [6] designates user-specified number of ghost processes and offloads RMA and point-to-point operations to the ghosts processes for asynchronous progress; it can be overwhelmed if the offloaded workload is overly heavy. The dynamic version of Casper [15] can disable inefficient asynchronous progress during a

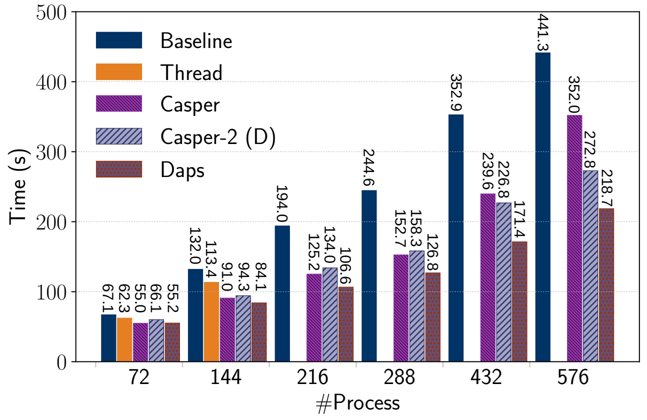


Fig. 7: Weak scaling evaluation of custom Two-Stage 3D BSPMM over up to 16 Bebop nodes. Comparing Daps with baseline and the best performing configuration of Thread and Casper (Thread with 6 or more nodes cannot be shown due to an MPICH bug, will add upon fix).

multistage execution, but the number of ghost processes cannot be dynamically changed. Vaidyanathan et al. [5] target the MPI+Threads programming model where MPI communication is offloaded to the corresponding background thread on each MPI process. Because all threads on an MPI process can share the same background thread, it does not involve the drawbacks of the thread model in traditional MPI-only programs. Ruhela et al. [35] improve the thread-based asynchronous progress for MPI-only model by reducing the frequency of progress polling from the thread. We note that all these mechanisms fall into the static asynchronous progress model, which is especially inefficient for multistage applications. Daps resolves this critical issue via a novel dynamic progress-stealing scheme.

Alternatively, PIOMan [36] is an external task scheduler for communication libraries. The task scheduler can asynchronously handle offloaded communication tasks such as polling progress on idle cores. However, it has to closely work with the thread scheduler to ensure core idleness. PAMI [37] uses communication threads for asynchronous progress on IBM Blue Gene/Q platform, but it requires special hardware and kernel support for such offloading. Sur et al. [16] exploit RDMA read and selective interrupt-based asynchronous progress on the InfiniBand cluster. On IBM systems, as described in [17]–[19], the system interrupt-based progress mechanism is studied. These approaches rely on either custom low-level threading subsystem or special hardware. Moreover, based on the case study with the NWChem application [20], a careful performance tradeoff has to be made between the thread-based and interrupt-based designs especially for non-contiguous communication.

B. Work Stealing

Traditional work stealing techniques are mainly investigated in the multithreading environment. Barghi et al. [38] utilize actor model and design a locality-aware stealing method to maximize performance on NUMA architecture. ADWS [29] designs a hierarchical work-stealing framework and performs stealing when tasks are within an activated range. LAWS [39] relies on a cache-friendly task pool and proposes triple-level work-stealing algorithm for maximal cache reuse. In addition, many other methods [28], [40], [41], also focus on NUMA architecture and propose the locality-aware work-stealing algorithm; the critical idea ineach case is to increase the local stealing chances in order to better utilize NUMA property and amortize the effect of remote task stealing. Unlike these works, Daps is designed to steal across processes, thus introducing unique challenges. Moreover, our progress-overhead-aware and MPI-context-aware optimizations are specific to the MPI environment.

Our previous work proposed interprocess work stealing to parallelize the data copy or computing tasks in MPI [11]. It steals only simple copy or predefined reduce computing tasks and still relies on the owner to process the active message. In contrast, Daps directly steals the progress task to provide asynchronous progress. Stealing complex tasks is nontrivial, as already shown in the paper. More important, Dapa showcases a more flexible work-stealing scheme in the interprocess space similar to that in the multithreading environment.

VIII. Conclusion

Lack of asynchronous progress is a long-lasting problem in MPI. Ideally, all data transmission can be offloaded to the network hardware so that the CPU resources can be dedicated to user computation. Today's HPC interconnects, unfortunately, still cannot handle many complex data transfers that are required by MPI. Consequently, software-level asynchronous progress has to be involved. Traditional software-level asynchronous progress mechanisms have to statically configure progress resources (i.e., CPU cores), forcing the user to perform repeated experiments to fine-tune the configuration for different applications. Such a method may even perform poorly for multiple-stage applications where each stage often forms a different communication and computation pattern. In this paper, we presented Daps, a novel dynamic asynchronous progress model based on interprocess work stealing. We formulated a detailed guideline for the prerequisites of a successful work stealing in the multiprocess space and utilized the PiP weak multiprocess model to support flexible data and code sharing as well as shared code execution. The Daps algorithm is highly optimized by leveraging MPI internal knowledge We also analyzed and addressed special implementation challenges that occurred when a stealing interacts with low-level network drivers and TLS-involved libraries. The evaluation was performed on an Intel OPA cluster. Compared with the state-of-the-art mechanisms, Daps achieves up to 20% improvement in the two-stage BSPMM kernel and a 1.18x speedup in the five-point 2D stencil.

Acknowledgment

This material was based upon work supported by the U.S. Dept. of Energy, Office of Science, Advanced Scientific Computing Research (SC-21), under contract DE-AC02-06CH11357. The experimental resource for this paper was provided by the Laboratory Computing Resource Center on the Bebop cluster at Argonne National Laboratory.

References

[1] W. D. Gropp, "Learning from the success of MPI," in *International Conference on High-Performance Computing*. Springer, 2001, pp. 81–92.

[2] J. Liu, J. Wu, and D. K. Panda, "High performance RDMA-based MPI implementation over infiniband," *International Journal of Parallel Programming*, vol. 32, no. 3, pp. 167–198, 2004.

[3] W. Jiang, J. Liu, H.-W. Jin, D. K. Panda, D. Buntinas, R. Thakur, and W. D. Gropp, "Efficient implementation of MPI-2 passive one-sided communication on infiniband clusters," in *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*. Springer, 2004, pp. 68–76.

[4] H. Pritchard, D. Roweth, D. Henseler, and P. Cassella, "Leveraging the cray linux environment core specialization feature to realize MPI asynchronous progress on cray XE systems," in *Proceedings of the Cray User Group Conference*, vol. 79, 2012, p. 130.

[5] K. Vaidyanathan, D. D. Kalamkar, K. Pamnany, J. R. Hammond, P. Balaji, D. Das, J. Park, and B. Joó, "Improving concurrency and asynchrony in multithreaded MPI applications using software offloading," in *SC'15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2015, pp. 1–12.

[6] M. Si, A. J. Pena, J. Hammond, P. Balaji, M. Takagi, and Y. Ishikawa, "Casper: An asynchronous progress model for MPI RMA on many-core architectures," in *2015 IEEE International Parallel and Distributed Processing Symposium*. IEEE, 2015, pp. 665–676.

[7] M. Si, A. J. Pena, J. Hammond, P. Balaji, M. Takagi, and Y. Ishikawa, "Dynamic adaptable asynchronous progress model for MPI RMA multiphase applications," vol. 29, no. 9. IEEE, 2018, pp. 1975–1989.

[8] U. A. Acar, A. Chargu'eraud, and M. Rainey, "Scheduling Parallel Programs by Work Stealing with Private Deques," in *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*, 2013, pp. 219–228.

[9] B. Sheridan and J. T. Fineman, "A Case for Distributed Work-Stealing in Regular Applications," in *Proceedings of the 6th ACM SIGPLAN Workshop on X10*, 2016, pp. 32–33.

[10] A. Tzannes, G. C. Caragea, R. Barua, and U. Vishkin, "Lazy Binary-Splitting: A Run-Time Adaptive Work-Stealing Scheduler," in *ACM Sigplan Notices*, vol. 45, no. 5. ACM, 2010, pp. 179–190.

[11] K. Ouyang, M. Si, A. Hori, Z. Chen, and P. Balaji, "CAB-MPI: Exploring interprocess work-stealing towards balanced MPI communication," in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2020, pp. 1–15.

[12] A. Hori, M. Si, B. Gerofi, M. Takagi, J. Dayal, P. Balaji, and Y. Ishikawa, "Process-in-process: Techniques for practical address-space sharing," in *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*, 2018, pp. 131–143.

[13] R. Brightwell, R. Riesen, and K. D. Underwood, "Analyzing the impact of overlap, offload, and independent progress for message passing interface applications," *The International Journal of High Performance Computing Applications*, vol. 19, no. 2, pp. 103–117, 2005.

[14] T. Hoefler and A. Lumsdaine, "Message progression in parallel computing-to thread or not to thread?" in *2008 IEEE International Conference on Cluster Computing*. IEEE, 2008, pp. 213–222.

[15] M. Si and P. Balaji, "Process-based asynchronous progress model for MPI point-to-point communication," in *2017 IEEE 19th International Conference on High Performance Computing and Communications; IEEE 15th International Conference on Smart City; IEEE 3rd International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*. IEEE, 2017, pp. 206–214.

[16] S. Sur, H.-W. Jin, L. Chai, and D. K. Panda, "RDMA read based rendezvous protocol for MPI over infiniband: Design alternatives and benefits," in *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, 2006, pp. 32–39.

[17] S. Kumar, G. Dozsa, G. Almasi, P. Heidelberger, D. Chen, M. E. Giampapa, M. Blocksome, A. Faraj, J. Parker, J. Ratterman *et al.*, "The deep computing messaging framework: Generalized scalable message passing on the blue gene/P supercomputer," in *Proceedings of the 22nd annual international conference on Supercomputing*, 2008, pp. 94–103.

[18] M. Krishnan, J. Nieplocha, M. Blocksome, and B. Smith, "Evaluation of remote memory access communication on the IBM blue gene/P supercomputer," in *2008 International Conference on Parallel Processing-Workshops*. IEEE, 2008, pp. 109–115.

[19] S. Kumar, Y. Sun, and L. V. Kalé, "Acceleration of an asynchronous message driven programming paradigm on IBM blue gene/Q," in *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*. IEEE, 2013, pp. 689–699.

[20] J. R. Hammond, S. Krishnamoorthy, S. Shende, N. A. Romero, and A. D. Malony, "Performance characterization of global address space applications: a case study with NWChem," *Concurrency and Computation: Practice and Experience*, vol. 24, no. 2, pp. 135–154, 2012.

[21] M. Valiev, E. J. Bylaska, N. Govind, K. Kowalski, T. P. Straatsma, H. J. V. Dam, D. Wang, J. Nieplocha, E. Apra, T. L. Windus *et al.*, "NWChem: A comprehensive and scalable open-source solution for large scale molecular simulations," *Computer Physics Communications*, vol. 181, no. 9, pp. 1477–1489, 2010.

[22] B. Goglin and S. Moreaud, "KNEM: A generic and scalable kernel-assisted intra-node MPI communication framework," *Journal of Parallel and Distributed Computing*, vol. 73, no. 2, pp. 176–188, 2013.

[23] T. Ma, G. Bosilca, A. Bouteiller, B. Goglin, J. M. Squyres, and J. J. Dongarra, "Kernel assisted collective intra-node mpi communication among multi-core and many-core cpus," in *2011 International Conference on Parallel Processing*. IEEE, 2011, pp. 532–541.

[24] M. Kerrisk. (2008) Overview of POSIX Shared Memory. [Online]. Available: http://man7.org/linux/man-pages/man7/shm_overview.7.html

[25] M. Kerrisk. (2020) Linux Programmer's Manual. [Online]. Available: https://man7.org/linux/man-pages/man2/process_vm_readv.2.html

[26] B. Goglin and S. Moreaud, "KNEM: A generic and scalable kernel-assisted intra-node MPI communication framework," *Journal of Parallel and Distributed Computing*, vol. 73, no. 2, pp. 176–188, 2013.

[27] N. Hjelm, P. Shamis, and J. Squyres, "XPMEM Linux Kernel Module," 2018. [Online]. Available: https://github.com/hjelmn/xpmem

[28] Q. Chen and M. Guo, "Contention and locality-aware work-stealing for iterative applications in multi-socket computers," *IEEE Transactions on Computers*, vol. 67, no. 6, pp. 784–798, 2017.

[29] S. Shiina and K. Taura, "Almost deterministic work stealing," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2019, pp. 1–16.

[30] OpenFabrics Interfaces (OFI). [Online]. Available: https://ofiwg.github.io/libfabric/

[31] Intel Corporation. OPA-PSM2 Github repository. [Online]. Available: https://github.com/cornelisnetworks/opa-psm2

[32] M. S. Birrittella, M. Debbage, R. Huggahalli, J. Kunz, T. Lovett, T. Rimmer, K. D. Underwood, and R. C. Zak, "Intel Omni-path Architecture: Enabling Scalable, High Performance Fabrics," in *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*, 2015, pp. 1–9.

[33] A. Hori, B. Gerofi, and Y. Ishikawa, "An implementation of user-level processes using address space sharing," in *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2020, pp. 976–984.

[34] J. M. Cecilia, J. M. García, and M. Ujaldón, "CUDA 2D stencil computations for the jacobi method," in *International Workshop on Applied Parallel Computing*. Springer, 2010, pp. 173–183.

[35] A. Ruhela, H. Subramoni, S. Chakraborty, M. Bayatpour, P. Kousha, and D. K. Panda, "Efficient asynchronous communication progress for MPI without dedicated resources," in *Proceedings of the 25th European MPI Users' Group Meeting*, 2018, pp. 1–11.

[36] F. Trahay and A. Denis, "A scalable and generic task scheduling system for communication libraries," in *2009 IEEE International Conference on Cluster Computing and Workshops*. IEEE, 2009, pp. 1–8.

[37] S. Kumar, A. R. Mamidala, D. A. Faraj, B. Smith, M. Blocksome, B. Cernohous, D. Miller, J. Parker, J. Ratterman, P. Heidelberger *et al.*, "PAMI: A parallel active message interface for the blue gene/Q supercomputer," in *2012 IEEE 26th International Parallel and Distributed Processing Symposium*. IEEE, 2012, pp. 763–773.

[38] S. Barghi and M. Karsten, "Work-Stealing, Locality-Aware Actor Scheduling," in *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2018, pp. 484–494.

[39] Q. Chen, M. Guo, and H. Guan, "LAWS: Locality-Aware Work-Stealing for Multi-Socket Multi-Core Architectures," in *Proceedings of the 28th ACM international conference on Supercomputing*. ACM, 2014, pp. 3–12.

[40] J. Deters, J. Wu, Y. Xu, and I.-T. A. Lee, “A NUMA-aware provably-efficient task-parallel platform based on the work-first principle,” in *2018 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2018, pp. 59–70.

[41] A. Drebes, A. Pop, K. Heydemann, N. Drach, and A. Cohen, “NUMA-aware scheduling and memory allocation for data-flow task-parallel applications,” in *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2016, pp. 1–2.