CAB-MPI: Exploring Interprocess Work-Stealing towards Balanced MPI Communication

Kaiming Ouyang*, Min Si†, Atsushi Hori‡, Zizhong Chen* and Pavan Balaji†
*Computer Science and Engineering
University of California, Riverside, USA
Email: kouya001@ucr.edu, chen@cs.ucr.edu
†Mathematics and Computer Science
Argonne National Laboratory, USA
Email: msi@anl.gov, balaji@anl.gov
‡RIKEN Center for Computational Science
RIKEN, Japan
Email: ahori@riken.jp

***Abstract*—Load balance is essential for high-performance applications. Unbalanced communication can cause severe performance degradation, even in computation-balanced BSP applications. Designing communication-balanced applications is challenging, however, because of the diverse communication implementations at the underlying runtime system. In this paper, we address this challenge through an interprocess work-stealing scheme based on process-memory-sharing techniques. We present CAB-MPI, an MPI implementation that can identify idle processes inside MPI and use these idle resources to dynamically balance communication workload on the node. We design throughput-optimized strategies to ensure efficient stealing of the data movement tasks. We demonstrate the benefit of work stealing through several internal processes in MPI, including intranode data transfer, pack/unpack for noncontiguous communication, and computation in one-sided accumulates. The implementation is evaluated through a set of microbenchmarks and proxy applications on Intel Xeon and Xeon Phi platforms.**

***Index Terms*—Work stealing, MPI, load balance, communication**

I. Introduction

MPI remains the dominant parallel programming model in high-performance computing (HPC) applications. A primary goal of MPI applications is to efficiently execute on large-scale systems while maintaining low communication overhead. The communication overhead is not only caused by the data transfer required by application algorithms but may be also caused by the synchronization between processes that are handling unbalanced workload. That is, the process that has finished its local work has to wait for the other busy processes (e.g., the one that handles heavier work) to complete at a synchronizing point before moving to the next step or iteration in the application. Such an issue not only degrades performance but also causes underutilization of hardware resources because the underlying cores are completely idle during waiting.

Application developers have put significant effort into balancing computational workload [1]–[4]. However, the balance of communication workload (e.g., data transfer and internal processing in MPI) is often not well optimized, resulting in considerable performance degradation. For instance, the stencil is a widely studied application pattern and is considered to be regular and balanced (i.e., bulk synchronous parallelism). As we show in Figure 1, however, a well-balanced seven-point three-dimensional stencil program can still present up to 45% idle time (i.e., the period idly waiting inside MPI) on some processes, resulting in 18% degradation in the overall performance (based on the estimated "ideal time" with balanced communication.[1]) Indeed, such idleness is caused mainly by the imbalance of communication.

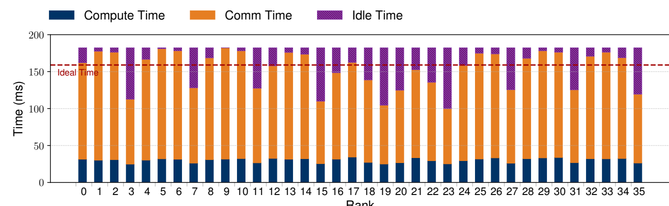


Fig. 1: Unbalanced communication in 3D 7-point stencil on 4 Broadwell nodes (Intel Xeon E5-2695v4 CPU, 36 processes per node). The experiment adopted the miniGhost stencil program [5] with parameters nx=ny=nz=50,$nvar$=100 and process grid 4×6×6. The time is measured for processes on the first node.

Pursuing evenly distributed communication at the application level is impractical mainly because the user of MPI cannot estimate the amount of work involved inside each MPI call. For instance, intranode communication and internode communication are usually implemented differently. Consequently, the required workloads are different even if the message size is the same. Within noncontiguous data transfer, depending on the data layout the workloads can be significantly different (e.g., the data transfer of the X-Z plane vs. that of the Y-Z plane in a 3D halo exchange). Moreover, even with the same type of data transfer, the amount of work might vary depending on the location of the communicating processes (e.g., cross-

[1]We obtained the ideal time by averaging the sum of compute time and communication time on all 36 processes on the node.

The submitted manuscript has been created by UChicago Argonne, LLC, Operator of Argonne National Laboratory ("Argonne"). Argonne, a U.S. Department of Energy Office of Science laboratory, is operated under Contract No. DE-AC02-06CH11357. The U.S. Government retains for itself, and others acting on its behalf, a paid-up nonexclusive, irrevocable worldwide license in said article to reproduce, prepare derivative works, distribute copies to the public, and perform publicly and display publicly, by or on behalf of the Government. The Department of Energy will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan. http://energy.gov/downloads/doe-public-access-plan.

NUMA data transfer usually takes longer than that inside a NUMA node).

To address this challenging issue, we believe that a runtime-level solution is essential. In this paper, we present CAB-MPI, a communication-auto-balance MPI implementation that internally balances various communication workloads in MPI. CAB-MPI is based on the concept of *interprocess work stealing* that utilizes idly waiting processes inside the MPI library to "steal" communication tasks from the other busy processes located on the same node, consequently achieving communication balance.

The work-stealing approach has been broadly investigated in multithreading programming [6]–[16]. Such an approach requires flexible data sharing because the worker (i.e., the one that steals work) has to access arbitrary data associated with the stolen task. Such a requirement is naturally met in multithreaded programs since the worker thread and the victim thread share the same virtual address space. In process-based MPI programs, however, a special memory-sharing technique has to be used because accessing data owned by a different process is prohibited by the operating system. As the prerequisite of the proposed interprocess work stealing, we analyze primary process-memory-sharing techniques that are available in the HPC community. We then present the implementation of CAB-MPI based on the process-in-process address-sharing technique [17].

Unlike traditional work-stealing solutions that are often designed for computational workloads, the work stealing in MPI specializes in communication. The dominant data-movement-centric workloads make the stealing tasks memory bandwidth bound. A performance-efficient work-stealing strategy must take into account the bandwidth limitation especially when cross-memory-domain data access is involved. More important, stealing a communication task has to involve multiple processes (e.g., sender, receiver, and worker in MPI point-to-point communication) and data buffers (i.e., including source, destination, and any intermediate buffers). Special locality-aware strategies must be designed for such a multiprocess multibuffer scenario. These challenges make our work-stealing design completely different from existing work. To the best of our knowledge, CAB-MPI is the first work that systemically explores interprocess work stealing for MPI-like communication workloads.

We demonstrate the performance benefit of the proposed approach in several MPI internal processes, including intranode data transfer, pack/unpack in noncontiguous data transfer, and reduce operations in the RMA accumulate communication. We also present a thorough experimental evaluation and analysis on Intel Xeon Broadwell and Knights Landing (KNL) platforms using a variety of microbenchmarks and proxy applications.

II. Shared-Memory Technique Analysis

Interprocess work stealing requires data sharing between processes mainly for two kinds of data. The first is a *shared data structure* to manage the available tasks on each process such as the queue structures used in CAB-MPI; the second kind is the *user data* associated with each communication task, which is usually managed by the user program (e.g., the source and destination buffers specified to the MPI send/receive calls). Unlike threads, processes cannot arbitrarily access the data owned by another process, because of limitations by the operating system (OS). Several process-memory-sharing techniques are used in the HPC community, but not all of them provide sufficient support for the required data sharing. In this section, we give a brief overview of each technique and discuss its suitability for use in CAB-MPI.

POSIX shared memory [18] allows two processes to collectively allocate a shared-memory segment. However, global variables or preallocated buffers (e.g., the user data associated with an MPI call) cannot be shared.

Cross-Memory-Attach (CMA) [19] and KNEM [20] are two kernel-assisted direct copy techniques. A process can directly read or write a buffer owned by the other process by using the system call provided by CMA or KNEM. To make a third process (e.g., the worker process in work stealing) perform the copy for two processes, however, it has to copy the data through a temporary buffer in its own memory space beforehand. Each data copy has to go through the kernel, making these approaches expensive.

XPMEM [21] is a Linux kernel module supporting cross-process memory mapping. A process can attach a remote memory segment to its local address space through an XPMEM system call and cache the segment handle for reuse. The data copy is performed completely in user space. For every newly used buffer on a process, however, the worker process still has to pay an expensive cost to attach the segment. Such a limitation can result in up to $O(p^2)$ attach overhead, where p is the number of processes on a node.

Process-in-process (PiP) [17] is a user-level address-space-sharing technique based on position-independent executables (PIE) and the `dlmopen()` Glibc function. The PiP environment allows every execution unit (called a PiP task) to behave as a normal OS process (i.e., each task owns a privatized variable set and can execute a different program) but share the same virtual address space with others located on the same node. Consequently, it enables arbitrary interprocess data access without involving additional overhead.

The thread-based MPI implementations allow complete data sharing across MPI processes. For instance, MPC is a thread-based language-processing system designed for hybrid MPI and OpenMP programming [22]. The MPC runtime creates threads running as MPI processes so that intranode data transfer can be highly optimized. AMPI over Charm++ [23], [24] implements MPI ranks over user-level threads in order to migrate ranks over different physical cores for dynamical workload balance. Both implementations, however, indicate several shortcomings of the thread-based model, such as inconvenient global variable privatization and lack of support for executing multiple programs.

In summary, PiP is the most suitable memory-sharing technique to support interprocess work stealing in MPI. Some

other approaches (i.e., POSIX shared memory, XPMEM, or the thread-based model), however, are also feasible with limitations in the user program. For instance, if the user agrees to allocate user data only from shared memory, POSIX shared memory would be sufficient for interprocess stealing.

In the following sections, we use the PiP-aware MPI [17] as the baseline implementation. To be specific, we extended the MPICH implementation of MPI (commit 8cccb4c5 from the master branch at https://github.com/pmodels/mpich). We modified the Hydra process launching of MPICH to spawn MPI processes as PiP tasks. All intranode data transfer routines were optimized following the *1-copy* protocol in the baseline implementation. Work stealing applies only to communication with medium-sized and large data; thus, discussion regarding small data communication is omitted in this paper.

III. Design and Implementation

In this section, we describe the design of the proposed work-stealing mechanism in CAB-MPI.

A. Basic Semantics Definition

The core concept of CAB-MPI is to employ idle MPI processes to steal the communication tasks from the other busy processes in order to balance workload. We call such an idle process a valid "worker." Below we define the semantics of worker, task, and their locality.

1) Worker Definition: We define that *a process becomes a valid worker of the work-stealing mechanism when it is idly waiting at an MPI blocking call*. A simple example is the `MPI_Barrier` call. Once a process arrives at the barrier, it has to idly wait until the last process in the communicator also arrives at the call. Therefore, the waiting process becomes a valid worker. When a process makes a call to `MPI_Recv`, for example, it becomes a valid worker until a matching message arrives. In sending calls, the process can also be a valid worker if it is waiting inside MPI for available communication resources or for response from the other process (e.g., in a rendezvous protocol). For nonblocking calls such as `MPI_Isend` and `MPI_Put`, the process returns immediately after initializing the sending; thus it cannot be a worker. However, it becomes a valid worker once it arrives at the blocking synchronization calls such as `MPI_Wait` and `MPI_Win_flush`. For nonblocking synchronization calls such as `MPI_Test`, we consider that the user wants to compute after the call; thus we do not make the process be a worker.

The worker status of a process is *time-specific*. For instance, a worker may become invalid after finishing a stealing task if it detects that its waiting condition is met (e.g., the incoming data has arrived). In MPICH-derived MPI implementations, this situation usually occurs when the process polls the progress engine.

2) Task Definition: MPI provides many types of routines to which work stealing can bring performance benefits. We summarize them in two categories: data-movement-centric routines and compute-centric routines. The former category includes any intranode communication calls such as `MPI_Send|Recv` and one-sided operations and any internal data movements for internode communication (e.g., data pack/unpack for noncontiguous data). The latter category refers to the reduce operation involved in some communication calls such as `MPI_Accumulate` and `MPI_Reduce`. We define the *the stealing task as moving or computing a certain amount of data from the source to the destination buffer*.

We note that for intranode data movement or compute routines, the buffers are usually the same as those of the user-specified buffers. For internode routines, however, either the source or the destination buffer is an internal buffer maintained by MPI. We give a detailed description in Section III-D.

Ownership determination. Task ownership identifies the locality of tasks and workers, which is a key performance factor in work stealing. Unlike traditional work-stealing scenarios, a stealing task in CAB-MPI involves at least a pair of processes. Thus, special rules must be designed to determine the ownership of a task. We define two common rules.

Rule 1. A task belongs with the involved process that will likely consume the result data.
Rule 2. If it is unknown what process will use the result data, the task belongs with the process that actually performs the data movement or computation before applying work stealing.

Based on these two rules, we describe the task ownership for each MPI communication mode. For intranode send/receive, the receiver process owns the involved data movement task(s) because it will likely use the received data (e.g., using it in a user computation) (*Rule 1* is applied). For intranode one-sided operations, however, the transferred data does not have a specific "consumer." Thus, the origin process that performs the work in the *1-copy* protocol owns the involved data movement or computing task(s) (*Rule 2* is applied). An internode operation may involve separate tasks on each node (e.g., an active-message-based noncontiguous `MPI_Accumulate` produces packing task(s) on the origin node and computing task(s) on the target node). In such a case, each task is owned by the operating process on each node (*Rule 2*). Collective operations are implemented based on active messages by default. Thus, the task ownership is similar to an internode operation.[2]

3) Locality Definition: Cross-memory domain (e.g., NUMA) work stealing may degrade performance especially for data movement tasks. The locality of stealing tasks and workers is an essential property for performance consideration. The granularity of locality varies on different hardware architecture and can be hierarchical. In this paper we consider only a single-level granularity for simplicity (NUMA node). To be specific, we define that *the locality of a task belongs to the NUMA node to which the owner process is bound*. Moreover, we use the term *local stealing* to describe the case where a worker steals a task from the local NUMA node; otherwise we describe it as *remote stealing*.

[2]Shared-memory-based collective optimization [25], [26] is orthogonal to this work; we leave work stealing for such tasks as future work.

B. Framework Design

We present our basic work-stealing framework in Figure 2. We separate the procedure into a task allocation flow from the view of the task owner and a work-stealing flow from the view of a worker. At the task allocation flow, the owner logically chunks the buffers and creates a separate task for each chunk. The task descriptor contains the information of buffer offset, chunk size, reduce operation (`MPI_REPLACE` is set for data movement tasks), and datatypes. A completion flag is used to determine whether the worker has finished the task. Each process maintains two queue structures: a first-in, first-out *task queue* shared with all potential workers and a private *track queue* that is used to track any completed tasks and reclaim the associated resources. The owner enqueues each created task into both queues (atomicity is required only for the shared task queue.[3]) At the work-stealing flow, a worker follows the stealing strategies (see Section III-C) to choose the victim process. The worker dequeues a task from the victim's task queue and then processes it. After the task is complete, the worker marks the completion flag in the task descriptor so that the owner can notice the completion when traversing its private track queue and can clean up resources.

Ensuring MPI semantics correctness. For send/receive communication the stealing tasks are created only after message matching. Thus, the message ordering is not broken by work stealing. For one-sided accumulate operations, the owner process creates the tasks for an operation only after obtaining permission to update the window (e.g., through a mutex lock in MPICH) and always waits for the completion of all tasks before processing the next operation. Hence, the required atomicity and ordering are ensured.

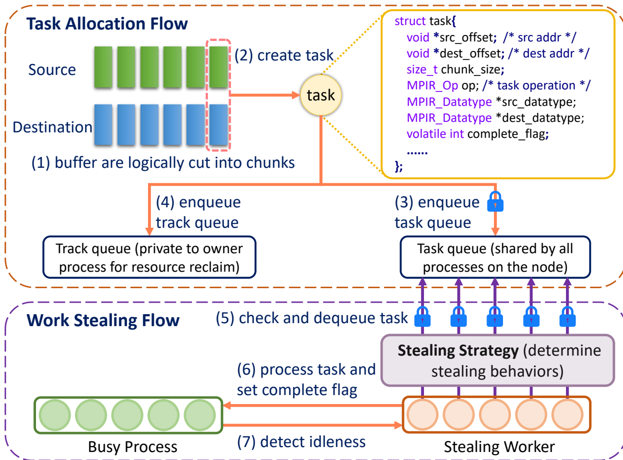


Fig. 2: High-level queue-based work-stealing framework.

C. Work-Stealing Strategies

In this section, we explore the strategies for victim selection through three work-stealing strategies.

[3]Our current implementation simply uses a lock-based single-producer-multiple-consumer queue. However, the implementation can be further optimized based on lock-free algorithms.

1) Localized Work Stealing: The worker can perform only local stealing based on the fact that intra-NUMA data access is always faster than that across NUMA nodes. Therefore, the data is always kept in the local cache, and the stealing never causes extra cross-NUMA data access. The selection of a victim from the local NUMA is based on a random protocol that is simple yet sufficient. If the victim's task queue is not empty, the worker dequeues a task and handles it; otherwise, the worker simply exits. We note that each worker checks only one victim at a time in order to keep the stealing routine lightweight. This approach allows the worker to frequently check whether its waiting condition is met so that it can switch back to its original work. If the worker status is still valid, it can re-enter the stealing routine again.

2) Mixed Work Stealing: Mixed work stealing extends the localized work-stealing version. If a worker cannot find any task from the selected local victim, it then proceeds to remote stealing following the same random victim selection method. All remote victims are maintained in a single pool for random selection even if the architecture contains multiple NUMA nodes (e.g., in KNL SNC4 mode). Similar to local stealing, each worker selects a remote victim only once, to keep the trial lightweight.

Discussion: localized vs. mixed stealing. For memory-bound tasks that are dominated by memory operations (e.g., `memcpy`), the performance is determined mainly by the achieved data access throughput. Therefore, localized work stealing should be the best approach if the number of local workers is sufficient. When the local workers are not enough to saturate the memory bandwidth, however, allowing remote stealing can improve memory throughput. Hence, mixed stealing works better in such a case. Unfortunately, none of the strategies can efficiently serve all use cases. Therefore, we further explore the third strategy based on throughput awareness.

3) Throughput-Aware Work Stealing: Throughput-aware work stealing is based on the notion that when the memory bandwidth of a NUMA node is not saturated, increasing remote stealing can improve overall throughput. When local stealing is sufficient to saturate the bandwidth, however, we need to avoid remote stealing in order to ensure high local-NUMA throughput. To demonstrate such a tradeoff, we use a simple `memcpy` microbenchmark to mimic the data movement tasks in MPI. Each process allocates the source and the destination buffers from the same NUMA node and performs `memcpy` with 64 KB of data 1,000 times. We adjust the number of processes that simultaneously perform the copy on every NUMA node and report the overall throughput on the node by summing the local throughput achieved by each process. The experimental platform consists of two NUMA nodes. We call processes on the first NUMA node local processes, and we call the ones on the other NUMA nodes remote processes. As shown in Figure 3, if we vary the number of remote processes for each fixed number of local processes, throughput improves only when the number of local processes is less than 4. When more local processes are performing the copy, adding remote processes significantly degrades overall throughput. Clearly,

we can divide the trend into a *bandwidth-unsaturated range* and a *bandwidth-saturated range* as indicated in the graph.

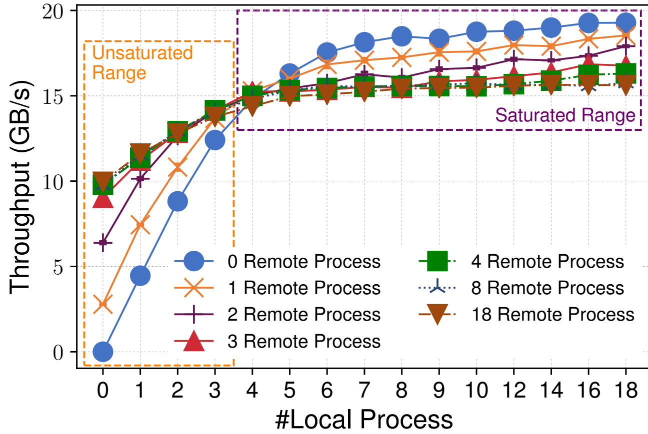


Fig. 3: Memcpy throughput with variable number of local and remote processes on a Broadwell node (two NUMA nodes each with 18 cores). The results are averaged from ten runs, and the error is less than 4%.

Based on the throughput analysis, we design the throughput-aware work-stealing strategy. The local stealing phase remains unchanged. When local stealing fails, it then tries to perform remote stealing. Unlike the mixed stealing strategy, it first checks the bandwidth status of the NUMA node associated with the selected remote victim. The worker steals a task from the victim only when the NUMA bandwidth is not saturated.

Precisely quantifying the bandwidth usage of a NUMA node is difficult because the processes perform a variety of tasks during runtime. Some of the tasks are generated by MPI while some others are from the user program; some tasks are memory-bound while some others are more compute-bound. Therefore, we make a conservative estimation based on the number of processes that are "possibly active" on that NUMA node. That is, we count a process as guaranteed idle only when it is idly waiting inside MPI; otherwise, we assume it is active and contributes to the bandwidth usage. We denote the number of active processes by n_{active}. We define the threshold of saturated local workers based on the results from Figure 3 (denoted by $N_{saturate}$. Value is 4 on our platform). Therefore, a worker checks whether the remote NUMA's bandwidth is saturated by comparing $n_{active} \geq N_{saturate}$. We emphasize that this method is conservative because we assume all active processes are performing memcpy-like tasks. However, such a method allows us to avoid any performance degradation that may be caused by remote stealing.

To keep the preferred local stealing fast, each process updates only a local flag. The flag is 1 by default. It becomes 0 only when the process becomes a valid worker and does not handle any stealing task. The worker that performs remote stealing checks the flag on each process on the remote NUMA to count n_{active}.

D. Work-Stealing Showcase

We exploit three internal aspects in CAB-MPI to showcase the proposed working-stealing method: intranode contiguous data transfer, noncontiguous data packing/unpacking, and the reduce operation in one-sided accumulate. We describe the task creation for each aspect. The consequent work stealing follows the generic framework and strategies as described in the preceding subsections.

1) Intranode Data Transfer: In the baseline PiP-aware MPI, the receiver process directly copies data from the sender process on the same node after exchanging the buffer addresses at handshake. As the simplest task type, we logically chunk such data copy into multiple chunks and expose each chunk as a stealing task.

2) Noncontiguous Data Packing: To transfer noncontiguous data, MPICH internally triggers the pack/unpack routines. For an intranode message, if both the source and the destination buffers are noncontiguous, an internal contiguous buffer is used; for internode messages, the data is first packed into an internal buffer on the sender process for network transfer and then unpacked into the destination buffer once it arrive on the receiving side. A similar approach is used for both send/receive and one-sided operations. We logically chunk the pack/unpack task and expose each as a stealing task.

3) Reduce Operation: Several MPI functions carry a reduce operation (e.g., `MPI_SUM`, `MPI_PROD`). Here we optimize the `MPI_Accumulate` function as an example. In the baseline implementation, the computation is performed by the origin process in an intranode accumulate through PiP's shared-memory environment; for any internode accumulate, it is implemented as an active message (i.e., the target process receives the data and then computes and updates the window). In either case, the computation is chunked and posted as stealing tasks. Each task always handles a separate data range. We note that a similar optimization can be easily applied to other MPI functions involving the reduce operation, such as `MPI_Reduce`. We omit its description because of space limitation.

E. Other Optimizations

We propose three techniques to optimize the showcases.

1) Reversed Task Enqueue: The receiver process commonly will access the data after communication. For large data transfer (e.g., larger than the last-level cache (LLC) size), the data at the low address of the destination buffer may be flushed out from cache when the transfer completes if the data movement starts from the low address. If the user later also loads data from the low address, extra cache misses can occur, and thus the post-communication access becomes slow. To reduce such cache misses, we propose to reverse the order of task enqueue. Specifically, we define three access patterns: from low to high address (`lo-to-hi`), from high to low address (`hi-to-lo`), and random access (`random`). We allow the user to provide a hint to MPI to indicate the access pattern with the info key `post_comm_access`. The info value is `random` by default. If `lo-to-hi` is specified, we post tasks

in reverse order; otherwise we post tasks from low to high address. We note that the stealing tasks might be performed out of order by different workers. Thus, this approach aims onlyto get a higher chance to keep data in cache.

2) Noncontiguous Task Bundle: In noncontiguous data transfer, an internal contiguous buffer is used together with the pack/unpack routines. On modern architectures [27] data stored in the internal buffer is likely cached when performing pack and then reused at unpack. If we create stealing tasks separately for pack and unpack routines, the tasks might be executed by different workers, resulting in inefficient use of cache. Consequently, we propose to combine the pack and the unpack work into a single task. To be specific, each stealing task carries data from a chunk of the source buffer to the corresponding chunk in the destination buffer. The internal buffer is allocated by each worker. In this way, the data packed into the internal buffer can be reused. We note that the resulting benefit is highly related to the layout of the source datatype. That is, if the layout contains a long stride between data elements, cache waste can be caused by inappropriate prefetching. In Section IV we demonstrate such a trend. Nevertheless, the proposed optimization never causes performance degradation compared with the original approach.

3) On-Demand Chunking: A small data chunk size may benefit performance because it can produce sufficient tasks for parallelism; however, overly creating tasks also causes more manipulation overhead, such as the costs required by task creation, enqueue, and dequeue. Therefore, we propose to adjust the chunk size "on demand" in order to maintain a reasonable degree of decomposition. For instance, for contiguous data transfer we set three message ranges and choose a different chunk size for each range based on profiling results. The appropriate value for the range thresholds and the chunk sizes should be tuned for different platforms. Our platform sets a 16 KB chunk size for small messages ($<$ 96 KB), a 32 KB chunk size for medium messages (96 KB $\leq$ size $<$ 512 KB), and a 64 KB chunk size for large messages ($\geq$ 512 KB).

IV. Experimental Configuration

The experiments were executed on a Broadwell cluster and a KNL cluster. The Broadwell cluster consists of 664 nodes. Each node contains two Intel Xeon E5-2695v4 processors with 36 cores in total. Its memory is 128 GB of DDR4 RAM divided into two NUMA nodes. The L1, L2, and L3 cache sizes are 32 KB, 256 KB, and 45 MB, respectively. The node of the KNL cluster uses a 64-core Intel Xeon Phi 7230 processor with 32 KB L1 cache, 1 MB L2 cache shared per 2 cores, 16 GB of MCDRAM, and 96 GB of DDR4. We set the cache mode with SNC-4 cluster (4 NUMA nodes) for all tests . All nodes are connected through the Intel Omni-Path interconnect. We used PiP-aware MPI extended from MPICH (commit 8cccb4c5 on the master branch) as the baseline implementation compared against the proposed CAB-MPI implementation. We used the gcc/gfortran compiler 4.8.5 to compile the MPI implementations and programs and used PAPI-5.7 for cache miss analysis. We set the $N_saturate$ threshold in the throughput-aware stealing strategy to 4 on Broadwell nodes and to 12 on KNL nodes based on our offline profiling by using the `memcpy` microbenchmark (see Section III-C3).

V. Microbenchmarks

In this section, we evaluate each showcase in CAB-MPI with a set of microbenchmarks. We also compare the stealing strategies and optimizations presented in Sections III-C and III-E, respectively. Unless specified otherwise, we enabled all optimizations in the showcase evaluation.

A. Intranode Data Transfer

We first evaluate work stealing for intranode data transfer. We use the experiments also to analyze the efficiency of localized, mixed, and throughput-aware work-stealing strategies. We extended the IMB-P2P PingPong test from the Intel MPI Benchmarks to add more processes waiting at a barrier so that they can join as stealing workers. Each of the PingPong processes touches the receive buffer after each round of data exchange (from low address to high address). We measure performance for both intra-NUMA and inter-NUMA PingPong. To isolate the performance of each strategy, we disabled all optimizations proposed in Section III-E and used a fixed 64 KB chunk size.

In Fig. 4a, processes 0–35 are placed sequentially from core 0 to 35; Fig. 4b uses the same approach. In Fig. 4c and 4d, processes 0 and 1 are placed on NUMA node 0 and node 1, respectively, to perform inter-NUMA pingpong. We first fill NUMA node 0 with processes and then fill other NUMA nodes sequentially.

A common trend observed from Figs. 4a and 4b is that speedup increases with increasing message size. The reason is that speedup is limited by the number of available tasks at small messages with fixed chunk size. In Fig. 4a, mixed work stealing always performs worse than the other strategies. The reason is that the workers from the local NUMA are already sufficient to saturate memory bandwidth. Thus, enabling remote stealing degrades performance. A comparison of localized and throughput-aware strategies shows that the latter have observable overhead at small messages mainly due to the bandwidth status checking. Figure 4b does not indicate such clear gaps on KNL because the high bandwidth of MCDRAM enables room for remote stealing. On the downside, however, remote stealing also forces the data of the destination buffer to be cached in different NUMA nodes, consequently causing extra overhead when the receiver touches the data. Similarly, we observe high deviation of the KNL results. Specifically, the data block (64 B) can be cached in different tiles after stealing. Consequently, the post-communication data touch suffers from varying access time subject to the location of the cached block.

In regard to inter-NUMA results (see Figs. 4c and 4d), we fix the message size to 8 MB and gradually add more processes starting from the first NUMA node. The PingPong processes are bound to the first two NUMA nodes, respectively. Before processes fill out the first NUMA node, the tasks generated

on the second NUMA node cannot be stolen in the localized strategy. Therefore, its performance is significantly worse than that of the other two. When more processes are added and the bandwidth becomes saturated, mixed stealing degrades performance because of inefficient remote stealing.

In conclusion, localized strategy can maximize memory throughput but lose remote stealing chances; mixed strategy can utilize remote stealing chances but may cause extra overhead when memory is saturated; throughput-aware strategy on average performs better than localized and mixed strategies and delivers close-to-optimal performance in all experiments. Nevertheless, they always significantly outperform the baseline. In the reminder of the evaluation, we use the throughput-aware strategy for all experiments.

B. Noncontiguous Data Transfer

We extended the PingPong test for noncontiguous data transfer. We used a 3D matrix of double, with the X dimension as the leading dimension and a fixed volume at 1 GB. We exchanged the X-Z plane in our experiments. The data layout is defined as a vector datatype. We increased the Z dimension with fixed Y dimension size at 2 doubles (the X dimension decreases).

1) Intranode Transfer: We compared two communication patterns, noncontiguous to contiguous (pack) and noncontiguous to noncontiguous (pack-unpack), on both Broadwell and KNL nodes. With increasing numbers of processes, we observe consistent speedup with all Z dimension sizes (see Fig. 5). We find up to 4x and 6.7x speedup in the pack tests on Broadwell and KNL, respectively. The speedup in the pack-unpack tests is close to 3.7x on Broadwell and 6.1x on KNL. We note that the speedup on KNL suddenly slows after the number of processes becomes more than 16 because remote stealing was not enabled in the throughput-aware strategy. Thus only 16 processes performed the work even when more processes were added on the remote NUMA nodes (each KNL NUMA node contains 16 processes).

2) Internode Transfer: To demonstrate the benefits of work stealing in internode communication, we performed the same pack-unpack PingPong test with the X-Z plane datatype on two Broadwell nodes. We expect that the internal packing on the sender and the unpacking on the receiver can be improved by work stealing. We fixed the Z dimension size at 256 count of doubles and gradually increased the number of idly waiting processes (i.e., workers) on each node. As shown in Fig. 6, the performance with stealing significantly outperforms that of the baseline and achieves up to 46% improvement. When the number of processes on each node is greater than 9, adding more workers does not help performance further because the memory bandwidth has been saturated. This trend matches our observation in the intranode experiments (Figure 5).

C. Accumulate Operation

We then evaluated the accumulate operation. To isolate the speedup in compute-centric reducing tasks, we used contiguous data with the double datatype in our experiments. We extended the IMB-RMA Accumulate test from the Intel MPI Benchmarks by replacing the lock-flush-unlock synchronization with fence. Thus, the other non-communicating processes can wait inside MPI and perform stealing. In the intra-NUMA experiments, both rank 0 and rank 1 were on the same NUMA node; all processes waiting at the fence call could steal the exposed reducing tasks. In the internode experiments, rank 0 and rank 1 were on separate nodes; stealing tasks were available only on the target node (node 1).

Figure 7 reports the results. Work stealing consistently improves performance for all data sizes. It delivers up to 3.7x speedup in the intra-NUMA test and more than 1.8x in the internode version. We note that the trend of the internode results is similar to that of the intra-NUMA version. The speedup is reduced because of the constant cost of network data transfer. While using a 128 KB data size, we notice both intra-NUMA and internode accumulate speedup gradually decreases because of task dequeue contention and bandwidth status checking overhead. In both experiments, we observe higher speedup with large data size (e.g., 8 MB) because it provides more work that can be accelerated by the workers. The increase of speedup slows after having more than 8 processes on the node because the ceiling of memory throughput is reached.

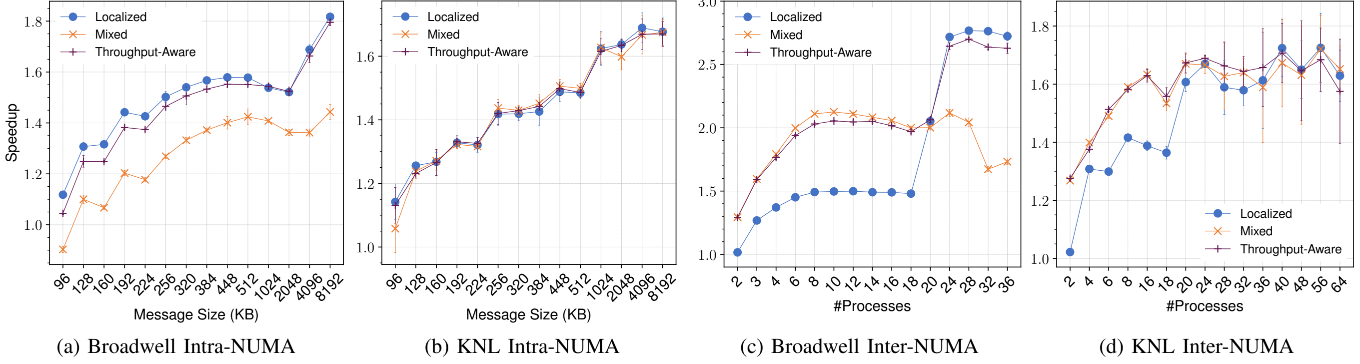
D. Optimizations Evaluation

We then separately analyzed the optimizations proposed in Section III-E.

1) Reversed Task Enqueue: We reused the PingPong benchmark used in Section V-A. We launched 36 processes on a Broadwell node and set the data size to 90 MB (twice the 45 MB LLC size on Broadwell). Each process accesses the data of the destination buffer from low address to high address after data exchange. Thus, we set the info hint `post_comm_access=lo-to-hi` to the world communicator. With the reversed task enqueue optimization, CAB-MPI posts tasks from high address to low address of the buffers. As shown in Fig. 8, this optimization can reduce the post-communication access time by 11%. The result can be clearly explained by the reduced L2 and L3 cache misses, as indicated in the graph.

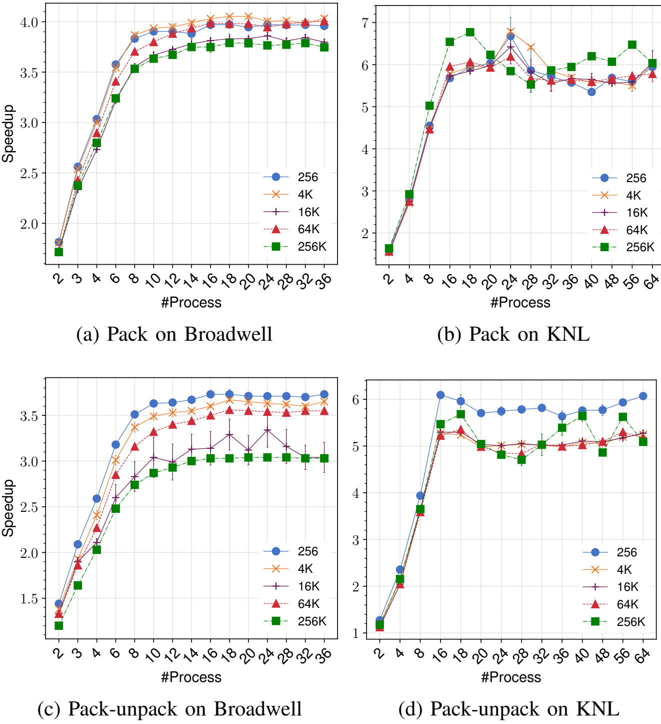
2) Noncontiguous Task Bundle: We reused the pack-unpack PingPong benchmark with the X-Z plane datatype to evaluate the noncontiguous task bundle optimization. We performed the experiment on a single Broadwell node. As reported in Fig. 9, the optimization significantly improves performance, contributing up to 1.5x speedup (with Z=256 count of doubles) for both intra-NUMA and inter-NUMA cases. The speedup rate decreases with larger Z dimension sizes (longer stride in the vector layout). The trend is expected, as we discussed in Section III-E2.

3) On-Demand Chunking: We analyzed the performance of different static chunk sizes by using the contiguous PingPong benchmark. We created 36 processes on a Broadwell node and varied the chunk size for different message sizes. As shown in Fig. 10, a small chunk size (e.g., 16 KB, 32 KB) is more



(a) Broadwell Intra-NUMA (b) KNL Intra-NUMA (c) Broadwell Inter-NUMA (d) KNL Inter-NUMA

Fig. 4: Intranode contiguous PingPong with comparison of localize, mixed, and throughput-aware stealing strategies: (a) and (b) vary the message size with fixed number of processes (36 and 64 for Broadwell and KNL, respectively); (c) and (d) vary the number of processes with fixed 8 MB message size. In all tests, only two processes perform PingPong; the others remain idle and behave as workers. In (c) and (d) the workers are sequentially increased from the first NUMA node. Core binding is set for all processes.



(a) Pack on Broadwell (b) Pack on KNL

(c) Pack-unpack on Broadwell (d) Pack-unpack on KNL

Fig. 5: Intra-NUMA noncontiguous PingPong with varying Z dimension sizes in the X-Z plane of a 3D matrix. Each line represents a Z dimension size (count of doubles).

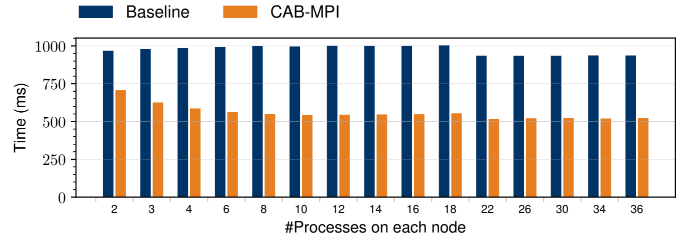
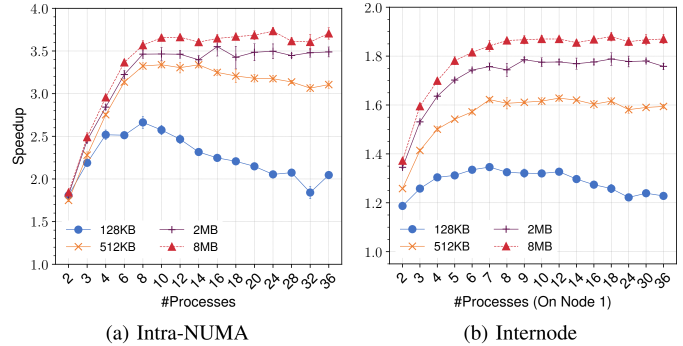


Fig. 6: Internode noncontiguous PingPong on two Broadwell nodes. The data layout uses the X-Z plane of a 3D matrix with Z=256 (count of doubles).



(a) Intra-NUMA (b) Internode

Fig. 7: One-sided Accumulate with `MPI_SUM` reduce operation and varying data size (from 128 KB to 8 MB) on Broadwell. Data is contiguous with the double datatype. Only rank 0 performs Accumulate; the others behave as workers.

beneficial for small messages; for large messages, however, large chunk sizes (e.g., 32–96 KB) perform better. The reason is that a small chunk size enables sufficient tasks for small messages; when a message becomes large, a large chunk size can ensure less task-stealing overhead. The proposed on-demand chunking allows CAB-MPI to set a different chunk size for different message sizes; thus it always delivers the best performance.

E. *Stealing Overhead Analysis*

The overheads of the work-stealing mechanism are caused by owner task creation, owner/worker queue operations, and operations for updating and checking n_{active} on every process (only in the throughput-aware strategy). We demonstrated the overheads by measuring two modified CAB-MPI versions

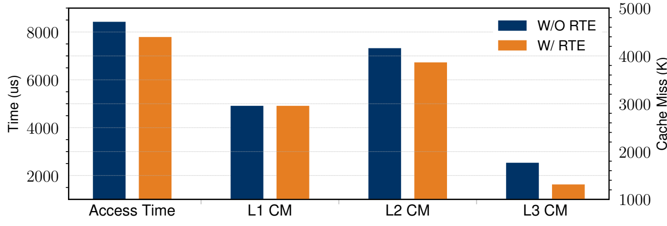
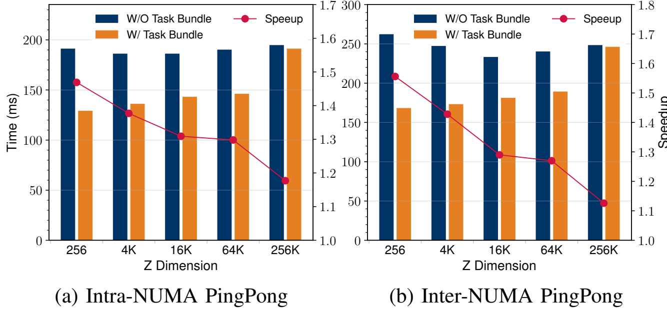


Fig. 8: Reversed task enqueue (RTE) evaluation based on intra-NUMA PingPong on a Broadwell node. Destination buffer access time, L1, L2, and L3 cache misses are reported.



(a) Intra-NUMA PingPong (b) Inter-NUMA PingPong

Fig. 9: Task bundle performance and speedup in intranode noncontiguous PingPong on Broadwell.

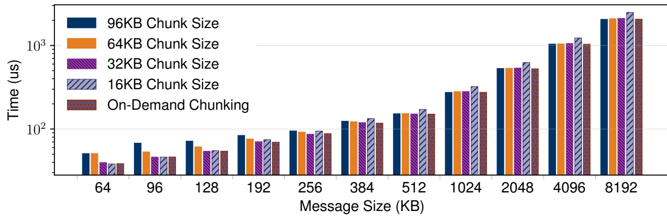


Fig. 10: On-demand chunking evaluation.

with the intra-NUMA contiguous PingPong test with small messages. The first version enables work stealing for any message size but keeps the chunk size unmodified. The owner process does not expose any stealing task because the message size is always smaller than a single chunk. Thus, the workers perform "empty checking" without stealing any task. We abbreviate this version as CABMPI-check-only. The second version also forces each message to split into two tasks (denoted by CABMPI-check-steal). Therefore, the remaining stealing overhead can be shown. As shown in Figure 11, CABMPI-check-only reports close to 0.15μs overhead on a Broadwell node in comparison with the baseline. This is caused mainly by the checking of n_{active} from processes on remote NUMA nodes. The overhead produced by CABMPI-check-steal is more significant (e.g., close to 6.5μs at 2 B message). We analyzed that the overhead is generated mainly by the lock contention on task queues that are concurrently accessed by 34 workers. However, we note that CAB-MPI is designed for medium and large message transfer (e.g., we set a threshold at 64 KB on our platform) and thus the contention overhead is negligible in practice. The small check-only overhead may degrade performance for applications that perform only small messages (i.e., no stealing). The user can disable work stealing to eliminate such an overhead.

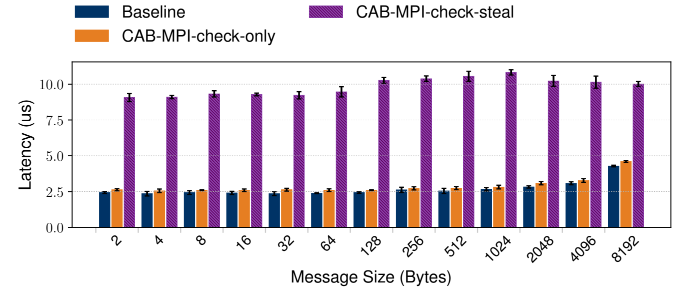


Fig. 11: Stealing overhead evaluation with small messages by using intra-NUMA PingPong on a Broadwell node with 36 processes. Similar trend is observed on KNL.

F. Shared-Memory-Based Intranode Data Transfer

Several MPI implementations utilize shared-memory techniques (e.g., CMA, XPMEM, PiP) to optimize MPI intranode communication. We compared CAB-MPI with these state-of-the-art optimizations. To be specific, we measured MPICH uses POSIX shared memory (denoted by MPICH-posix), MPICH with the XPMEM cooperative protocol [28] (MPICH-xpmem-coop), OpenMPI using CMA (version 4.0.3, denoted by OMPI-cma), PiP-aware MPI extended from MPICH (baseline) [17], and MPC based on thread-based data sharing (version 3.4.0) [22]. [4] The MPICH options use commit 427cdb07 from the master branch. We compared these approaches with CAB-MPI through the intra-NUMA PingPong test on a single Broadwel node as shown in Fig. 12. We note that the baseline PiP-aware MPI performs copy only on the receiver whereas MPICH-xpmem-coop utilizes both the sender and receiver to perform the copy, thus the latter shows better performance. Nevertheless, CAB-MPI improves the performance over all existing approaches by utilizing the local idle processes.

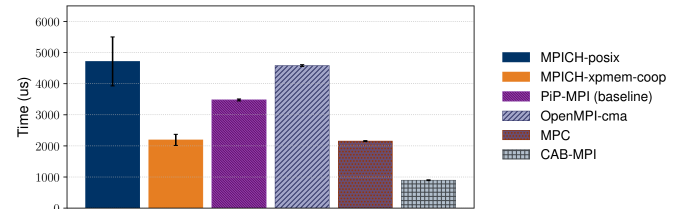


Fig. 12: Comparison of shared-memory-based optimizations by measuring intra-NUMA PingPong with a fixed message size at 8 MB and fixed number of processes at 36 on a Broadwell node. Only two processes perform PingPong; the others remain idle or behave as workers.

[4] MPC uses modified gcc 7.3.0 and software package which may cause unfair comparison with the other approaches.

VI. Application Evaluation and Analysis

We evaluated our approach on two miniapplications: miniGhost and BSPMM.

A. MiniGhost

MiniGhost is a miniapplication developed for exploring the context of exchanging interprocess boundary data that is widely seen in finite difference and finite volume computations [5]. MiniGhost is often used to mimic different stencils used in HPC applications. Our experiments used its 3D 7-point stencil where each process computes a 7-point stencil for $nvar$ number of 3D grids each with $(nx \times ny \times nz)$ dimension. We used the default bulk synchronous parallel with message aggregation (BSPMA) method where each plane of the grids is accumulated into a single message and exchanged with the neighbor. For each of the X-Y, Y-Z, and X-Z planes, we defined a different vector derived datatype to describe the layout of data in $nvar$ grids and directly specify it in the halo-exchange communication. For instance, the accumulated message for the X-Y plane on each process can be represented with a vector with $nvar$ count of blocks each with $(nx \times ny)$ length and $(nx \times ny \times nz)$ stride. Compared with the manual pack/unpack-based implementation in the original miniGhost code, this approach allows MPI to directly copy noncontiguous data into the internal buffer that is ready for data transfer. We fixed the data size to 1 GB ($nx \times ny \times nz \times nvar \times$ sizeof(double)=1 GB) on each process and set nx, ny, and nz equal (each grid is a cube). Thus, the global problem size is 1 GB$\times P$, where P is the total number of processes. We also modified the miniGhost code to use the MPI Cartesian topology in order to generate the optimal process grid (e.g., $8 \times 8 \times 9$ with 576 processes).

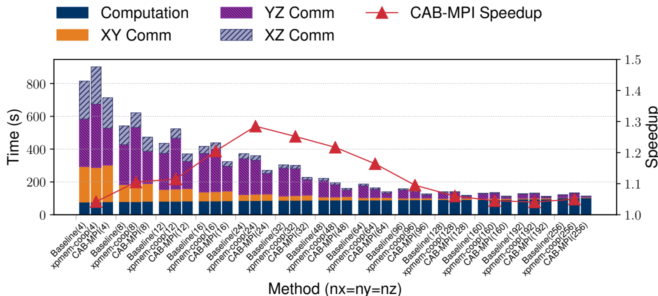


Fig. 13: MiniGhost with 3D 7-point stencil and BSPMA method running on 576 cores (16 nodes) on Broadwell. The global data size is fixed to 576 GB with varying nx=ny=nz, $nvar$ local parameters; the optimal $(8 \times 8 \times 9)$ process grid is used.

Figure 13 presents the execution time and speedup with varying problem parameters. We increased nx, ny, and nz at the same time; hence $nvar$ decreases. The overhead of the computing portion remains similar for all inputs. When nx=ny=nz are small, the dominant overhead is caused by the halo exchange communication with extremely sparse data elements. When the grid size becomes large and $nvar$ decreases, the program becomes more compute-bound, and the communication overhead is generated mainly by the Y-Z plane. CAB-MPI improves the internal pack/unpack speed for all three planes. However, it achieves the best speedup for the Y-Z plane. This also justifies the reason that the speedup increases from grid size 4 to 24. For larger grid sizes, the constant computing portion causes the major overhead, and thus the overall speedup decreases. The best speedup achieved by CAB-MPI is 1.3x at nx=ny=nz=24, $nvar$=9709. Unlike the observation from Fig. 12, MPICH-xpmem-coop performs even worse than baseline (PiP-aware MPI with *1-copy*) for grid sizes smaller than 16. The reason is that its cooperative copy is not process idleness aware; thus, adding more workload on the sender process aggravates load imbalance.

Figure 14 shows the miniGhost weak-scaling performance with CAB-MPI on up to 128 Broadwell nodes (4,608 processes) by using a fixed set of parameters nx=ny=nz=24, $nvar$=9709 on each process. Roughly speaking, CAB-MPI delivers improved performance with varying number of processes. The speedup gradually decreases at large scale, however, because the overhead of network data transfer becomes dominant. Nevertheless, CAB-MPI always outperforms the baseline MPI implementation and MPICH-xpmem-coop.

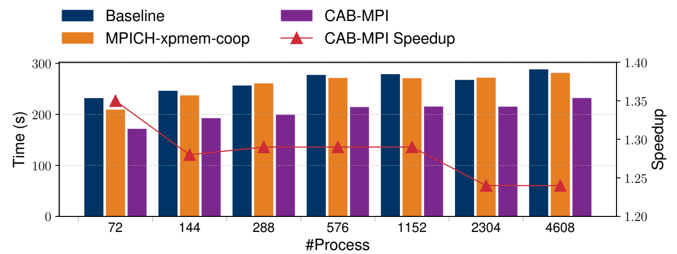


Fig. 14: Weak-scaling evaluation of miniGhost with 3D 7-point stencil and BSPMA method running on up to 128 Broadwell nodes. Each process uses a fixed set of parameters nx=ny=nz=24, $nvar = 9709$ (1 GB local data size, and more than 4 TB global data size on 128 nodes).

B. BSPMM

NWChem [29] is a widely used computational chemistry application suite. NWChem is developed on top of Global Arrays over the MPI one-sided model [30], [31]. A typical *get–compute–update* pattern is widely used in all the internal phases of NWChem, which every process essentially performs by varying the size of matrix-matrix multiplication for multidimensional tensor contraction by coordinating with others through get and accumulate operations. BSPMM is a miniapplication that mimics the one-sided *get–compute–update* computation in NWChem through a 2D sparse matrix multiplication $A \times B = C$. Each process asynchronously gets subblocks from the global matrices A and B, performs `dgemm` with the subblocks locally, and then accumulates the result into the remote C matrix. The ownership of each subblock computation is scheduled by updating a global shared counter with MPI atomic fetch_and_op. The subblock data is

represented as a strided subarray derived dataype in MPI. We expect that CAB-MPI can optimize BSPMM from intranode data transfer (for both get and accumulate), pack for internode accumulates (noncontiguous get does not apply because it is transferred via multiple RDMA requests in MPICH), and the reduce computation associated with each accumulate.

We set each global matrix size to 102400×102400 and used block size 1024 (both in count of doubles) with double data elements. We performed strong scaling on the Broadwell cluster on up to 1,152 processes. As shown in Fig. 15, both get and accumulate can be improved with CAB-MPI on a single node (36 processes). When scaling across multiple nodes, internode accumulates becomes the dominant overhead in the overall execution time and thus contributes to higher speedup. We achieved the best speedup of 1.4x on 144 processes (4 nodes). We also notice that the overall speedup gradually decreases after scaling over 144 processes. The reason is that the proportion of the reduce computation reduces in each accumulate since the network data transfer takes longer time. MPICH-xpmem-coop achieves performance similar to that of the baseline because its cooperative protocol cannot apply to one-sided communication where the remote process is not required to make an MPI call explicitly.

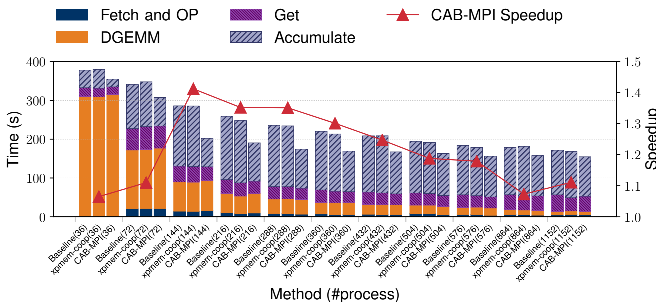


Fig. 15: BSPMM strong scaling and overhead analysis on Broadwell using global matrix size 102400×102400 and block size 1024 (both in count of double elements).

C. Discussion of Application-Level Performance Impact

CAB-MPI can benefit both regular and irregular applications. For instance, the miniGhost evaluation showed improved performance in the regular bulk synchronous parallelism pattern where we observed that stealing performs mainly in blocking calls such as `MPI_Wait_all` and `MPI_Barrier`. BSPMM is a typical example of the irregular application pattern where CAB-MPI performs stealing in blocking `MPI_Win_flush` calls. However, we note a limitation of CAB-MPI in that it relies on the semantics of MPI blocking calls to identify idle processes (i.e., valid workers). Hence it cannot help applications that use only nonblocking calls to check the completion of messages (e.g., `MPI_Test`). The concept of work stealing is still applicable to such applications; however, an additional method is required for process idleness determination. We plan to address it in future work.

VII. Related Work

In this paper, we propose an interprocess work-stealing mechanism in MPI to dynamically balance the MPI internal tasks by using idle processes at runtime. The implementation is based on a process-memory-sharing environment. We divide the related work into two broad topics: interprocess load balance and work stealing.

A. Interprocess Load Balance

Dynamic load balance is a common approach for irregular workloads or for applications adapting heterogeneous execution environments. This approach is widely utilized in both domain applications and runtime systems. Flaherty et al. [1] and Biswas et al. [2] introduced their dynamic load balancer approaches for irregular workloads in mesh applications by repartitioning domains. Sheridan et al. [32] presented a distributed work-stealing scheme for X10 regular applications. At runtime level, AMPI [23] executes processes on top of user-level threads and adopts Charm++ [24] to migrate tasks between processes to dynamically balance workloads. The lightweight user-level thread-based implementation allows the user to overdecompose the problem and create more tasks than the number of underlying cores. Therefore, migrating tasks across cores can potentially make a workload balanced on each core. The task in AMPI is essentially an MPI rank containing both user computation and communication work. AMPI applications have to periodically invoke the AMPI migrate function in order to allow the runtime to move tasks across cores for load balance. Unlike these approaches, CAB-MPI focuses on the workload balance of the MPI internal work including both data movement tasks and compute-centric reduce tasks. CAB-MPI does not require the user application to overdecompose, and the communication load balance is fully transparent to user applications. Our work is based on the model that *the user computation is managed and balanced by each individual application while the MPI runtime balances common internal workloads; hence both contribute to overall workload balance.*

B. Work Stealing

Traditional work-stealing mechanisms are designed for multithreading environments. The work-stealing strategies often focus on computing tasks. LAWS [33] involves a triple-level work-stealing algorithm to make idle threads steal tasks from local workers, the local cache-friendly task pool, and the remote cache-friendly task pool, in order to maximize cache reuse. ADWS [10] provides hierarchical localized work stealing to steal tasks only in an activated range, for better data locality. HotSLAW [34] extends stealing beyond intranode to distributed environments; it hierarchically picks a victim for work stealing to keep data access as local as possible. Barghi et al. [35] designed a locality-aware work stealing based on the actor model and NUMA architectures. Many other methods [36]–[44] also have tackled NUMA-aware work stealing by increasing local data access to mitigate NUMA effects on remote task stealing. Instead of creating tasks beforehand,

cooperative stealing [45], [46] utilizes the message-passing-based approach where victims create tasks only when the worker sends a stealing request, in order to avoid overhead caused by concurrent deques. The work-stealing strategies proposed in our work are different from these solutions in several ways. We focus on the internal tasks of MPI that often involve multiple data buffers that may locate in different memory domains. Thus, the locality-aware optimizations must be entirely redesigned. Moreover, the performance of MPI task stealing is sensitive to memory throughput. Hence, we propose a throughput-aware strategy that can maximize the throughput by prioritizing local stealing while still allowing remote stealing when the number of local idle processes is not sufficient. We do not utilize cooperative stealing in our scenario because the stealing must be sufficiently lightweight.

VIII. Conclusion

Communication imbalance is ubiquitous among HPC applications. Eliminating unbalanced communication at the application level is difficult mainly because of the challenges to estimate the amount of workloads. Load balance will not be accurate if the application developer distributes communication loads based only on the message size, because the data transfer overhead may vary at runtime in different situations (e.g., intranode vs. internode, contiguous vs. noncontiguous). To this end, we presented CAB-MPI, an MPI implementation that can dynamically balance MPI communication through novel interprocess work stealing. The proposed communication balance is transparent to user applications. We have designed several stealing strategies and optimizations based on the unique features of the MPI internal work. We showcased the benefit of the work-stealing mechanism through three types of MPI internal work: intranode data transfer, pack/unpack for noncontiguous data movement, and computation in one-sided accumulates. We evaluated the solution by using a set of microbenchmarks and proxy applications on both Intel Xeon and Xeon Phi platforms. Evaluation results indicate up to 1.3x improved performance in the stencil-based miniGhost proxy application over 576 Xeon cores and a 1.4x speedup in the one-sided BSPMM application.

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] J. E. Flaherty, R. M. Loy, C. Özturan, M. S. Shephard, B. K. Szymanski, J. D. Teresco, and L. H. Ziantz, "Parallel Structures and Dynamic Load Balancing for Adaptive Finite Element Computation," *Applied Numerical Mathematics*, vol. 26, no. 1–2, pp. 241–263, 1998.

[2] R. Biswas, S. K. Das, D. J. Harvey, and L. Oliker, "Parallel Dynamic Load Balancing Strategies for Adaptive Irregular Applications," *Applied Mathematical Modelling*, vol. 25, no. 2, pp. 109–122, 2000.

[3] B. Hendrickson and K. Devine, "Dynamic Load Balancing in Computational Mechanics," *Computer Methods in Applied Mechanics and Engineering*, vol. 184, no. 2, pp. 485–500, 2000.

[4] K. D. Devine, E. G. Boman, and G. Karypis, *Partitioning and Load Balancing for Emerging Parallel Applications and Architectures*, 2006, pp. 99–126.

[5] R. F. Barrett, C. T. Vaughan, and M. A. Heroux, "MiniGhost: A Miniapp for Exploring Boundary Exchange Strategies using Stencil Computations in Scientific parallel computing," *Sandia National Laboratories, Tech. Rep. SAND*, vol. 5294832, p. 2011, 2011.

[6] Y. Guo, J. Zhao, V. Cave, and V. Sarkar, "SLAW: A Scalable Locality-Aware Adaptive Work-Stealing Scheduler," in *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*. IEEE, 2010, pp. 1–12.

[7] R. D. Blumofe and C. E. Leiserson, "Scheduling Multithreaded Computations by Work Stealing," *Journal of the ACM (JACM)*, vol. 46, no. 5, pp. 720–748, 1999.

[8] O. Tardieu, H. Wang, and H. Lin, "A Work-Stealing Scheduler for X10's Task Parallelism with Suspension," *ACM SIGPLAN Notices*, vol. 47, no. 8, pp. 267–276, 2012.

[9] 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.

[10] S. Shiina and K. Taura, "Almost Deterministic Work Stealing," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2019, p. 47.

[11] N. S. Arora, R. D. Blumofe, and C. G. Plaxton, "Thread Scheduling for Multiprogrammed multiprocessors," *Theory of computing systems*, vol. 34, no. 2, pp. 115–144, 2001.

[12] D. Chase and Y. Lev, "Dynamic Circular Work-Stealing Deque," in *Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures*. ACM, 2005, pp. 21–28.

[13] U. A. Acar, G. E. Blelloch, and R. D. Blumofe, "The Data Locality of Work Stealing," in *Proceedings of the twelfth annual ACM symposium on Parallel algorithms and architectures*. ACM, 2000, pp. 1–12.

[14] D. Hendler and N. Shavit, "Non-Blocking Steal-Half Work Queues," in *Proceedings of the twenty-first annual symposium on Principles of distributed computing*. ACM, 2002, pp. 280–289.

[15] Q. Chen, Z. Huang, M. Guo, and J. Zhou, "Cab: Cache Aware Bitier Task-Stealing in Multi-Socket Multi-Core Architecture," in *2011 International Conference on Parallel Processing*. IEEE, 2011, pp. 722–732.

[16] Q. Chen, M. Guo, and Z. Huang, "Adaptive Cache Aware Bitier Work-Stealing in Multisocket Multicore Architectures," *IEEE Transactions on Parallel and Distributed Systems*, vol. 24, no. 12, pp. 2334–2343, 2012.

[17] 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*. ACM, 2018, pp. 131–143.

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

[19] J. Vienne, "Benefits of Cross Memory Attach for MPI Libraries on HPC Clusters," in *Proceedings of the 2014 Annual Conference on Extreme Science and Engineering Discovery Environment*. ACM, 2014, p. 33.

[20] 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.

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

[22] M. Pérache, H. Jourdren, and R. Namyst, "MPC: A Unified Parallel Runtime for Clusters of NUMA Machines," in *Proceedings of the 14th International Euro-Par Conference on Parallel Processing*, ser. Euro-Par'08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 78–88.

[23] M. Bhandarkar, L. V. Kalé, E. de Sturler, and J. Hoeflinger, "Adaptive Load Balancing for MPI Programs," in *International Conference on Computational Science*. Springer, 2001, pp. 108–117.

[24] L. V. Kale and S. Krishnan, "CHARM++: A Portable Concurrent Object Oriented System Based on C++," in *OOPSLA*, vol. 93. Citeseer, 1993, pp. 91–108.

[25] S. Jain, R. Kaleem, M. G. Balmana, A. Langer, D. Durnov, A. Sannikov, and M. Garzaran, "Framework for Scalable Intra-Node Collective Operations Using Shared Memory," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, ser. SC '18. IEEE Press, 2018.

[26] J. M. Hashmi, S. Chakraborty, M. Bayatpour, H. Subramoni, and D. K. Panda, "Designing Efficient Shared Address Space Reduction Collectives for Multi-/Many-Cores," in *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2018, pp. 1020–1029.

[27] P. Hammarlund, A. J. Martinez, A. A. Bajwa, D. L. Hill, E. Hallnor, H. Jiang, M. Dixon, M. Derr, M. Hunsaker, R. Kumar *et al.*, "Haswell: The Fourth-Generation Intel Core Processor," *IEEE Micro*, vol. 34, no. 2, pp. 6–20, 2014.

[28] S. Chakraborty, M. Bayatpour, J. Hashmi, H. Subramoni, and D. K. Panda, "Cooperative Rendezvous Protocols for Improved Performance and Overlap," in *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2018, pp. 361–373.

[29] 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.

[30] J. Nieplocha, B. Palmer, V. Tipparaju, M. Krishnan, H. Trease, and E. Apra, "Advances, Applications and Performance of the Global Arrays Shared Memory Programming Toolkit," *The International Journal of High Performance Computing Applications*, vol. 20, no. 2, pp. 203–231, 2006.

[31] J. S. Dinan, P. Balaji, J. R. Hammond, S. Krishnamoorthy, and V. Tipparaju, "Supporting the Global Arrays PGAS Model using MPI One-Sided Communication," in *2012 IEEE International Parallel and Distributed Processing Symposium*, May 2012.

[32] 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.

[33] 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.

[34] S.-J. Min, C. Iancu, and K. Yelick, "Hierarchical Work Stealing on Manycore Clusters," in *Fifth Conference on Partitioned Global Address Space Programming Models (PGAS11)*, vol. 625, 2011.

[35] 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.

[36] 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.

[37] Q. Chen, M. Guo, and Z. Huang, "CATS: Cache Aware Task-Stealing Based on Online Profiling in Multi-Socket Multi-Core Architectures," in *Proceedings of the 26th ACM international conference on Supercomputing*. ACM, 2012, pp. 163–172.

[38] R. M. Yoo, C. J. Hughes, C. Kim, Y.-K. Chen, and C. Kozyrakis, "Locality-Aware Task Management for Unstructured Parallelism: A Quantitative Limit Study," in *Proceedings of the twenty-fifth annual ACM symposium on Parallelism in algorithms and architectures*. ACM, 2013, pp. 315–325.

[39] J. Lifflander, S. Krishnamoorthy, and L. V. Kale, "Optimizing Data Locality for Fork/Join Programs using Constrained Work Stealing," in *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2014, pp. 857–868.

[40] A. Drebes, A. Pop, K. Heydemann, N. Drach, and A. Cohen, "NUMA-Aware Scheduling and Memory Allocation for Data-Flow Task-Parallel Applications," in *ACM Sigplan Notices*, vol. 51, no. 8. ACM, 2016, p. 44.

[41] 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.

[42] M. Shaheen and R. Strzodka, "NUMA Aware Iterative Stencil Computations on Many-Core Systems," in *2012 IEEE 26th International Parallel and Distributed Processing Symposium*. IEEE, 2012, pp. 461–473.

[43] J. Paudel, O. Tardieu, and J. N. Amaral, "On the Merits of Distributed Work-Stealing on Selective Locality-Aware Tasks," in *2013 42nd International Conference on Parallel Processing*. IEEE, 2013, pp. 100–109.

[44] H. Zhao, Q. Chen, Y. Qiu, M. Wu, Y. Shen, J. Leng, C. Li, and M. Guo, "Bandwidth and Locality Aware Task-stealing for Manycore Architectures with Bandwidth-Asymmetric Memory," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 15, no. 4, p. 55, 2018.

[45] 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.

[46] T. Hiraishi, M. Yasugi, S. Umatani, and T. Yuasa, "Backtracking-Based Load Balancing," *ACM Sigplan Notices*, vol. 44, no. 4, pp. 55–64, 2009.