UNIVERSITY OF CALIFORNIA
RIVERSIDE

Exploring Interprocess Techniques for High-Performance MPI Communication

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Kaiming Ouyang

March 2022

Dissertation Committee:

Dr. Zizhong Chen, Chairperson
Dr. Laxmi Bhuyan
Dr. Daniel Wong
Dr. Zhijia Zhao
Dr. Min Si

Copyright by
Kaiming Ouyang
2022

The Dissertation of Kaiming Ouyang is approved:

____

____

____

____

____

Committee Chairperson

University of California, Riverside

Acknowledgments

First of all, I would sincerely and extremely thank my Ph.D. advisor Dr. Zizhong Chen and supervisor Dr. Min Si for their careful and generous guidance, support, patience in my Ph.D. life and study.

Dr. Zizhong Chen gave me the full research freedom that allowed myself to explore whatever research topic I am interested in. He always kindly gave me comments, suggestions and encouragement when I faced the challenges during research. Without Dr. Zizhong Chen, I would not be able to understand the beauty of high-performance computing and will not acquire the pleasant research progress and results. His kindness and patience drove me to keep going even if I was in dilemma. Dr. Min Si, as my supervisor at Argonne National Laboratory, instructed me to conduct the state-of-the-art MPI research works when I was a long-term intern at Argonne National Laboratory. She taught me with great care how to conduct Ph.D. research step by step and always encouraged me to share my research work and interact with other researchers. Her supports and comments act as a very important role in my research. Without Dr. Min Si, I would not have a chance to learn so many things and collaborate with wonderful people. In addition, I would especially thank Dr. Zizhong Chen and Dr. Min Si for their invaluable cares and understandings when I got illness and depressed in my Ph.D journey. Their kind supports bring me the power and confidence to overcome the unexpected difficulty.

I would thank Dr. Laxmi Bhuyan, Dr. Daniel Wong, Dr. Zhijia Zhao for serving as my Ph.D. dissertation committee. They gave me positive feedbacks, comments and suggestions so that I was able to finish the high-quality Ph.D. thesis.

It is my great pleasure to work with fantastic people at Argonne National Laboratory. Besides Dr. Min Si, Dr. Pavan Balaji also provided me many opportunities and supports when I was an intern. His straightforward comments and advice always enlightened me and helped me better understand the key point of a problem. Other group members such as Dr. Shintaro Iwasaki, Dr. Rohit Zambre, Dr. Hui Zhou, Dr. Yanfei Guo, Ken Raffenetti and so on are very kind and easygoing and I really appreciate their helps, comments and suggestions when I was in PMRS group. I would also thank my research collaborator Dr. Atsushi Hori who has provided me many supports on PiP shared memory techniques and gave me many valuable suggestions. Moreover, I also want to specially thank Dr. Yanfei Guo for his supervision, instruction and support.

I am so fortunate to meet the people at UCR and be able to make friends with them. Dr. Panruo Wu, Dr. Dingwen Tao, Dr. Hongbo Li, Dr. Jieyang Chen, Dr. Sihuan Li, Dr. Xin Liang, Yuanlai Liu, Dr. Kai Zhao, Yujia Zhai, Quan Fan, Elisabeth Giem and Dr. Chengshuo Xu all gave me invaluable and beautiful memory. I would thank them for their kindness and help during my Ph.D. life.

Funding Acknowledgement I am grateful to the funding supports 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 thesis was provided by the Laboratory Computing Resource Center on the Bebop cluster at Argonne National Laboratory.

Publication Acknowledgement I acknowledge that part of this thesis was published or submitted previously in the following conferences.

- Chapter 2 was previously published [91] in International Conference for High Performance Computing, Networking, Storage and Analysis (SC'20), Is Everywhere We Are, November 9-19, 2020
- Chapter 3 was previously published [90] in Proceedings of the 2021 IEEE International Conference on Cluster Computing (Cluster'21), Virtual Conference, September 7-10, 2021
- Chapter 4 was previously submitted [89] to 36th ACM International Conference on Supercomputing (ICS'22), Virtual Conference, June 27-30, 2022

To my parents for all the support.

ABSTRACT OF THE DISSERTATION

Exploring Interprocess Techniques for High-Performance MPI Communication

by

Kaiming Ouyang

Doctor of Philosophy, Graduate Program in Computer Science
University of California, Riverside, March 2022
Dr. Zizhong Chen, Chairperson

In exascale computing era, applications are executed at larger scale than ever before, which results in higher requirement of scalability for communication library design. Message Passing Interface (MPI) is widely adopted by the parallel application nowadays for interprocess communication, and the performance of the communication can significantly impact the overall performance of applications especially at large scale.

There are many aspects of MPI communication that need to be explored for the maximal message rate and network throughput.

Considering load balance, communication load balance is essential for high-performance applications. Unbalanced communication can cause severe performance degradation, even in computation-balanced Bulk Synchronous Parallel (BSP) applications. MPI communication imbalance issue is not well investigated like computation load balance. Since the communication is not fully controlled by application developers, designing communication-balanced applications is challenging because of the diverse communication implementations at the underlying runtime system.

In addition, MPI provides nonblocking point-to-point and one-sided communication models where asynchronous progress is required to guarantee the completion of MPI communications and achieve better communication and computation overlap. Traditional mechanisms either spawn an additional background thread on each MPI process or launch a fixed number of helper processes on each node. For complex multiphase applications, unfortunately, severe performance degradation may occur due to dynamically changing communication characteristics.

On the other hand, as the number of CPU cores and nodes adopted by the applications greatly increases, even the small message size MPI collectives can result in the huge communication overhead at large scale if they are not carefully designed. There are MPI collective algorithms that have been hierarchically designed to saturate inter-node network bandwidth for the maximal communication performance. Meanwhile, advanced shared memory techniques such as XPMEM, KNEM and CMA are adopted to accelerate intra-node MPI collective communication. Unfortunately, these studies mainly focus on large-message collective optimization which leaves small- and medium-message MPI collectives suboptimal. In addition, they are not able to achieve the optimal performance due to the limitations of the shared memory techniques.

To solve these issues, we first 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. The experimental results show the benefits of CAB-MPI through several internal processes in MPI, including intranode data transfer,

pack/unpack for noncontiguous communication, and computation in one-sided accumulates through a set of microbenchmarks and proxy applications on Intel Xeon and Xeon Phi platforms. Then, we propose a novel Dynamic Asynchronous Progress Stealing model (Daps) to completely address the asynchronous progress complication; Daps is implemented inside the MPI runtime, and it dynamically leverages idle MPI processes to steal communication progress tasks from other busy computing processes located on the same node. We compare Daps with state-of-the-art asynchronous progress approaches by utilizing both microbenchmarks and HPC proxy applications, and the results show the Daps can outperform the baselines and achieve less idleness during asynchronous communication. Finally, to further improve MPI collectives performance, we propose *Process-in-Process based Multiobject Interprocess MPI Collective* (PiP-MColl) design to maximize small and medium-message MPI collective performance at a large scale. Different from previous studies, PiP-MColl is designed with efficient multiple senders and receivers collective algorithms and adopts Process-in-Process shared memory technique to avoid unnecessary system call and page fault overhead to achieve the best intra- and inter-node message rate and throughput. We focus on three widely used MPI collectives `MPI_Scatter`, `MPI_Allgather` and `MPI_Allreduce` and apply PiP-MColl to them. Our microbenchmark and real-world HPC application experimental results show PiP-MColl can significantly improve the collective performance at a large scale compared with baseline PiP-MPICH and other widely used MPI libraries such as OpenMPI, MVAPICH2 and Intel MPI.

Contents

List of Figures xiii

List of Tables xvi

1 Introduction 1

1.1 Problem Statement . . . 5
1.2 Thesis Statement . . . 5

2 CAB-MPI: Exploring Interprocess Work-Stealing towards Balanced MPI Communication 7

2.1 Introduction . . . 7
2.2 Shared-Memory Technique Analysis . . . 11
2.3 Design and Implementation . . . 13
2.3.1 Basic Semantics Definition . . . 13
2.3.2 Framework Design . . . 16
2.3.3 Work-Stealing Strategies . . . 17
2.3.4 Work-Stealing Showcase . . . 22
2.3.5 Other Optimizations . . . 23
2.4 Experimental Configuration . . . 25
2.5 Microbenchmarks . . . 26
2.5.1 Intranode Data Transfer . . . 26
2.5.2 Noncontiguous Data Transfer . . . 28
2.5.3 Accumulate Operation . . . 32
2.5.4 Optimizations Evaluation . . . 33
2.5.5 Stealing Overhead Analysis . . . 36
2.5.6 Shared-Memory-Based Intranode Data Transfer . . . 37
2.6 Application Evaluation and Analysis . . . 38
2.6.1 MiniGhost . . . 38
2.6.2 BSPMM . . . 41
2.6.3 Discussion of Application-Level Performance Impact . . . 42
2.7 Summary . . . 43

3 Daps: A Dynamic Asynchronous Progress Stealing Model for MPI Communication 45
3.1 Introduction . . . 45
3.2 Background . . . 48
3.2.1 MPI Progress . . . 48
3.2.2 Asynchronous Progress . . . 50
3.3 Limitation of Static Asynchronous Progress . . . 50
3.4 Dynamic Asynchronous Progress Design . . . 53
3.4.1 Basic Definition . . . 53
3.4.2 Prerequisite Analysis . . . 54
3.4.3 Progress-Stealing Algorithm . . . 58
3.5 Implementation Challenges . . . 61
3.5.1 Network Interaction . . . 61
3.5.2 Thread Local Storage . . . 63
3.6 Experimental Evaluation . . . 66
3.6.1 Asynchronous Progress Capability . . . 67
3.6.2 Static Progress v.s. Dynamic Progress . . . 68
3.6.3 Scalability . . . 73
3.7 Summary . . . 73

4 Efficient Process-in-Process based Multiobject Interprocess MPI Collectives for Large-Scale Applications 76
4.1 Introduction . . . 76
4.2 Background . . . 79
4.2.1 POSIX Shared Memory . . . 79
4.2.2 Data Exchange Based Shared Memory . . . 80
4.2.3 Data Sharing Based Shared Memory . . . 80
4.2.4 Userspace Address Space Sharing . . . 81
4.3 Multiobject Interprocess MPI Collective Design . . . 81
4.3.1 Message Rate and Network Throughput . . . 82
4.3.2 Communication Cost Model . . . 83
4.3.3 PiP-MColl based Collective Algorithm Design . . . 84
4.3.4 Auxiliary MPI Collectives . . . 94
4.4 Experimental Results . . . 96
4.4.1 Experimental Setup . . . 96
4.4.2 Microbenchmark Evaluation . . . 97
4.4.3 Applications Evaluation . . . 104
4.5 Summary . . . 107

5 Conclusions 108

Bibliography 111

List of Figures

2.1 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 [11] with parameters nx=ny=nz=50,$nvar$=100 and process grid 4×6×6. The time is measured for processes on the first node. 9
2.2 High-level queue-based work-stealing framework. 18
2.3 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%. 20
2.4 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. 29
2.5 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). 30
2.6 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). 31
2.7 One-sided Accumulate with 12MPI_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. 33
2.8 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. 34
2.9 Task bundle performance and speedup in intranode noncontiguous PingPong on Broadwell. 35
2.10 On-demand chunking evaluation. 35
2.11 Stealing overhead evaluation with small messages by using intra-NUMA Ping-Pong on a Broadwell node with 36 processes. Similar trend is observed on KNL. 37

2.12 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. 38
2.13 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. 39
2.14 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). 41
2.15 BSPMM strong scaling and overhead analysis on Broadwell using global matrix size 102400×102400 and block size 1024 (both in count of double elements). 43

3.1 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. 52
3.2 Expected behaviors when a progress worker calls into `l2`$_{\texttt{entry_func()}}$ shared by the owner; the corresponding assessment question ID is marked behind each line. 57
3.3 Progress-stealing timeline with Psm2 network driver. 63
3.4 Evaluating asynchronous progress capability with Get, Put, Accumulate, and Isend/Irecv with two internode process. 69
3.5 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). Stage 1 us computation-heavy, and stage 2 is communication-heavy. 71
3.6 Comparing Daps with static progress using five-point 2D stencil with a total problem size of $4096 * 4096$ on 8 Bebop nodes. Showing computation and communication breakdown and the speedup of the communication portion compared to baseline. 72
3.7 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). 74

4.1 The inter-node message rate and network throughput of 4KB and 128KB point-to-point with various pairs of senders and receivers locating on two separate nodes. The network interconnect is Intel Omni-Path. 83
4.2 High-level design of PiP-MColl MPI_Scatter with overlapped intranode scatter. Shown as an example are 3 senders and 1 receiver on a node. The figure shows only the partial stages; the rest of the stages will repeat the same operations in stage 1 for each new generated subgroup. 86
4.3 Example of two-step PiP-MColl allgather algorithm for small-message communication with 16 nodes and 3 objects per node. 88
4.4 High-level PiP-MColl allgather algorithm for medium- and large-message size. The figure shows an example with 3 nodes and 3 objects per node and one step of the ring communication. The intra- and internode communications run in parallel for overlapping. 90
4.5 PiP-MColl–based large-message intranode reduce communication with 4 processes on a node. Each buffer is chunked based on the number of processes, and all data will be reduced into the root process destination buffer. 94
4.6 MPI_Scatter performance with different message size and numbers of nodes. PiP-MColl scatter uses the same algorithm throughout the tests. 98
4.7 MPI_Allgather performance with various message size and number of nodes. *PiP-MColl Opt* is the optimal case where the large-message algorithm is switched at 64 kB; *PiP-MColl Small* is the case where only the small-message algorithm is used throughout the test. 101
4.8 MPI_Allreduce performance with various double-type message counts and nodes. For message-based execution, MPI_Allreduce switches to the large-message algorithm when message count is larger than or equal to 8 kB. . . 103
4.9 MPI_Allgather performance comparison among Intel-MPI, OpenMPI, MVAPICH2, and PiP-MColl with different message sizes on 256 Xeon Broadwell nodes. 105
4.10 Strong scaling N-body simulation performance from 4 to 256 nodes. The particle is set as 10^5 and average runtime of multiple iterations is presented. 106

List of Tables

3.1 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. 65

4.1 Summary of MPI collective communication cost model symbols. 84

Chapter 1

Introduction

Message Passing Interface (MPI) is widely used in HPC applications for low latency and high network throughput communication. Especially in exascale era, applications are usually executed at large scale so that MPI communication plays an important role at the overall performance.

Considering the importance of MPI communication, many studies have been performed to improve the performance. One of research directions is to load balance the communication workload among processes. 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. [41] and Biswas et al. [15] introduced their dynamic load balancer approaches for irregular workloads in mesh applications by repartitioning domains. Sheridan et al. [103] presented a distributed work-stealing scheme for X10 regular applications. At runtime level, AMPI [12] executes processes on top of user-level threads and adopts Charm++ [71] to mi-

grate 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.

There are also researches that explore the benefits of work stealing which can be adopted by MPI communication load balance design. Traditional work-stealing mechanisms are designed for multithreading environments. The work-stealing strategies often focus on computing tasks. LAWS [25] 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 [104] provides hierarchical localized work stealing to steal tasks only in an activated range, for better data locality. HotSLAW [87] 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. [9] designed a locality-aware work stealing based on the actor model and NUMA architectures. Many other methods[32, 27, 128, 83, 36, 24, 101, 93, 130] 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 [4, 56] 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.

On the other hand, MPI asynchronous progress is also a critical part which can significantly impact the performance of communication. Jiang et al. [69] propose a thread-based asynchronous progress method on the InfiniBand clusters for one-side communication. Pritchard et al. [97] utilize CoreSpec capability provided by Cray XE Systems to dedicate one or more cores to background threads for asynchronous progress. Casper [109] designates user-specified number of ghost processes and offloads RMA and point-to-point operations to the ghosts processes for asynchronous progress.

The dynamic version of Casper[106] can disable inefficient asynchronous progress during a multistage execution, but the number of ghost processes cannot be dynamically changed. Vaidyanathan et al. [123] 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. [99] improve the thread-based asynchronous progress for MPI-only model by reducing the frequency of progress polling from the thread. Alternatively, PIOMan [118] 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 [78] 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. [112] exploit RDMA read and selective interrupt-based asynchronous progress on the InfiniBand cluster. On IBM systems, as described in [77, 76, 79], the

system interrupt-based progress mechanism is studied. In addition, MPI collectives are widely adopted in current HPC application and must be carefully designed to provide good scalability for large-scale execution. Many studies have been done to improve MPI collective performance. Thakur et al. [116] focus purely on the internode MPI collective algorithm design and can provide efficient collective communication in the general case. Jain et al. [67] propose a collective framework that implements *release* and *gather* primitives to implement all types of MPI collectives. This method is general and based on POSIX-SHMEM for data exchange to deliver high performance. LiMiC-, KNEM-, and CMA-based collective designs [34, 86, 22] are proposed to bypass the POSIX-SHMEM limitation. In their designs, processes single-copy exchange data through kernel and accelerate intranode collective performance. Hashmi et al. [53] design XPMEM-based MPI collectives to improve reduce-related MPI collectives with userspace data sharing. Processes expose and attach the private buffers through the system calls `xpmem_make` and `xpmem_attach`; then they can perform reduction directly from the attached buffer to achieve zero-copy communication.

Moreover, Kandalla et al. [72] propose a multileader–based allgather algorithm to increase network throughput, and the improved version propsoed by Parsons et al. [92] presents the POSIX-SHMEM–based multisender design to utilize multiple senders to increase network throughput. Other thread-based works [64, 94, 82, 113] utilize threads instead of processes to achieve high-bandwidth intranode communication.

1.1 Problem Statement

This thesis mainly focuses on three important problems that exist in current exascale computing era where HPC applications are executed on large-scale clusters.

The first problem we focus on is communication imbalance in which the processes can spend various amount of time in communication even if the communication workload is well-balanced by application developers. This issue causes idleness, resource waste and worse performance, which severely limits the scalability of MPI communication.

The second problem is the suboptimal MPI asynchronous progress. Asynchronous progress is usually triggered by the preallocated threads or processes, but the statically allocated threads or processes can result in resource waste when there is not enough asynchronous progress to perform during execution.

The last problem is the inefficient design of small- and medium-message MPI collectives at large scale. The traditional MPI collectives rely on the POSIX, XPMEM, CM, KNEM shared memory techniques with one leader per node for inter-node communication. The designs are not able to deliver maximal performance for small- and medium-message MPI collectives at large scale.

1.2 Thesis Statement

In this thesis, we propose the corresponding methods to mitigate or solve the issues. For the CAB-MPI design, it mitigates the imbalance for applications at large scale during communication where idle processes are able to steal the communication workload from other on-node processes; Daps provides the capability for idle processes to dynamically poll

the asynchronous progress of busy processes to improve overall performance; PiP-MColl is an efficient design for small- and medium-message MPI collectives which provide maximal message rate and bandwidth usage at large scale.

Chapter 2

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

2.1 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 [41, 15, 55, 33]. 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 2.1, 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.

Pursuing evenly distributed communication at the application level is impractical mainly because the users of MPI cannot precisely 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 very 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

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

communicating processes (e.g., cross-NUMA data transfer usually takes longer than that inside a NUMA node).

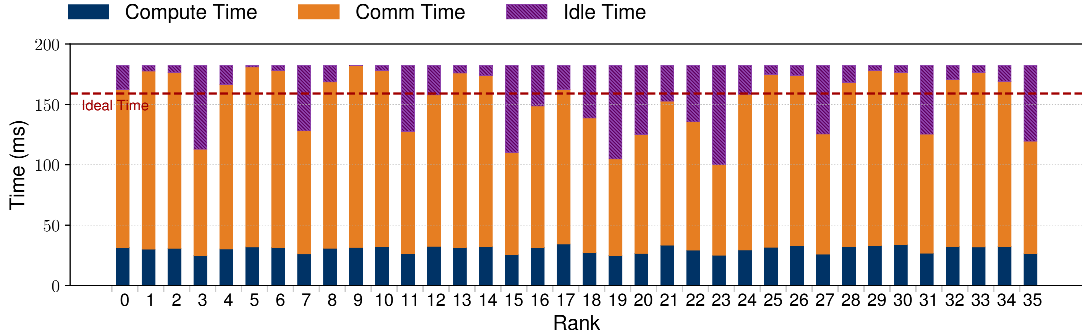


Figure 2.1: Unbalanced communication in 3D 7-point stencil on 4 Broadwell nodes (Intel Xeon E5-2695v4 CPU, 36 processes per node). The experiment adopted the miniGhost stencil program [11] with parameters nx=ny=nz=50,$nvar$=100 and process grid 4×6×6. The time is measured for processes on the first node.

To address this challenging issue, we believe that a runtime-level solution is essential. In this thesis, 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 [49, 16, 114, 120, 104, 6, 23, 3, 54, 28, 26]. 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 [63].

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.

2.2 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 [75] 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) [125] and KNEM [45] are two kernel-assisted techniques. A process can directly read/write a buffer of another process by using the system call provided by CMA or KNEM. To make a third process perform the copy for two processes, 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 [57] 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) [63] 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 [95]. The MPC runtime creates threads running as MPI processes so that intranode data transfer can be highly optimized. AMPI over Charm++[12, 71] 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 [63] 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 thesis.

2.3 Design and Implementation

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

2.3.1 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.

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.

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 2.3.4.

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]

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 thesis 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.*

2.3.2 Framework Design

We present our basic work-stealing framework in Figure 2.2. 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

[2] Shared-memory-based collective optimization [68, 53] is orthogonal to this work; we leave work stealing for such tasks as future work.

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 2.3.3) 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.

2.3.3 Work-Stealing Strategies

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

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

[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.

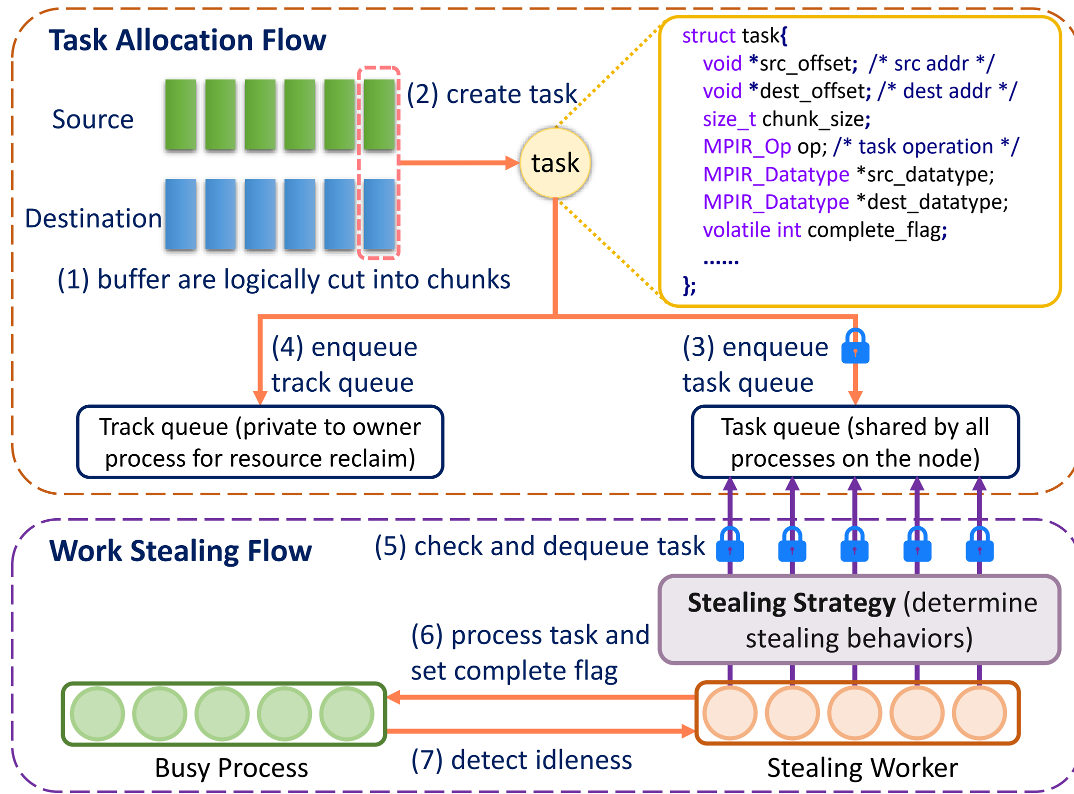


Figure 2.2: High-level queue-based work-stealing framework.

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.

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.

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 2.3, 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.

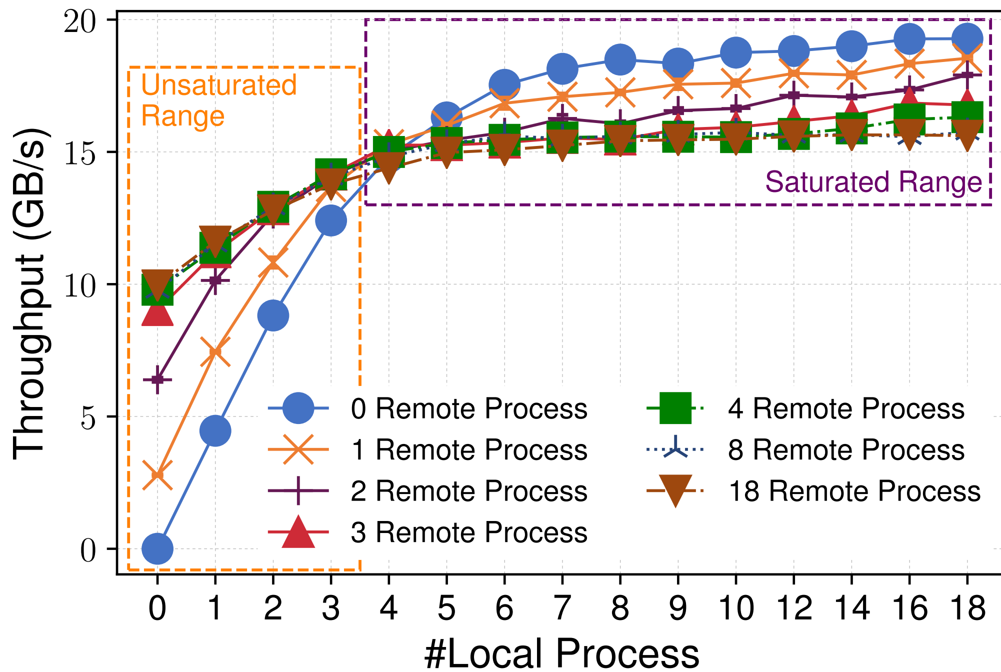


Figure 2.3: Memcpy throughput with variable number of local and remote processes on a Broadwell node (two NUMA nodes each with 18 cores). The results are averaged from ten runs, and the error is less than 4%.

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 2.3 (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}.

2.3.4 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.

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.

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.

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.

2.3.5 Other Optimizations

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.

Noncontiguous Task Bundle

In noncontiguous data transfer, an internal contiguous buffer is used together with the pack/unpack routines. On modern architectures[51] 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 2.4 we demonstrate such a trend. Nevertheless, the proposed optimization never causes performance degradation compared with the original approach.

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).

2.4 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 2.3.3).

2.5 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 2.3.3 and 2.3.5, respectively. Unless specified otherwise, we enabled all optimizations in the showcase evaluation.

2.5.1 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 2.3.5 and used a fixed 64 KB chunk size.

In Figure 2.4a, processes 0–35 are placed sequentially from core 0 to 35; Figure 2.4b uses the same approach. In Figure 2.4c and 2.4d, 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. 2.4a and 2.4b 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 Figure 2.4a, 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 2.4b 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 Figure 2.4c and 2.4d), 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.

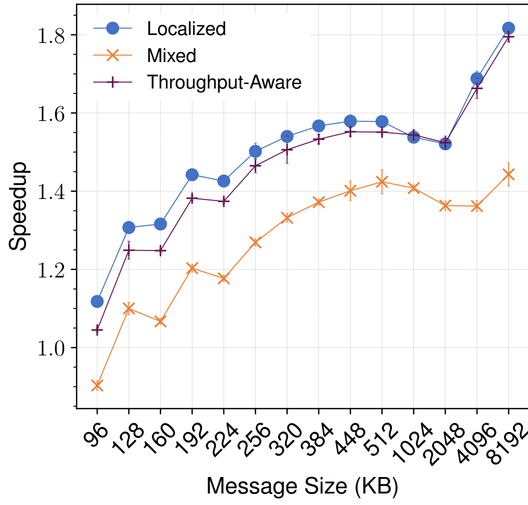
In conclusion, localized strategy can maximize memory throughput but lose remote stealing chances; mixed strategy can utilize remote stealing chances but may cause extra overhead when memory is saturated; throughput-aware strategy on average performs better than localized and mixed strategies and delivers 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.

2.5.2 Noncontiguous Data Transfer

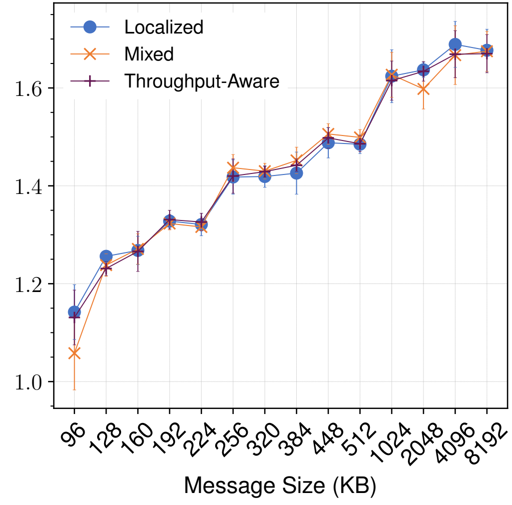
We extended the PingPong test for noncontiguous data transfer. We used a 3D matrix of double, with the X dimension as the leading dimension and a fixed volume at 1 GB. We exchanged the X-Z plane in our experiments. The data layout is defined as a vector datatype. We increased the Z dimension with fixed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

Intranode Transfer

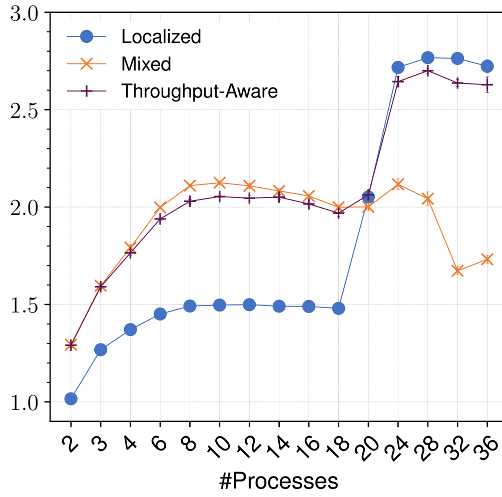
We compared two communication patterns, noncontiguous to contiguous (pack) and noncontiguous to noncontiguous (pack-unpack), on both Broadwell and KNL nodes. With increasing numbers of processes, we observe consistent speedup with all Z dimension sizes (see Figure 2.5). We find up to 4x and 6.7x speedup 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-



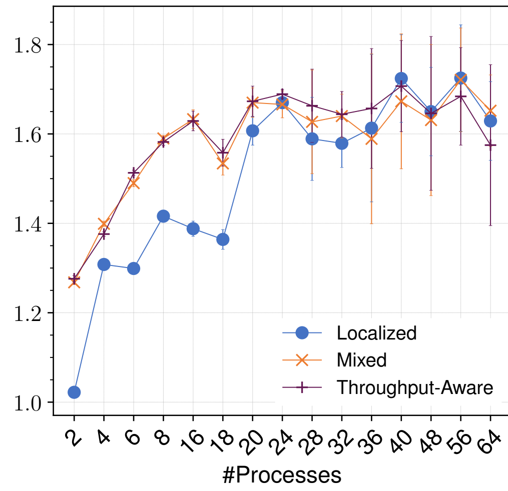
(a) Broadwell Intra-NUMA



(b) KNL Intra-NUMA



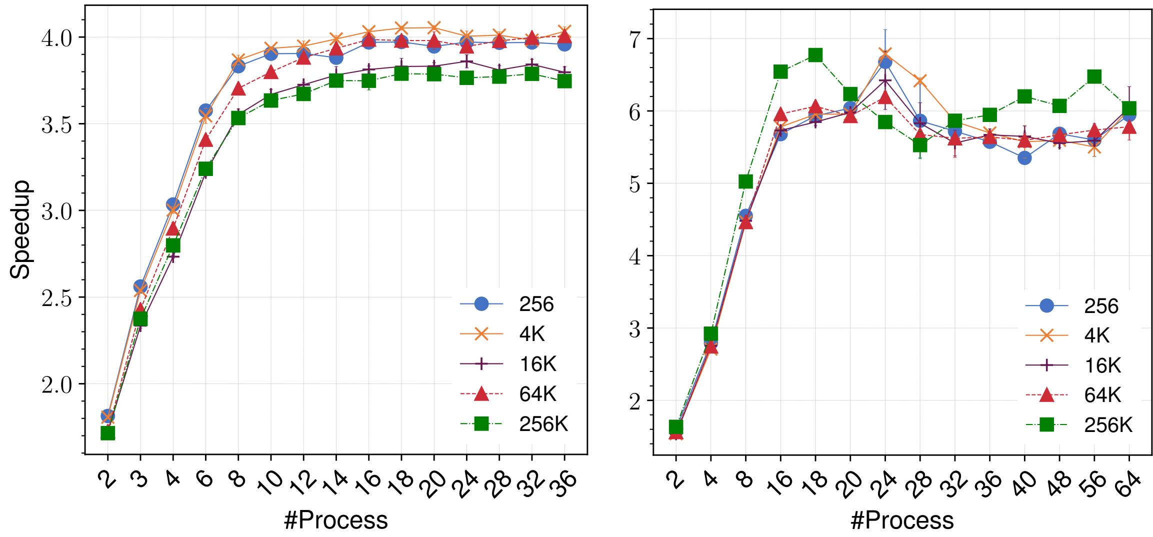
(c) Broadwell Inter-NUMA



(d) KNL Inter-NUMA

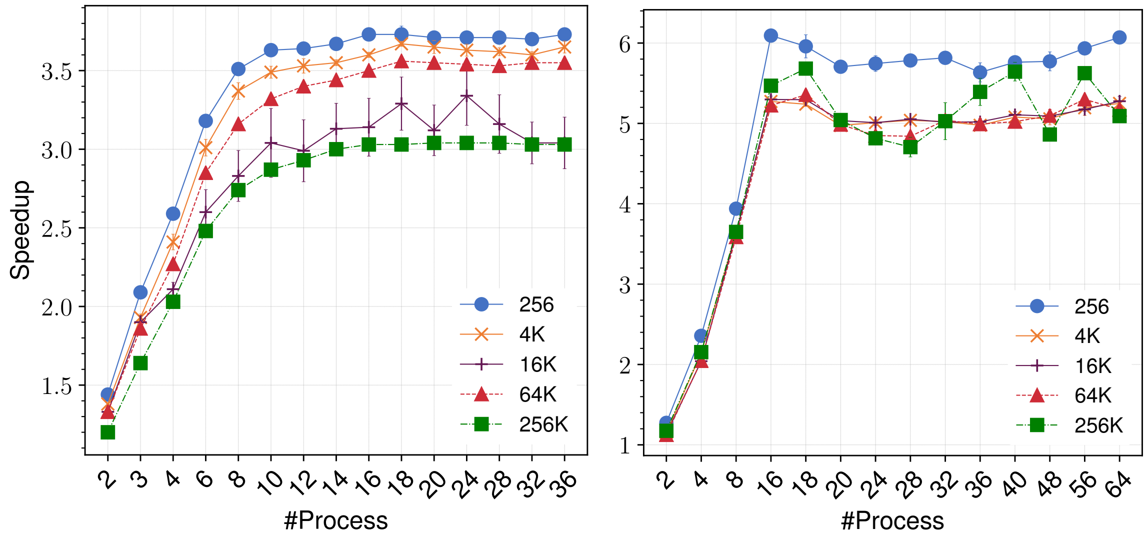
Figure 2.4: Intranode contiguous PingPong with comparison of localize, mixed, and throughput-aware stealing strategies: (a) and (b) vary the message size with fixed number of processes (36 and 64 for Broadwell and KNL, respectively); (c) and (d) vary the number of processes with fixed 8 MB message size. 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.

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).



(a) Pack on Broadwell

(b) Pack on KNL



(c) Pack-unpack on Broadwell

(d) Pack-unpack on KNL

Figure 2.5: 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).

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 Figure 2.6, 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 2.5).

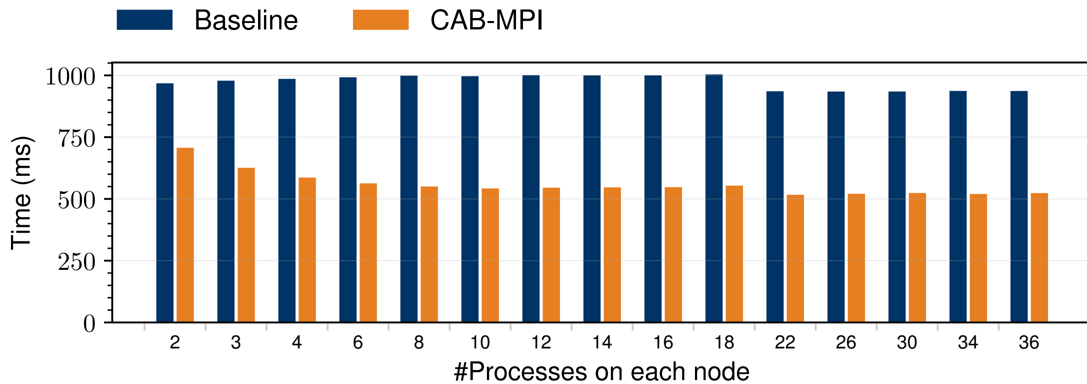
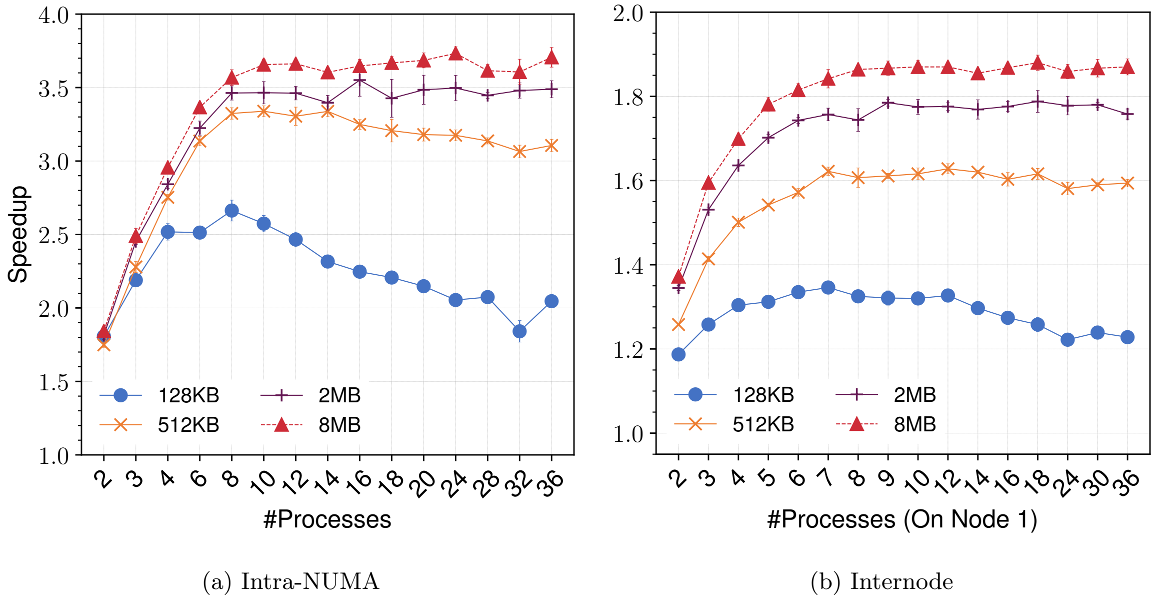


Figure 2.6: Internode noncontiguous PingPong on two Broadwell nodes. The data layout uses the X-Z plane of a 3D matrix with Z=256 (count of doubles).

2.5.3 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 2.7 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.



(a) Intra-NUMA

(b) Internode

Figure 2.7: 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.

2.5.4 Optimizations Evaluation

Reversed Task Enqueue

We reused the PingPong benchmark used in Section 2.5.1. 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 Figure 2.8, 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.

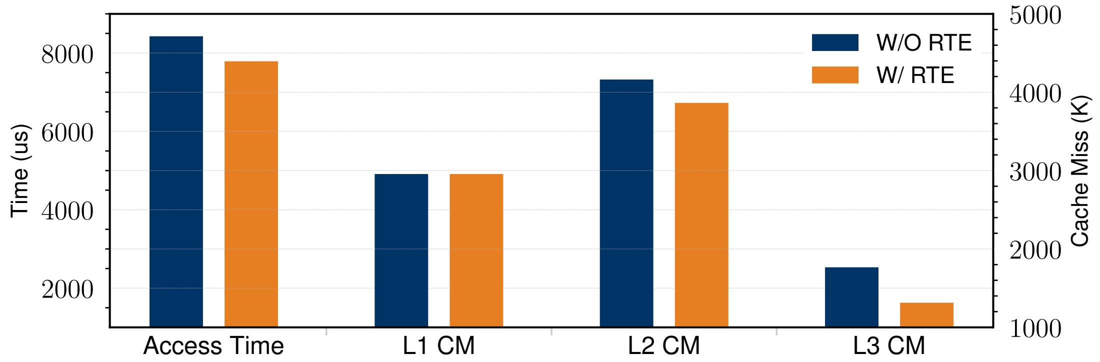


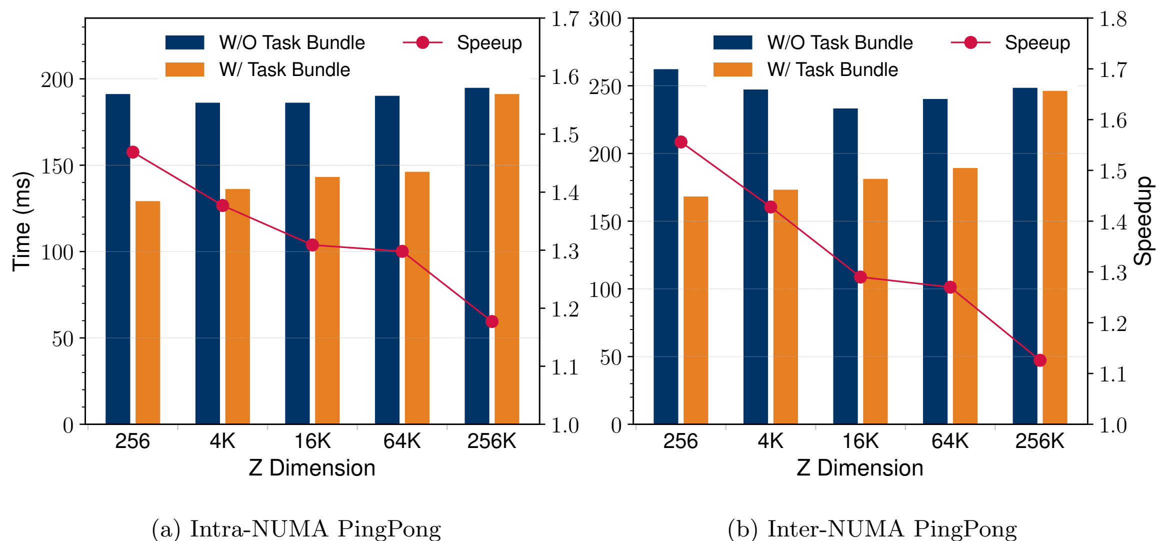
Figure 2.8: Reversed task enqueue (RTE) evaluation based on intra-NUMA PingPong on a Broadwell node. Destination buffer access time, L1, L2, and L3 cache misses are reported.

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 Figure 2.9, 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 2.3.5.

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 Figure 2.10, a small chunk size (e.g., 16 KB, 32 KB) is more 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



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

Figure 2.9: Task bundle performance and speedup in intranode noncontiguous PingPong on Broadwell.

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.

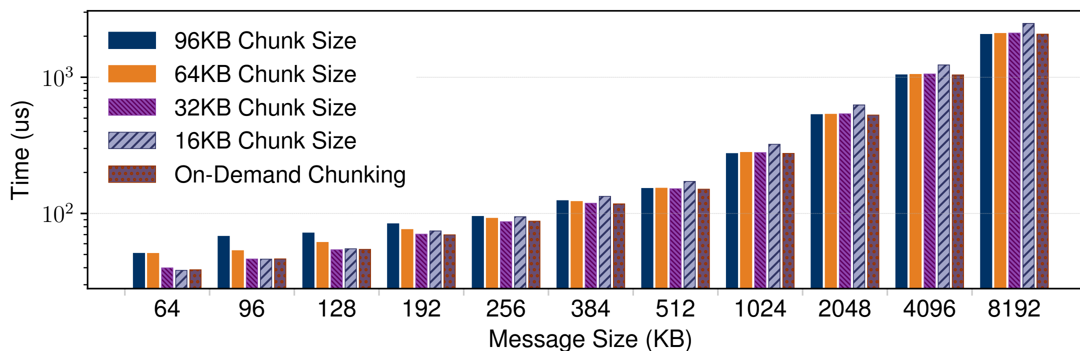


Figure 2.10: On-demand chunking evaluation.

2.5.5 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 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 2.11, 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.

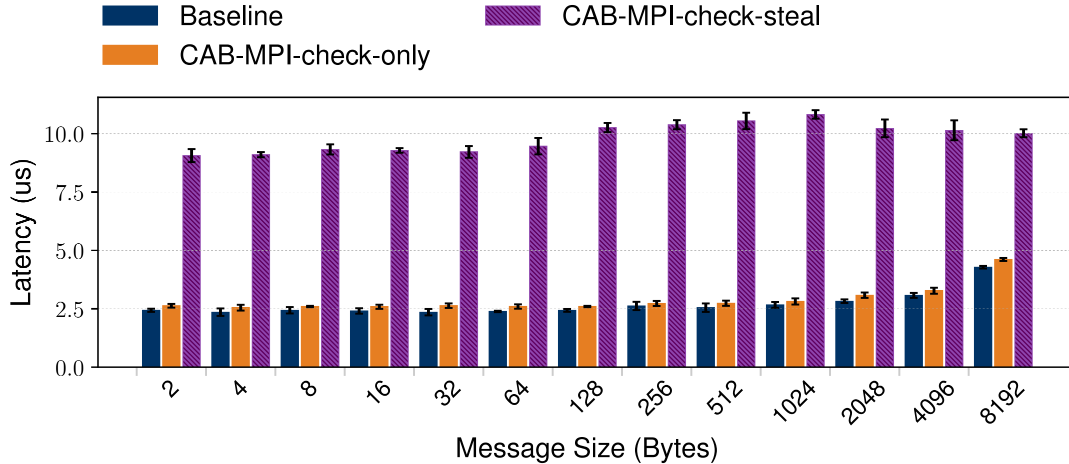


Figure 2.11: Stealing overhead evaluation with small messages by using intra-NUMA Ping-Pong on a Broadwell node with 36 processes. Similar trend is observed on KNL.

2.5.6 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[21] (MPICH-xpmem-coop), OpenMPI using CMA (version 4.0.3, denoted by OMPI-cma), PiP-aware MPI extended from MPICH (baseline)[63], and MPC based on thread-based data sharing (version 3.4.0)[95]. [4] The MPICH options use commit 427cdb07 from the master branch. We compared these approaches with CAB-MPI through the intra-NUMA Ping-Pong test on a single Broadwel node as shown in Figure 2.12. 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.

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

Nevertheless, CAB-MPI improves the performance over all existing approaches by utilizing the local idle processes.

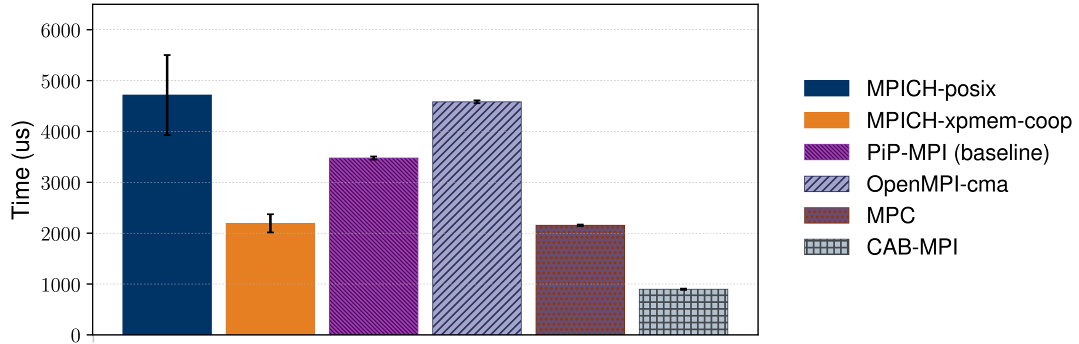


Figure 2.12: Comparison of shared-memory-based optimizations by measuring intra-NUMA PingPong with a fixed message size at 8 MB and fixed number of processes at 36 on a Broadwell node. Only two processes perform PingPong; the others remain idle or behave as workers.

2.6 Application Evaluation and Analysis

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

2.6.1 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 [11]. 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).

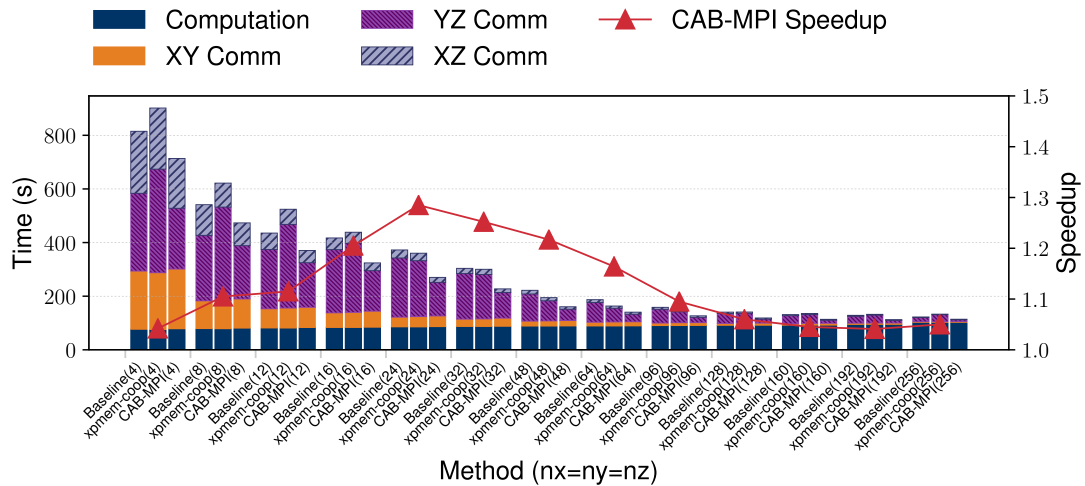


Figure 2.13: MiniGhost with 3D 7-point stencil and BSPMA method running on 576 cores (16 nodes) on Broadwell. The global data size is fixed to 576 GB with varying nx=ny=nz, $nvar$ local parameters; the optimal $(8 \times 8 \times 9)$ process grid is used.

Figure 2.13 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 Figure 2.12, 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 2.14 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.

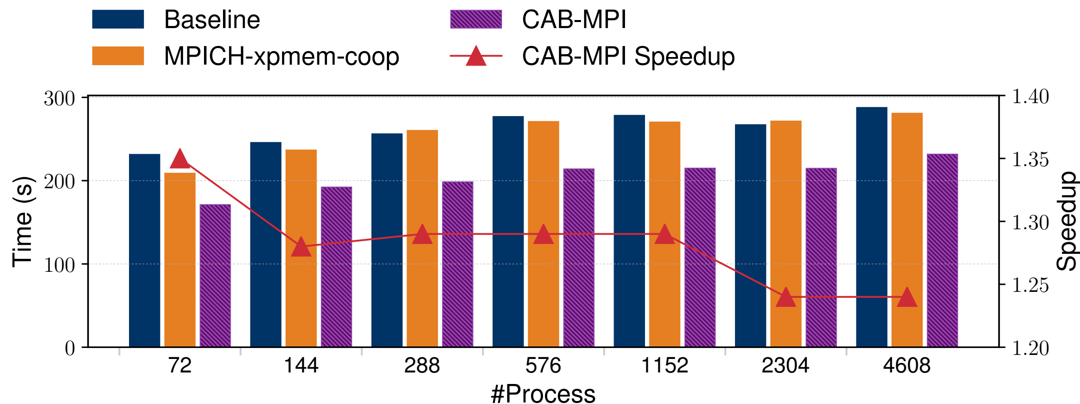


Figure 2.14: Weak-scaling evaluation of miniGhost with 3D 7-point stencil and BSPMA method running on up to 128 Broadwell nodes. Each process uses a fixed set of parameters nx=ny=nz=24, $nvar$ = 9709 (1 GB local data size, and more than 4 TB global data size on 128 nodes).

2.6.2 BSPMM

NWChem [124] is a widely used computational chemistry application suite. NWChem is developed on top of Global Arrays over the MPI one-sided model[88, 35]. 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 Figure 2.15, 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.

2.6.3 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

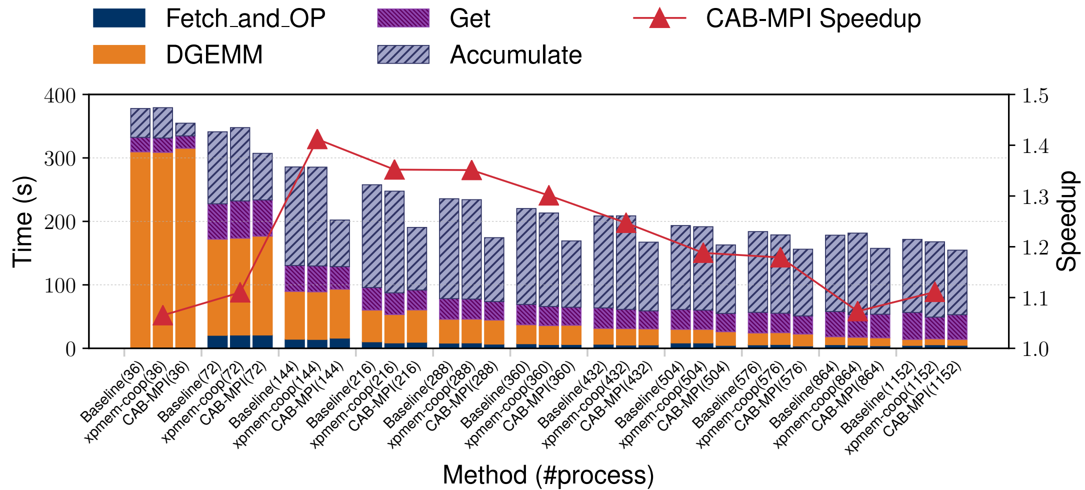


Figure 2.15: BSPMM strong scaling and overhead analysis on Broadwell using global matrix size 102400×102400 and block size 1024 (both in count of double elements).

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.

2.7 Summary

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.

Chapter 3

Daps: A Dynamic Asynchronous Progress Stealing Model for MPI Communication

3.1 Introduction

MPI [47] 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 [84].

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 [69, 97, 123] 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 [109, 110] 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 thesis 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 [4, 103, 120, 91] and MPI asynchronous progress [69, 97, 123, 109, 110] 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 [63] 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 [91], 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 thesis 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 non-blocking 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.

3.2 Background

3.2.1 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 thesis "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.

[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.

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).

3.2.2 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 [17, 60]. Asynchronous progress is the mechanism to ensure that the MPI progress 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 [69, 97, 123], process-based [109, 106], and system interrupt-based [112, 77, 76, 79, 52]. 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.

3.3 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 deter-

mine 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 [124], 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 [52, 110], 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 3.6.2). 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 Sec-

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

tion 4.4). The remaining cores run user processes. Figure 3.1 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.

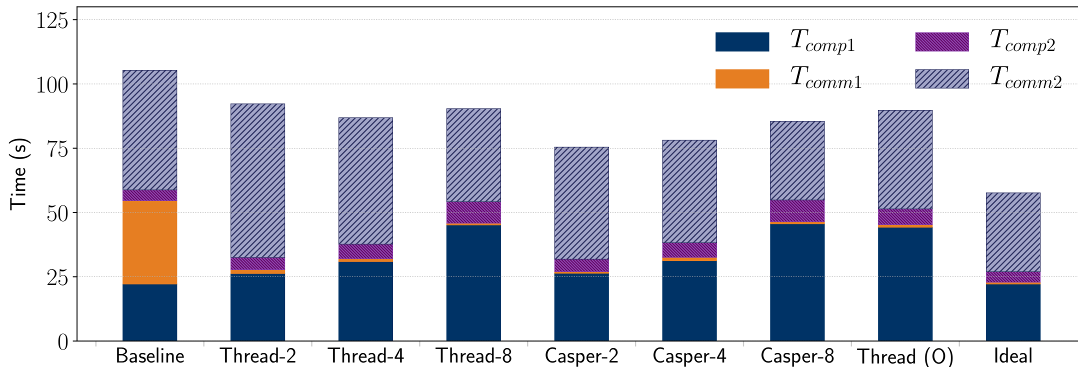


Figure 3.1: Two-stage BSPMM execution time on 4 Broadwell nodes each with 36 cores. The first stage (T_{comp1}+T_{comm1}) is computation intensive, and the second stage (T_{comp2}+T_{comm2}) is communication intensive. The fifure compares the original MPI (Baseline), thread-based asynchronous progress 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.

3.4 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.

3.4.1 Basic Definition

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

Progress Task

Section 3.2.1 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, 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 3.4.3. 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 mem-

ory 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 3.5.

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.

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.

3.4.2 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 [45, 86, 63, 91]. 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 [73], Cross-Memory-Attach (CMA) [74], KNEM [46], XPMEM [58], and Process-in-Process (PiP) [63]. 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 [63, 91] has demonstrated the usage of PiP tasks as MPI processes and optimized the interprocess data transmission via data sharing.

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 3.2 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.

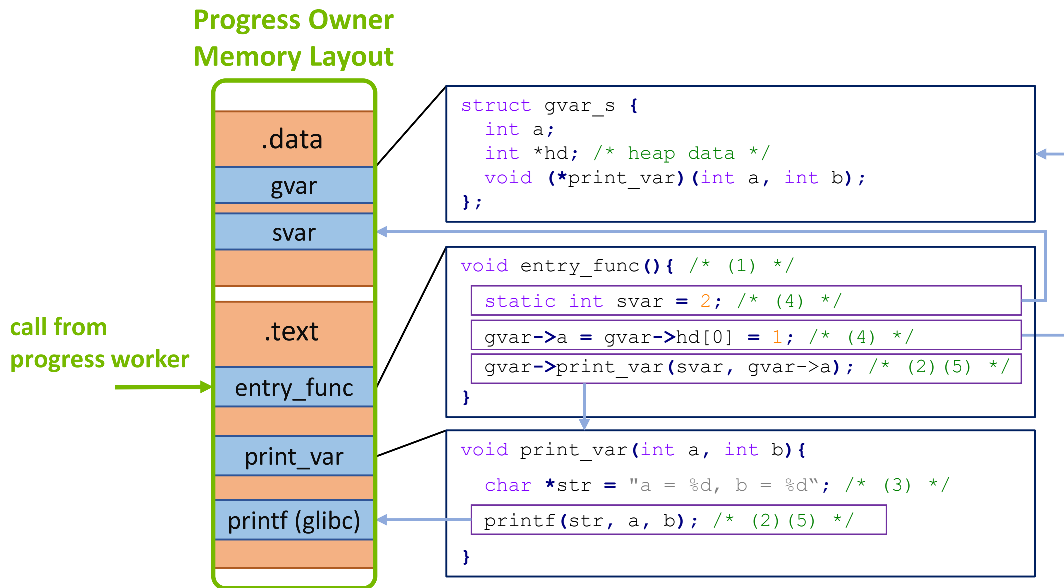


Figure 3.2: Expected behaviors when a progress worker calls into `entry_func()` shared by the owner; the corresponding assessment question ID is marked behind each line.

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 on all processes located in the same PiP VAS. Thus, they can be correctly

[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.

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 [63], and the baseline in the remaining sections refers to PiP-aware MPICH (extended from bb595ca0 of MPICH main branch).

3.4.3 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.

Locality Aware

Similar to previous work-stealing studies [91, 24, 104], 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 [91] discussed locality-aware stealing in detail. We follow a similar approach in Daps and thus omit the locality impact analysis.

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.

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 steal-

ing. *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.

3.5 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 [63, 91]. Nevertheless, in this thesis we systemically diagnose the issues with the current software stack and present solutions.

3.5.1 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 thesis with the open-source Libfabric[1] and Intel Performance Scaled Messaging 2 (Psm2)[65] 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[65, 13]. 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.

One concern may be that the delayed kernel notification may degrade performance. We analyzed the possible scenarios with and without progress stealing and compare

[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.

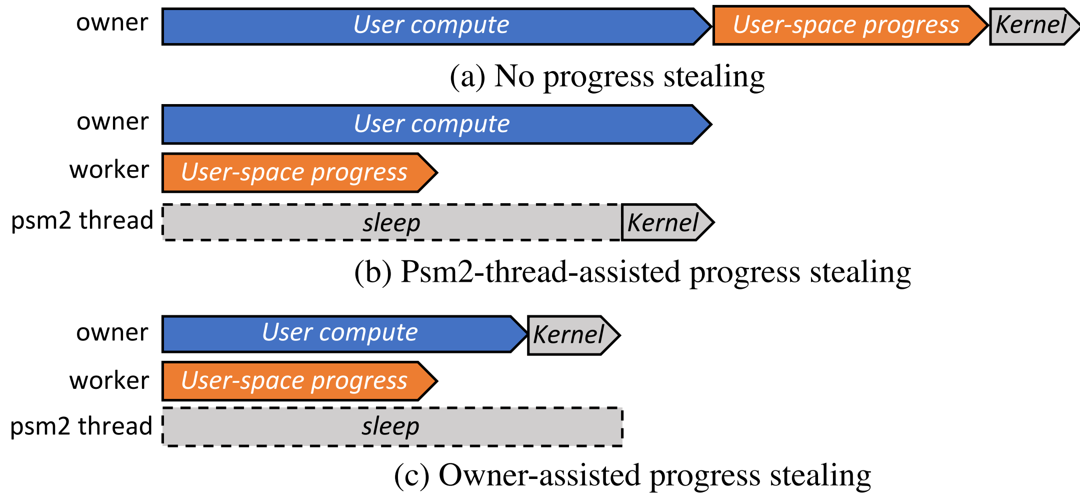


Figure 3.3: Progress-stealing timeline with Psm2 network driver.

the timeline in Figure 3.3. Figure 3.3(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 3.3(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 3.3 (a) with a zero computation). Whenever the user computation is not zero, Daps improves performance.

3.5.2 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 3.4.2). 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.

We survey the usage of TLS in all libraries that may be called in an MPI progress task. Table 3.1 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]). 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.

[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 3.1: 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.

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 [62] 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.

3.6 Experimental Evaluation

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

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

the baseline implementation which includes single-copy based optimization for intranode messages[63]. 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 3.5.

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 3.6.2 and 3.6.3, we also include the dynamic Casper extension [110], 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 [110], 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.

3.6.1 Asynchronous Progress Capability

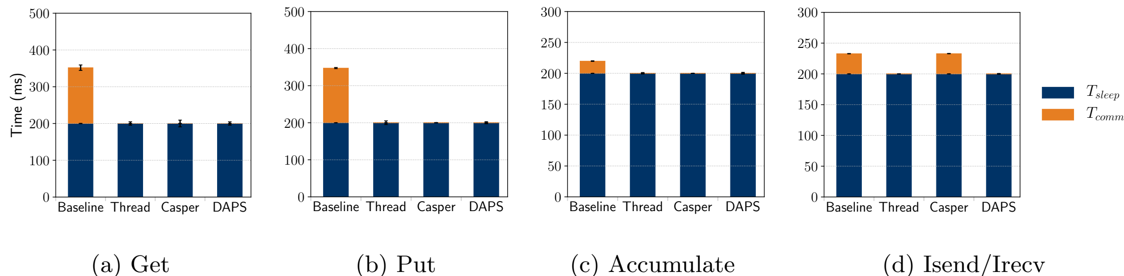
We first design a set of microbenchmarks to ensure the asynchronous progress capability of all mechanisms in internode RMA and point-to-point communication. We employ two communication processes each is launched on a separate node. In our three RMA tests (including Get, Put, and Accumulate), process-1 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-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 3.4 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 3.4d.

3.6.2 Static Progress v.s. Dynamic Progress

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



(a) Get (b) Put (c) Accumulate (d) Isend/Irecv

Figure 3.4: Evaluating asynchronous progress capability with Get, Put, Accumulate, and Isend/Irecv with two internode process.

Custom Two-Stage BSPMM

Block sparse matrix multiplication (BSPMM) is the proxy application of computational chemistry software suite NWchem [124]. 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 [110], 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

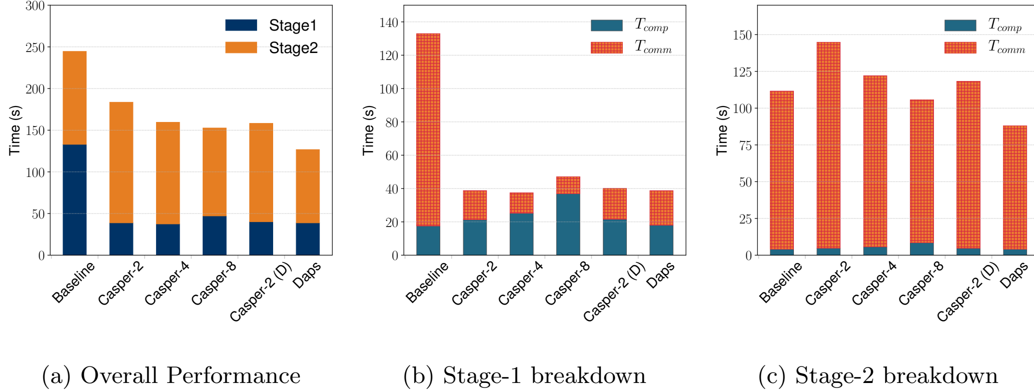
operations, we omit the `fetch_and_op` based global counter update in the original BSPMM which is used for subblock scheduling.

Figure 3.5 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 3.5b), 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 3.5c 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.



(a) Overall Performance (b) Stage-1 breakdown (c) Stage-2 breakdown

Figure 3.5: 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). Stage 1 us computation-heavy, and stage 2 is communication-heavy.

Five-Point 2D Stencil

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 3.6 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

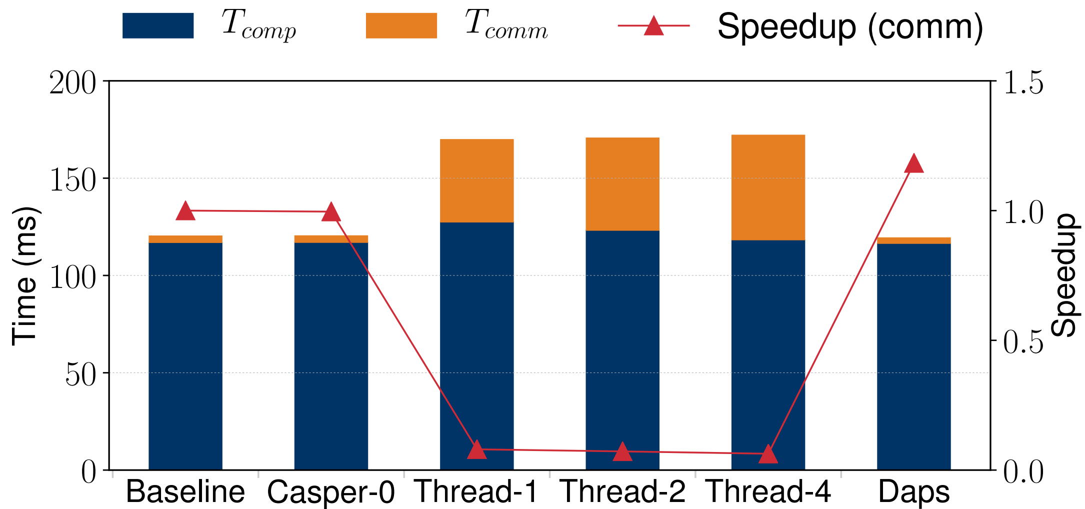


Figure 3.6: Comparing Daps with static progress using five-point 2D stencil with a total problem size of $4096 * 4096$ on 8 Bebop nodes. Showing computation and communication breakdown and the speedup of the communication portion compared to baseline.

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

3.6.3 Scalability

By utilizing the same custom Two-Stage BSPMM as described in Section 3.6.2, 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 3.7. 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.

3.7 Summary

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.

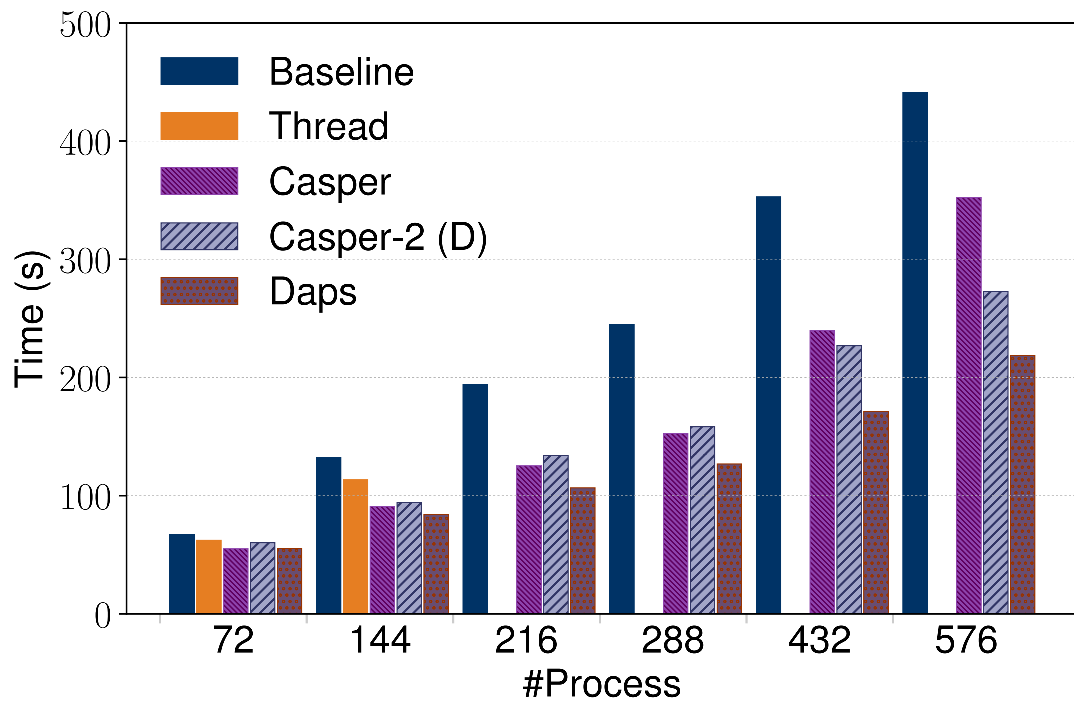


Figure 3.7: Weak scaling evaluation of custom Two-Stage 3D BSPMM over up to 16 Bebop nodes. Comparing Daps with baseline and the best performing configuration of Thread and Casper (Thread with 6 or more nodes cannot be shown due to an MPICH bug, will add upon fix).

Such a method may even perform poorly for multiple-stage applications where each stage often forms a different communication and computation pattern. in this thesis, 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.

Chapter 4

Efficient Process-in-Process based Multiobject Interprocess MPI Collectives for Large-Scale Applications

4.1 Introduction

Message Passing Interface (MPI) collectives have been widely adopted in different research fields [7, 127, 2, 8, 66] because of the low latency and high throughput they deliver on distributed-memory systems. In the era of exascale computing, the number of cores per node and nodes on which MPI applications execute becomes larger than ever before. This scenario brings about higher MPI collective scalability demands; even small- or medium-

message collectives can result in significant communication overhead so that their designs become critical for performance optimization [29].

Generally, MPI collectives consist of intra- and internode communication. Although internode collective communication is usually responsible for most of the overhead and its performance optimization is the major consideration, intranode collective communication overhead is no longer negligible nowadays because of the increased cores per node [67]. Therefore, many methods have been proposed to optimize both intranode and internode collective communication in order to obtain maximal communication performance.

Reduction in shared address space [53] has been proposed to accelerate `MPI_Allreduce` and `MPI_Reduce` communication; it utilizes the interprocess shared-memory (SHMEM) capability of XPMEM [57] to achieve zero-copy intranode reduction. On the other hand, kernel-assisted interprocess data copy techniques such as LiMiC [34], KNEM [86], and Cross Memory Attach (CMA) [22] are also adopted to speed up intranode communication of `MPI_Allgather`, `MPI_Scatter` and many other MPI collectives. Although these methods can bring faster intranode communication and more optimized collective algorithms, the benefits are significant only when the message size is large enough because they can degrade performance for small- or medium-message collective performance due to the expensive system call and page fault overhead. Parsons et al. [92] demonstrate the efficient MPI collective algorithms with a POSIX shared-memory (POSIX-SHMEM) [75] multisender design. Although the multisender brings about better network bandwidth utilization, POSIX-SHMEM limits the efficiency of algorithms that cannot achieve high performance for large-message collective communication because of the double copy overhead that inherently resides in it.

To solve these issues, we propose a *Process-in-Process–based multiobject interprocess MPI collective (PiP-MColl)* design that maximizes intra- and internode message rate and network throughput and eliminates the drawbacks of existing collective algorithms for small- and medium-message MPI collectives. Under the Process-in-Process [63] environment, all processes on a node are loaded into the same virtual memory space so that they can access the private memory of each other like threads in userspace. PiP-MColl utilizes the features of PiP to avoid extra data copy and expensive system-related overhead so that it is able to deliver higher message rate and network throughput. More important, the MPI collective algorithms must be carefully redesigned to provide high parallelism for both intra- and internode communication and mitigate the impact of the potential overhead (e.g., process synchronization), which becomes the most challenging part of our work.

With newly designed multiobject (i.e., multiple senders and receivers) collective algorithms, PiP-MColl is able to saturate the message rate and bandwidth by sending/receiving multiple messages in parallel without causing extra overhead. In addition, overlapping-supported PiP-MColl can hide the intranode communication under internode communication; it helps bring better network bandwidth usage for medium- and large-message cases.

We apply PiP-MColl to the well-known MPICH library and show the benefits for `MPI_Scatter`, `MPI_Allgather`, and `MPI_Allreduce`, three widely used MPI collectives. We measure the performance of collective microbenchmarks and a real-world HPC application N-body with various message sizes and execution scales (up to 256 Xeon Broadwell nodes) and compare with the baseline implementation PiP-MPICH and three other widely used

MPI libraries: Intel-MPI, OpenMPI, and MVAPICH2. The results show PiP-MColl is able to provide significant message rate and network throughput for small- and medium-message MPI collectives.

4.2 Background

This section provides background information about the existing widely adopted shared-memory techniques in MPI collective designs.

MPI collectives mainly involve intra- and internode communication. Since the number of cores per node has reached up to tens and even hundreds, the overhead of intranode collective communication no longer is negligible. In order to optimize the performance of intranode collectives, different shared-memory techniques are adopted.

4.2.1 POSIX Shared Memory

POSIX-SHMEM is natively supported by the Linux kernel and widely adopted in MPI design because of its portability and efficiency. To exchange data, processes must collectively allocate shared-memory buffers through system calls; senders copy user data into the shared-memory buffer, and receivers copy data out into the receive buffer. This exchange mechanism brings fast communication when the message size is small since process synchronization is not required; however, for medium- and large-message communication, it results in double data copy, which causes lower performance compared with other kernel-assisted shared-memory techniques.

4.2.2 Data Exchange Based Shared Memory

LiMiC and KNEM work in similar ways; they are both Linux kernel modules that support direct data copy from one process to another. Each LiMiC- and KNEM-based communication involves system calls to operate user-level buffer segments through the kernel. Senders register the user-level buffer in the kernel space, obtain a kernel-created buffer key, and send the key to receivers; on the other hand, receivers get the key and retrieve the data through the kernel system call. The CMA mechanism has been integrated into the Linux kernel where senders and receivers can exchange data through `process_vm_writev` or `process_vm_readv`. Although CMA provides a simple and native way for interprocess data exchange functionality, it still involves system calls whenever a data transmission happens.

LiMiC, KNEM, and CMA are all designed for *data exchange* instead of *data sharing*. They are not able to avoid unnecessary data copy during collective communication. For example, in `MPI_Allreduce`, each process must exchange data with the other processes before performing reduction. This data exchange results in extra data copy that can be avoided by *data sharing*.

4.2.3 Data Sharing Based Shared Memory

XPMEM is also a Linux kernel module but supports data sharing among processes in the userspace. Senders can expose the user buffer in the beginning, and receivers attach the buffer in their own address space and perform the data exchange. XPMEM allows receivers to access private buffers of senders without extra copy required with LiMiC, KNEM, and CMA, thus bringing about huge benefits for the reduce-based MPI communi-

cation. Similarly, XPMEM requires system calls for buffer expose and attachment so that it usually benefits large-message communication.

4.2.4 Userspace Address Space Sharing

Process-in-Process (PiP) is a userspace shared address space technique that does not need system calls in order to achieve interprocess data exchange. In the PiP environment, MPI processes are loaded into the same virtual memory space. Each process has its own separate context (e.g., static variables) but can access the private data of other processes like threads. No system call overhead is involved throughout the communication. Unlike the conventional shared memory techniques, this shared address space technique allows us to access any data structures including pointers. This feature brings the possibility to deliver optimal performance for all collective communication sizes compared with other shared-memory techniques.

In the following, we present the design of the MPI collective multiobject algorithms in the PiP environment.

4.3 Multiobject Interprocess MPI Collective Design

In this section, we first analyze the factors which affect message rate and network throughput to support our multiobject (i.e., multiple sender and receiver) design; secondly, we explain the communication cost model used for the theoretical performance analyses; thirdly, we detail the PiP-based multiobject collective algorithm designs for three widely used MPI collectives `MPI_Scatter`, `MPI_Allgather` and `MPI_Allreduce`.

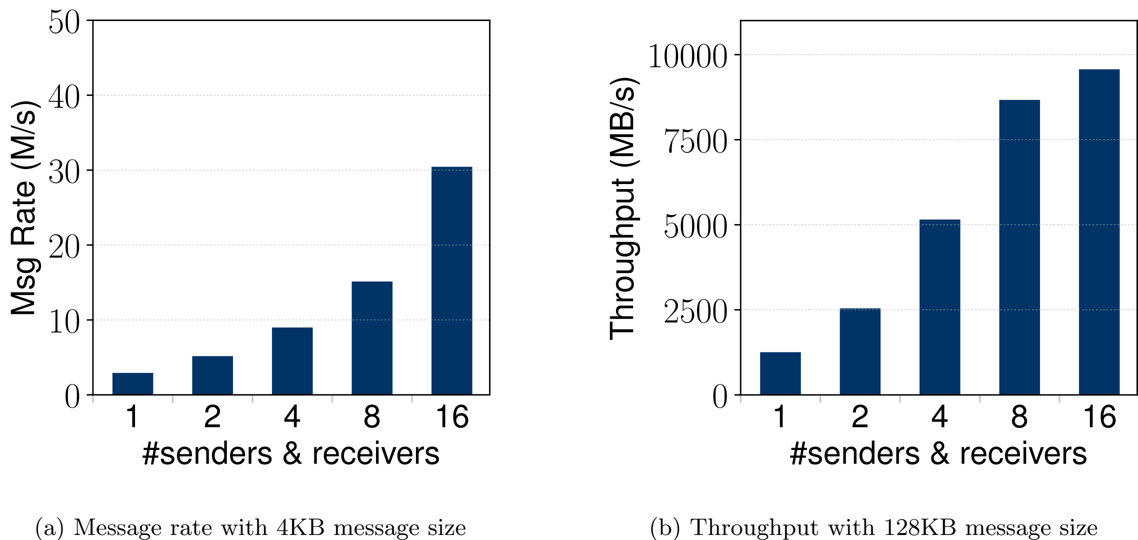
4.3.1 Message Rate and Network Throughput

Message rate and network throughput are two important metrics to evaluate the efficiency of MPI communication. For small-message communication, higher message rate means better parallelism; for medium- or large-message communication, higher network throughput means better network bandwidth utilization.

For large-message communication, the network bandwidth can usually be saturated by one process; this implies internode collective performance should be improved in algorithm level to reduce overall communication volume. However, for small- and medium-message communication, the data transmission is hard to saturate the corresponding hardware (i.e., memory and network interface card) by only one process.

To show the feasibility of the multiobject design, considering widely deployed network interconnect Intel Omni-Path [14], we show the performance of point-to-point communication with 4KB and 128 KB message size and various pairs of senders and receivers locating on two separate nodes in Figure 4.1.

The Figure 4.1 proves that if there are more senders to issue messages out and receivers to receive the data in parallel, we are able to get higher message rate and better network throughput due to better hardware utilization. The results build the foundation which motivate us to choose as many objects as possible in PiP-MColl for maximal collective communication performance.



(a) Message rate with 4KB message size

(b) Throughput with 128KB message size

Figure 4.1: The inter-node message rate and network throughput of 4KB and 128KB point-to-point with various pairs of senders and receivers locating on two separate nodes. The network interconnect is Intel Omni-Path.

4.3.2 Communication Cost Model

The Hockney cost model [59] $\alpha + M * \beta$ is a simple and frequently used model for data transmission performance; α represents the start-up latency per message, M is the size of message in byte; and β stands for the transmission speedup (s/byte). Since we focus on MPI collectives with intra- and internode codesigned algorithms, to more precisely show the theoretical performance, we extend this model to include both intra- and internode communication runtime.

We define α_r and α_e as intra- and internode start-up latency; β_r and β_e stand for intra- and internode transmission time per byte, respectively; γ is reduction speed (s/byte); M is message size in byte; P is the number of processes on a node; and N is the number of nodes on which MPI collectives execute.

Symbol	Meaning
α_r	intranode start-up latency
β_r	intranode transmission speed (s/byte)
P	#processes per node
M	message size (byte)
C_b	data size per process
N_{id}	node id
B_k	Bruck algorithm base
A_s	source buffer address
Symbol	Meaning
α_e	internode start-up latency
β_e	internode transmission speed (s/byte)
N	#nodes
γ	reduction speed (s/byte)
R_l	process local rank
N_r	root process node id
S_p	Bruck algorithm step
A_d	destination buffer address

Table 4.1: Summary of MPI collective communication cost model symbols.

The symbols mentioned above and those defined in the following section are summarized in Table 4.1.

4.3.3 PiP-MColl based Collective Algorithm Design

MPI_Scatter

In `MPI_Scatter`, the global root process designated by the user will evenly scatter the data to other processes. To implement this functionality, traditional MPI usually chooses a binomial tree algorithm [105] where only one pair of sender and receiver per node is selected for internode data exchange. In contrast, PiP-MColl selects all processes on a node as senders and receivers to scatter and receive data among nodes in parallel for all message sizes (i.e., small, medium, and large messages). In the PiP-MColl `MPI_Scatter`

design, we define the local rank of a process on a node as R_l ranging from 0 to $P-1$. For description convenience also we assume that the user-designated global root process is a local root process. N is the power of $P+1$; C_b is the number of bytes each process should receive into the destination buffer in scatter; and N_{id} is the node id ranging from 0 to $N-1$.

Figure 4.2 shows the high-level design of PiP-MColl `MPI_Scatter` with overlapped intranode scatter. The corresponding algorithm can be described as follows.

Step 1: Share sending buffer address. For the global root node or nodes that just receive the data, if the local root process has not shared the source buffer address A_s, the local root process posts the address to all processes on the node and proceeds to next step.

Step 2: Asynchronously scatter data through network. Each process on a node with data finds the paired process with global rank $((R_l+1)*\frac{N}{P+1}+N_{id})*P$; then, it asynchronously sends data from address $A_s+(R_l+1)*\frac{N}{P+1}*C_b*P$ with $\frac{N}{P+1}*C_b*P$ bytes.

Step 3: Perform intranode scatter. For all processes on a node containing data, each process finds offset address $A_s+R_l*C_b$ and copies C_b bytes into the receiving buffer.

Step 4: Wait until internode scatter completes. Each process waits until its own internode sending requests issued in Step 2 complete; the paired processes wait until receiving all data.

Step 5: Recursively execute Step 1 to Step 4. Update $N=\frac{N}{P+1}$. If $N==1$, the algorithm completes; otherwise, go back to Step 1.

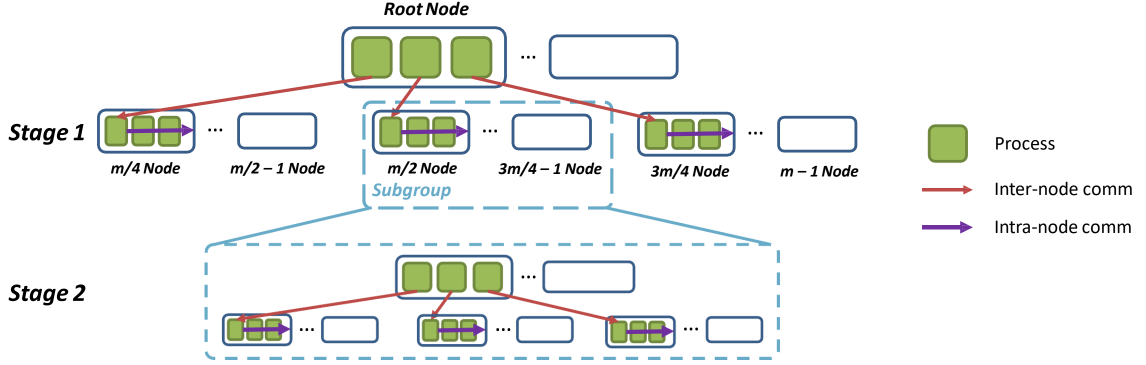


Figure 4.2: High-level design of PiP-MColl MPI_Scatter with overlapped intranode scatter. Shown as an example are 3 senders and 1 receiver on a node. The figure shows only the partial stages; the rest of the stages will repeat the same operations in stage 1 for each new generated subgroup.

Considering the theoretical runtime analyses, we can present the intranode scatter runtime as Equation 4.1 and the internode scatter runtime as Equation 4.2. Since our algorithm overlaps intra- and internode scatter, the overall runtime can be summarized as $T = Max(T_{intrascatter}, T_{interscatter})$.

$$T_{intrascatter} = \alpha_r + P * C_b * \beta_r \tag{4.1}$$

$$T_{interscatter} = \alpha_e * \lceil log_{P+1}N \rceil + C_b * (N-1) * P * \beta_e \tag{4.2}$$

MPI_Allgather

Generally, different algorithms are adopted for `MPI_Allgather` based on the communication message size. Traditionally, for small messages, Bruck algorithm [18] can be adopted when the number of processes is a non-power of two, and recursive doubling algorithm [116] can be adopted for power-of-two cases; for medium and large messages, a

ring algorithm is applied to achieve minimal internode communication volume. To further improve allgather performance, we design two PiP-MColl allgather algorithms for different message sizes. Figure 4.3 illustrates a high-level overview of the PiP-MColl allgather algorithm for small message cases, which can be described as follows.

Step 1: Perform intranode gather to local root process. Local processes perform `MPI_Gather` to gather data into the local root process destination buffer A_d.

Step 2: Initialize parameters. The multiobject Bruck algorithm step is initialized as $S_p = 1$ and the base of the multiobject Bruck algorithm as $B_k = P + 1$.

Step 3: Find the paired source and destination process. Each process sets $N_{offset} = (R_l + 1) * S_p$ and finds the paired source node $N_{src} = (N_{id} + N_{offset})\%N$ and destination node $N_{dst} = (N_{id} - N_{offset})\%N$. The paired source process rank is $N_{src} * N + R_l$, and the destination process rank is $N_{dst} * N + R_l$.

Step 4: Perform send and receive. We define C_b as the receiving bytes from each process in allgather and A_d as the starting address of the destination buffer of the local root process. Each process sends $C_b * S_p$ bytes from the local root process buffer to destination process and receives $C_b * S_p$ bytes from the source process into address $A_d + C_b * S_p * (R_l + 1)$. Then, each process updates $S_p = S_p * B_k$. If S_p is smaller than or equal to $\frac{N}{B_k}$, we repeat Step 3 to Step 4; otherwise, we go to Step 5.

Step 5: Deal with the remainder. If N is not a power of B_k, we have the remaining $N - S_p$ nodes for the final step. Each process takes $Rem = Max(Min(S_p, N - S_p * R_l), 0)$ remainder; if $Rem > 0$, the process will send and receive the $Rem * C_b$ bytes to and from the paired destination and source process.

Step 6: Shift data and broadcast. Local root process shifts the data into correct sequence [18] and broadcasts to other processes.

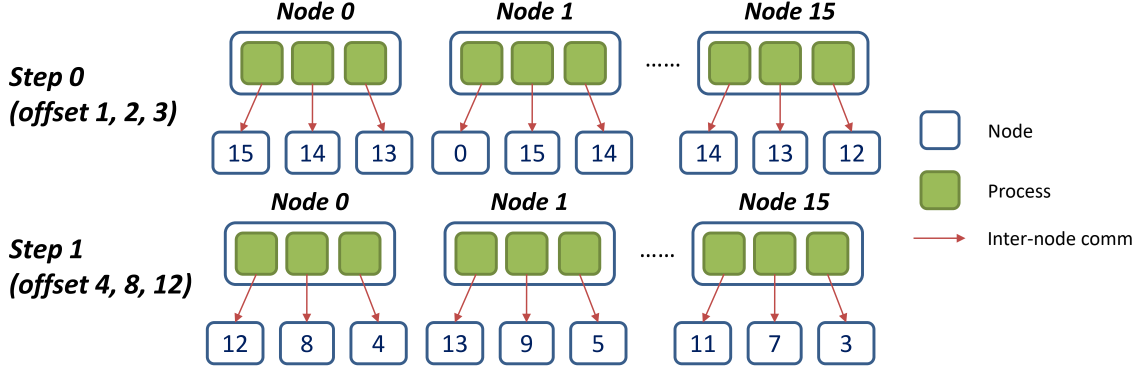


Figure 4.3: Example of two-step PiP-MColl allgather algorithm for small-message communication with 16 nodes and 3 objects per node.

Based on the small-message allgather algorithm mentioned above, we provide Equation 4.3, which shows the intranode gather runtime, and Equation 4.4, which shows the internode allgather runtime [1]. Since we do not overlap intra- and internode communication in this case, the overall runtime is $T = T_{intra-gathers} + T_{inter-allgathers}$.

$$T_{intra-gathers} = \alpha_r + (1 + N * P) * (P - 1) * C_b * \beta_r \tag{4.3}$$

$$T_{inter-allgathers} = \alpha_e * \lceil log_{P+1} N \rceil + (C_b * P - 1) * C_b * P * \beta_e \tag{4.4}$$

On the other hand, for medium- and large-message allgather, we adopt the multi-object ring algorithm to maximize network bandwidth utilization.

[1] Since the bandwidth term is not as important as the latency term when the message size is small, to simplify the equation, we omit the remainder processing overhead. This is the same for the MPI_Allreduce runtime analyses.

Figure 4.4 provides an overview of the algorithm. The detailed procedures can be described as follows.

Step 1: Intranode gather to local root process. Similar to the small-message algorithm, all processes on a node gather data into the destination buffer A_d of the local root process.

Step 2: Identification of paired processes and parameter initialization. Each process finds left source node $N_{src} = (N_{id} - 1)\%N$ and right destination node $N_{dst} = (N_{id} + 1)\%N$; then the paired source process rank is $P * N_{src} + R_l$, and the destination process rank is $P * N_{dst} + R_l$. Each process then sets $N_{doffset} = N_{id}$ and $N_{soffset} = N_{src}$ for node reference.

Step 3: Multiobject send and receive in ring pattern. Each process sets the address offset $A_{doffset} = C_b * R_l + N_{doffset} * P * C_b$ and $A_{soffset} = C_b * R_l + N_{soffset} * P * C_b$; then each process asynchronously sends C_b bytes starting from $A_d + A_{doffset}$ to the destination process and receives C_b bytes in $A_d + A_{soffset}$.

Step 4: Overlapped intranode broadcast. The local root process broadcasts $P * C_b$ bytes at address $A_d + A_{doffset}$ to other processes on the node.

Step 5: Parameter check and update. If the ring communication step is smaller than $N - 1$, every process updates $N_{doffset} = N_{soffset}$, $N_{soffset} = (N_{soffset} - 1)\%N$ and jumps back to Step 3; otherwise, the algorithm completes.

Similarly, medium- and large-message allgather involves an intranode gather, an overlapped intranode broadcast, and an internode multiobject ring communication. The intranode gather and broadcast runtime are shown in Equation 4.5 and 4.6 (the detailed

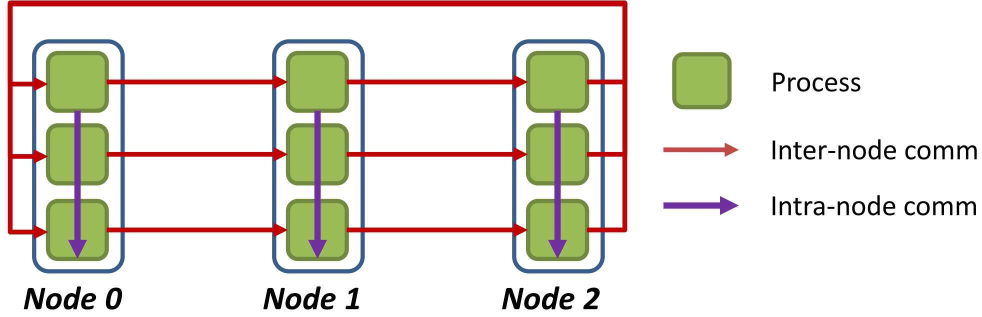


Figure 4.4: High-level PiP-MColl allgather algorithm for medium- and large-message size. The figure shows an example with 3 nodes and 3 objects per node and one step of the ring communication. The intra- and internode communications run in parallel for overlapping.

gather and broadcast algorithm design are explained in Section 4.3.4); the internode runtime is presented in Equation 4.7.

The overall runtime is $T = T_{intra-gatherl} + Max(T_{intra-bcastl}, T_{inter-allgatherl})$.

$$T_{intra-gatherl} = \alpha_r + (P-1) * C_b * \beta_r \tag{4.5}$$

$$T_{intra-bcastl} = \alpha_r * (N-1) + (P-1) * N * P * C_b * \beta_r \tag{4.6}$$

$$T_{inter-allgatherl} = \alpha_e * (N-1) + P * C_b * (N-1) * \beta_e \tag{4.7}$$

MPI_Allreduce

Traditionally, allreduce adopts a recursive doubling algorithm [116] for small messages and Rabenseifner's algorithm [98] for large messages, which performs a reduce-scatter followed by an allgather. To maximize performance, we design the recursive PiP-MColl Bruck algorithm for small-message allreduce. The part of the design in allreduce is similar

to allgather, but allreduce requires an extra reduction operation after each data transmission. In addition, when the number of nodes is not a power of $P+1$, we need to compute the reduction results of the reminder recursively. C_b bytes are for reduction. The algorithm is designed as follows.

Step 1: Intranode reduce. All processes on a node perform intranode binomial reduce and store final results into destination buffer A_d of the local root process.

Step 2: Parameter initialization All processes set S_p as 1 and B_k as $P+1$, which is the base of the multiobject Bruck algorithm.

Step 3: Identification of paired source and destination process. Each process assigns $N_{offset} = (R_l + 1) * S_p$ and finds the paired source node $N_{src} = (N_{id} + N_{offset})\%N$ and destination node $N_{dst} = (N_{id} - N_{offset})\%N$. The paired source process rank is $N_{src} * N + R_l$, and the destination process rank is $N_{dst} * N + R_l$.

Step 4: Send, receive, and reduce. Each process sends C_b bytes from the local root process buffer A_d to the destination process, receives C_b bytes from the source process in a temporary buffer, and performs an intranode reduce.

Step 5: Handling of current stage remainder. Every process sets $S_p = S_p * B_k$. If $Rem = N\%S_p$ is not zero, we need to perform an intranode reduction for the reminder. If $S_p == B_k$, then Rem number of processes perform intranode reduction using the received data and store the results in a new remainder buffer A_r; if $S_p > B_k$, then $\lceil \frac{Rem*B_k}{S_p} \rceil$ number of processes perform intranode reduction using the received data and previous reminder results and store the results in a new reminder buffer A_r. If S_p is smaller than or equal to $\frac{N}{B_k}$, go back to Step 3; otherwise, go to Step 6.

Step 6: Final stage for remainder. If N is not a power of B_k, each process sets $Rem = Min((N - S_p) - R_l * S_p, S_p)$. If $Rem > 0$ and $Rem == S_p$, the process sends C_b bytes from A_s to the destination process and receives C_b bytes from the source process. On the other hand, if $Re > 0$ and $Re \neq S_p$, the process sends C_b bytes from A_r to the destination process and receives C_b bytes from the source process. The processes with $Re > 0$ on the node perform an intranode reduce.

Step 6: Broadcasting of results. The local root process broadcasts the global reduction results to all processes on the node, and the algorithm completes.

The small-message allreduce contains an intranode reduce whose runtime can be represented as Equation 4.8 and an internode allreduce whose runtime is shown in Equation 4.9. The overall runtime is $T = T_{intra-reduces} + T_{inter-allreduces}$.

$$T_{intra-reduces} = \alpha_r * \lceil log_2 P \rceil + C_b * \lceil log_2 P \rceil * \beta_r + C_b * \lceil log_2 P \rceil * \gamma \quad (4.8)$$

$$T_{inter-allreduces} = \alpha_e * \lceil log_{P+1} N \rceil + C_b * P * \lceil log_{P+1} N \rceil * \beta_e + C_b * \lceil log_{P+1} N \rceil * \gamma \quad (4.9)$$

For medium- and large-message allreduce, PiP-MColl performs a multiobject reduce-scatter followed by PiP-MColl based allgather. We assume N is divisible by P, C_b is divisible by N, and the data chunk size is S_c The algorithm is described as follows.

Step 1: Perform intranode reduce. All processes on a node perform intranode reduce (the detailed algorithm is explained in Section 4.3.4) and store final results in the local root process destination buffer.

Step 2: Post buffer address. Each process posts the source data buffer address to all other processes on the node and gets the local root process destination buffer address A_d.

Step 3: Find paired node range and process. Each process finds the paired node range from $\frac{N*R_l}{P}$ to $\frac{N*(R_l+1)}{P}$. For a paired node N_p, the paired destination process rank is $N_p * P + R_l$.

Step 4: Perform internode reduce-scatter. For each paired node, a process finds the data chunk starting from $A_d + \frac{C_b*N_p}{N}$ to $A_d + \frac{C_b*(N_p+1)}{N}$ and sends it to the paired process. Then, if $N_p == N_{id}$, the process receives $N-1$ chunks from the paired source processes and reduce in the corresponding chunk; otherwise, it sends the chunk to the paired destination process.

Step 5: Perform internode allgather with intranode broadcast. After Step 4, each node owns the partial results of allreduce. All processes need to perform internode allgather followed by intranode broadcast to obtain the complete global results, and the algorithm completes.

$$T_{intra-reducel} = \alpha_r * (P-1) + C_b * P * \gamma \tag{4.10}$$

$$T_{inter-rscatterl} = \alpha_e * (P-1) + \frac{(N-1)*C_b}{N} * \beta_e + \frac{C_b}{N} * (N-1) * \gamma \tag{4.11}$$

The algorithm contains intranode reduce, internode reduce-scatter, internode allgather, and intranode broadcast. Among them, internode allgather and intranode broadcast have been analyzed in Section 4.3.3. Intranode reduce and internode reduce-scatter

runtime can be represented in Equation 4.10 and 4.11. The overall runtime should be $T = T_{intra-reducel} + T_{inter-rscatterl} + Max(T_{intra-bcastl}, T_{inter-allgatherl})$.

4.3.4 Auxiliary MPI Collectives

For the algorithms presented above, the MPI collectives we focus on are `MPI_Scatter`, `MPI_Allgather`, and `MPI_Allreduce`. However, other auxiliary intranode MPI collectives—including `MPI_Bcast`, `MPI_Gather`, and `MPI_Reduce`—are designed as the building blocks of the three MPI collectives in focus here. In the following, we briefly talk about the designs of the auxiliary MPI collectives in the PiP environment.

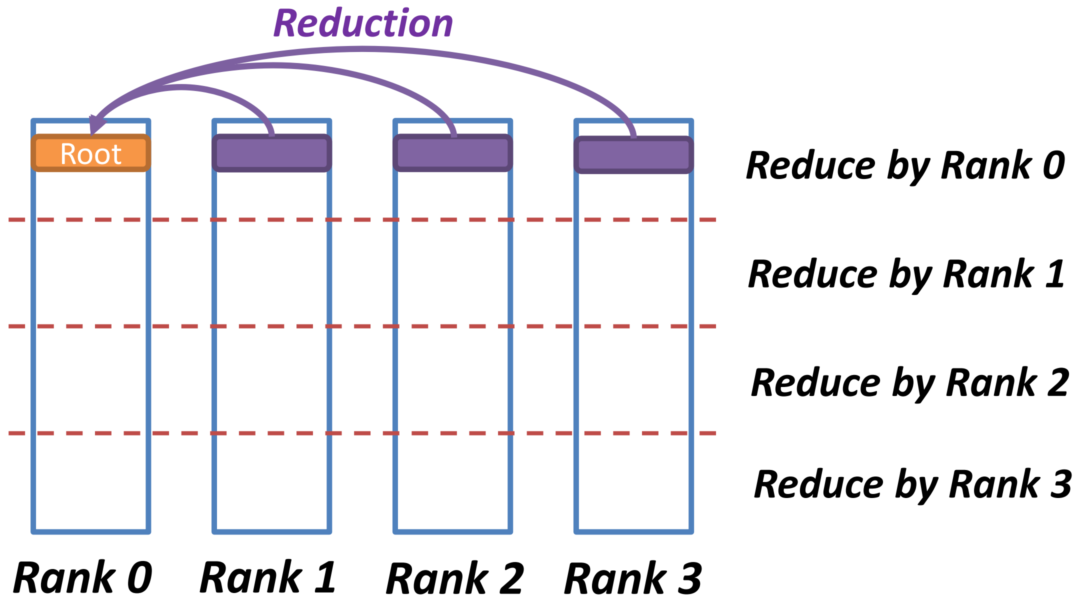


Figure 4.5: PiP-MColl–based large-message intranode reduce communication with 4 processes on a node. Each buffer is chunked based on the number of processes, and all data will be reduced into the root process destination buffer.

`MPI_Bcast` is a one-to-all operation where one root process broadcasts the data to all processes in the group. For small-message broadcast, the root process first copies the source data into a temporary buffer; next it posts the address of the temporary buffer to all processes, which then copy data into their destination buffer. For large-message broadcast, the root process posts its source buffer address to all processes in the beginning; then the processes copy data into their destination buffer. In this case, the root process needs to wait until all processes complete the data copy.

`MPI_Gather` is an all-to-one operation where one root process gathers data from all processes in the group. The same algorithm is applied to both small- and large -message communication. The root process first posts its destination buffer to all processes; then every process copies the source data into the designated position in the buffer. In this case also, the root process needs to wait until all processes complete the data copy.

For `MPI_Reduce`, for small messages we simply adopt the binomial algorithm for reduction, For large messages, the root process posts its destination buffer, and every process posts its source data buffer; then each process is responsible for a chunk of buffer reduction. That is, if there are N processes, every posted buffer is evenly cut into N chunks, and process i will reduce the ith chunk from all source data buffers into the ith chunk of the destination buffer. Figure 4.5 shows the reduction pattern. Correspondingly, the root process needs to wait until all processes complete the data reduction.

4.4 Experimental Results

In this section we first show the large-scale microbenchmark performance of `MPI_Scatter`, `MPI_Allgather`, and `MPI_Allreduce`; next, we compare PiP-MColl with the other widely used MPI libraries:, namely, Intel MPI, MVAPICH2, and OpenMPI; we then measure the performance of a real-world application N-body problem to further prove the effectiveness of PiP-MColl.

4.4.1 Experimental Setup

We perform all experiments with a 256-node cluster with 18 processes on each node.[2] Each node contains two Intel Xeon E5-2695v4 Broadwell processors with 36 cores in total, and each NUMA node attaches 64 GB DDR4 memory locally, which amounts to 128 GB of memory on a node. The nodes are connected via the Intel OPA interconnect with maximal 97 Mpps (million per port per second) message rate and 100 Gbps bandwidth. Hyperthreading is disabled on all nodes. We use PiP-based MPICH (extended from commit bb595ca0 of the MPICH main branch) as the baseline implementation, which includes single-copy–based optimization for medium and large intranode message communication [63]. 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 Glibc libraries are patched in order to support PiP task spawn. For the MPI library comparison, we use Intel-MPI (version 2017.3), OpenMPI (version 4.1.2), and MVAPICH2 (version 2.3.6) to compare with PiP-MColl using `MPI_Allgather` communication.

[2] in this thesis we focus mainly on high-performance multiobject collective algorithm design instead of hierarchical design [82]. Therefore, we ignore the NUMA feature and use only one socket per node for performance showcase. However, the work can be easily extended to NUMA-aware version.

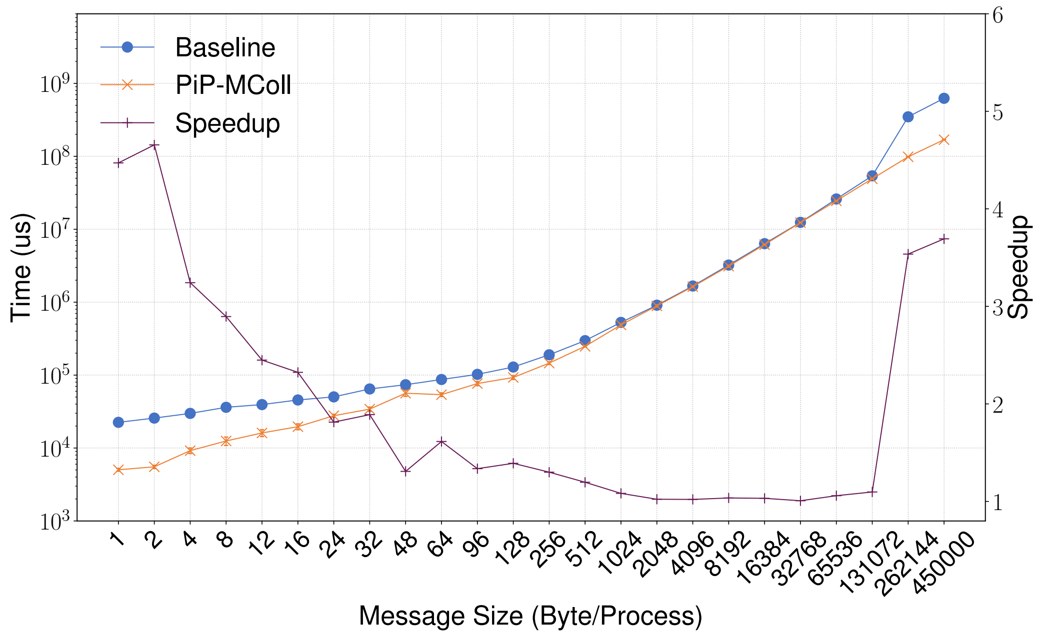
4.4.2 Microbenchmark Evaluation

In this section we present the microbenchmark performance of `MPI_Scatter`, `MPI_Allgather`, and `MPI_Allreduce`. The microbenchmark is designed in two stages: a warm-up stage and an execution stage. Each stage runs the same number of iterations. For small messages (i.e., 1 B–1 kB), we run 10,000 iterations and compute the average time per iteration as the final runtime; for medium messages, we run 1,000 iterations for 1–8 kB and 100 iterations for 8–128 kB; for large messages (i.e., 128 kB or more), we run 10 iterations. All microbenchmarks are repeatedly performed in 10 rounds in order to measure their standard deviation. The main focus of PiP-MColl is to optimize small and medium message-size MPI communication, where we expected the speedup. However, we also present the performance of large-message collectives to prove the feasibility and effectiveness of PiP-MColl.

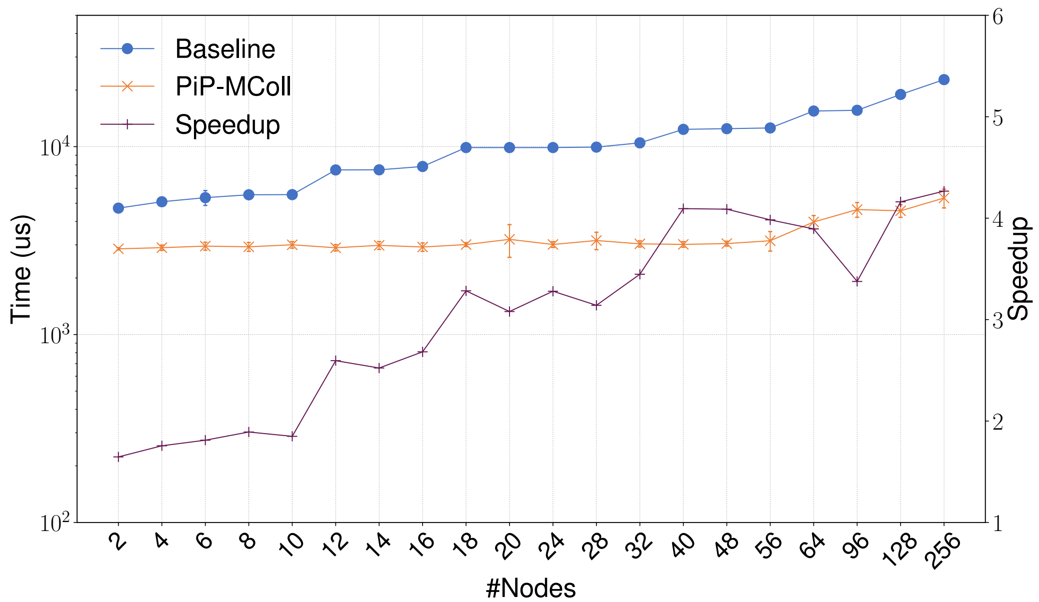
MPI_Scatter

We first show the performance of `MPI_Scatter` with various message sizes and numbers of nodes. Figure 4.6a shows the scatter performance with different message sizes (M_{size}) per process (i.e., overall data size on the root process is $M_{size} * \#process$) on 256 Broadwell nodes with 18 processes on each node.

In all cases PiP-MColl outperforms the baseline; however, PiP-MColl achieves the best speedup when the message size is small enough (i.e., around 2 bytes), and the speedup decreases constantly along with the increase of message size until 64 kB. The reason is that the network bandwidth is gradually saturated by the larger messages so that the benefits of multiobject scatter become less significant. For large messages, although PiP-MColl is not



(a) Message-based test on 256 nodes.



(b) Node-based test with 1-byte message size.

Figure 4.6: MPI_Scatter performance with different message size and numbers of nodes. PiP-MColl scatter uses the same algorithm throughout the tests.

able to bring about great speedup, it will not degrade the performance originally delivered by the baseline. We also notice the abnormal speedup increment when message sizes are 256 kB and 440 kB; this is due to the performance degradation of baseline instead of PiP-MColl. Theoretically, the baseline should be slightly slower than PiP-MColl in these cases.

On the other hand, in Figure 4.6b we present the performance of `MPI_Scatter` with fixed 1-byte message size and increasing numbers of nodes. We see that PiP-MColl beats the baseline at both small and large scale for small-message communication. We also notice when the number of nodes increases, the overall speedup gradually increases; this is because the multiobject design takes much fewer steps to complete in internode scatter (as shown in Equation 4.2 latency term), which provides the higher message rate at larger scale.

MPI_Allgather

We present the performance of `MPI_Allgather` with different message sizes and numbers of nodes. Since different algorithms are adopted for small and large messages in PiP-MColl, to better understand the performance, we mainly focus on the optimal allgather performance (*PiP-MColl Opt*); we show the PiP-MColl performance with only a small-message algorithm (*PiP-MColl Small*) as a reference.

Figure 4.7a shows the `MPI_Allgather` performance with different message sizes from 1 byte to 440 kB on 256 Xeon Broadwell nodes. *PiP-MColl Opt* performs better than the baseline in all cases, and it switches to the large-message algorithm after 32 kB message size. We note that for medium-size messages (e.g., 2 kB), the multiobject benefits will be negligible, and the major performance improvement is from intranode communication. In addition, *PiP-MColl Small* presents worse performance when the message size is larger

than 32 kB. This is because the overhead of the bandwidth term in Equation 4.4 becomes prominent. This issue is solved in *PiP-MColl Opt*, however.

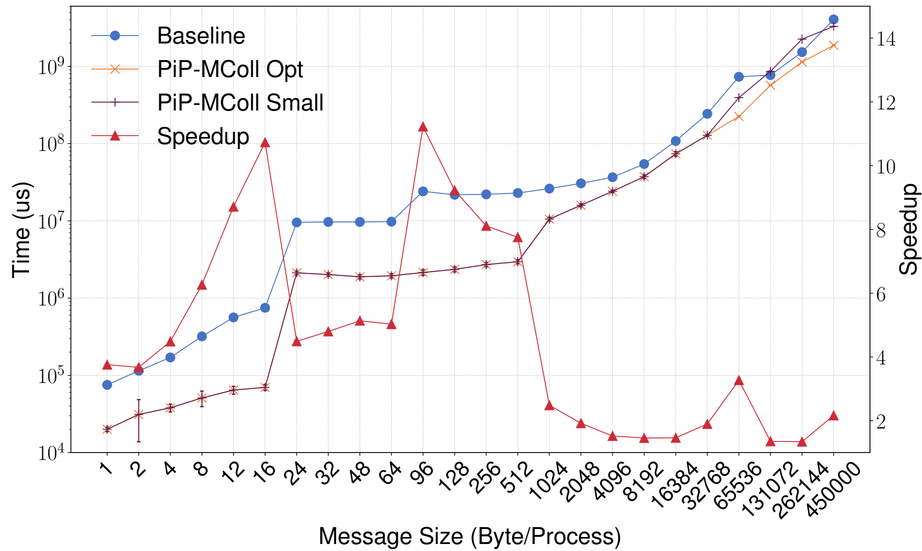
On the other hand, Figure 4.7b shows the `MPI_Allgather` performance with various numbers of nodes at fixed 1-byte message size per process. PiP-MColl outperforms the baseline in all cases; however, we notice that the speedup does not increase along with the increment in the number of nodes as it does with `MPI_Scatter`. This is because the multiobject design in `MPI_Allgather` requires synchronization among processes per node in the small-message algorithm Step 4, and the synchronization overhead is comparable to the communication runtime when the message size is not large enough. If the number of nodes is larger than $P+1$ (in our case, $P+1$ is equal to 19), this overhead persists in each repeated step, thus limiting overall speedup.

In summary, PiP-MColl `MPI_Allgather` is able to outperform the baseline with various numbers of nodes and message sizes using appropriate algorithm switch.

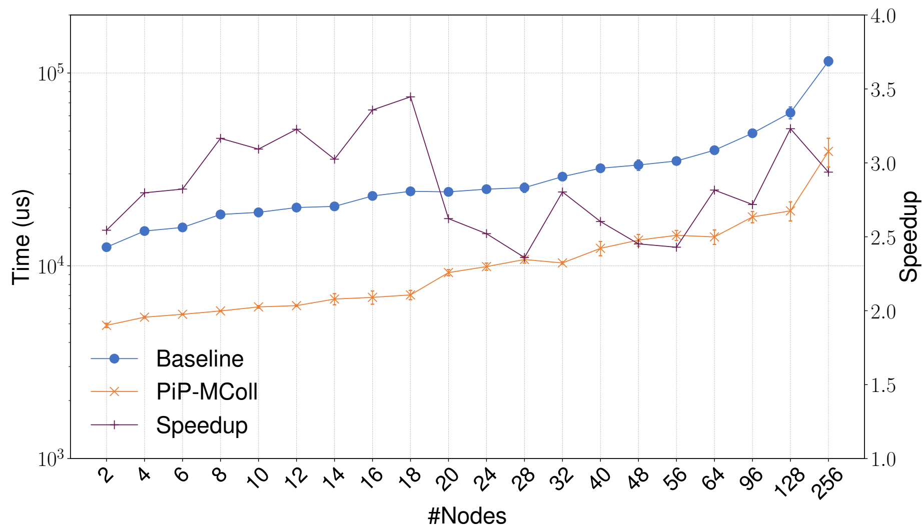
MPI_Allreduce

Similarly, we measure `MPI_Allreduce` performance with various message sizes and numbers of nodes. In this test we use double data type and the *MPI_SUM* operation as the input of `MPI_Allreduce`.

Figure 4.8a presents the performance with different message counts on 64 nodes. Similar to PiP-MColl `MPI_Allgather`, the large-message algorithm in PiP-MColl `MPI_Allreduce` is switched on at 8K message counts (i.e., 64 kB). However, we notice that the PiP-MColl performance is worse than the baseline when message counts are between 2K and 32K.



(a) Message-based test on 256 nodes.



(b) Node-based test with 1-byte message size.

Figure 4.7: MPI_Allgather performance with various message size and number of nodes. *PiP-MColl Opt* is the optimal case where the large-message algorithm is switched at 64 kB; *PiP-MColl Small* is the case where only the small-message algorithm is used throughout the test.

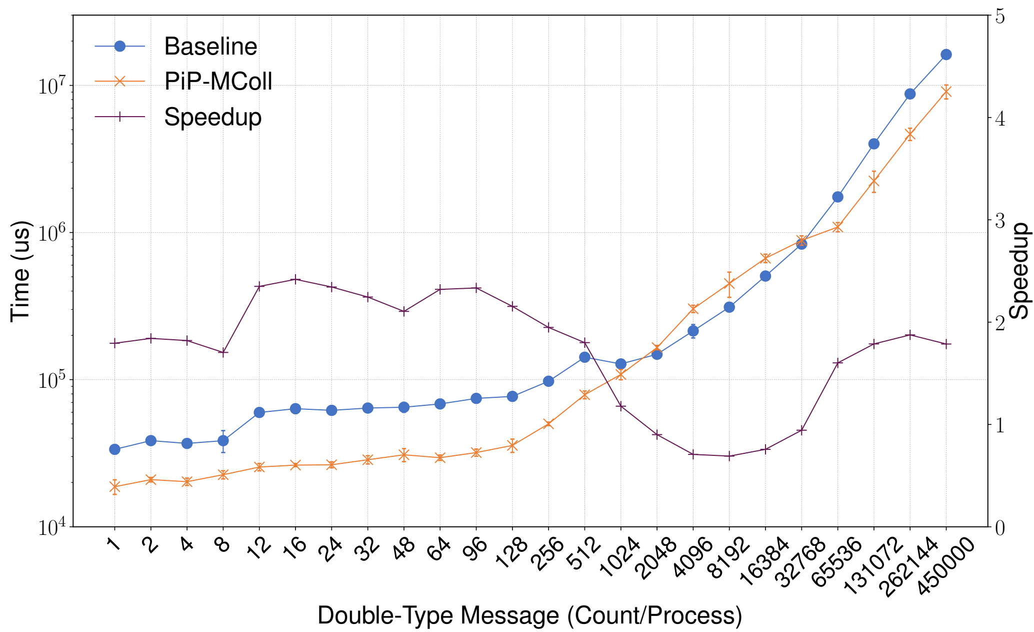
The reason is that in these cases we are still using the small-message algorithm. The increased message sizes result in the smaller multiobject benefits, which cannot offset the synchronization overhead. In addition, the results are different from MPI_Allgather because the message size will not increase along with the communication, which results in smaller ratio of communication to synchronization overhead; therefore, the PiP-MColl MPI_Allreduce not able to provide a performance boost similar to PiP-MColl MPI_Allgather in these cases.

On the other hand, when PiP-MColl switches to the large-message algorithm, it is not able to achieve better performance from 8K to 32K message counts. The reason is that the message sizes are still relatively small where the large-message algorithm benefits of PiP-MColl cannot offset the synchronization overhead. However, when the message count is larger than 32K, PiP-MColl performs better than the baseline because of more efficient design of large-message algorithm.

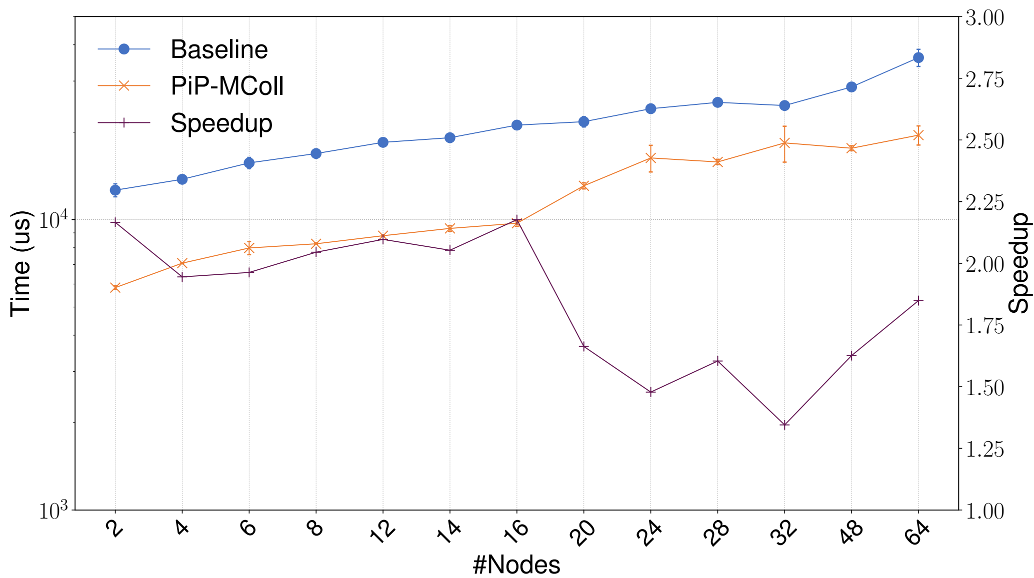
Considering scalability, Figure 4.8b shows the performance on different number of nodes with fixed 1 message count. In all cases, PiP-MColl outperforms the baseline, and this is because of the multiobject benefits which brings about higher message rate. Similar to `MPI_Allgather`, when the number of nodes is larger than $P+1$, the synchronization overhead persists in each repeated step of the algorithm which will limit overall speedup.

MPI Implementation Comparison

In this part we compare PiP-MColl with the widely used MPI libraries Intel-MPI, OpenMPI, and MVAPICH2 using `MPI_Allgather` communication. Figure 4.9 shows



(a) Message-based test on 64 nodes.



(b) Node-based test with fixed 1 message count.

Figure 4.8: MPI_Allreduce performance with various double-type message counts and nodes. For message-based execution, MPI_Allreduce switches to the large-message algorithm when message count is larger than or equal to 8 kB.

the corresponding performance. We notice for large messages (e.g., 440 kB), all of the MPI libraries provide excellent performance so that PiP-MColl does not provide too much performance benefit over them. For small and medium-message sizes (i.e., from 1 bytes to 8 kB), however, PiP-MColl presents the best performance, surpassing the other MPI libraries because of the multiobject benefits of PiP-MColl, which provides higher message rate and network bandwidth utilization.

In summary, although PiP-MColl is not able to deliver much better performance for large messages at large scale, it achieves the best performance for small and medium-size message communication.

4.4.3 Applications Evaluation

In this section we apply PiP-MColl to a real-world application, an N-body simulation, which is widely used in astronomy, and show the strong-scaling performance.

N-body simulation [119] is the foundation of numerous scientific applications where the movement of particles (e.g., stars) in the dynamic systems is simulated over time under certain forces (e.g., gravity) [117]. Current N-body simulation is usually implemented in the MPI environment because of the huge amount of data computation and communication requirements. On the other hand, the parallel algorithms that can be applied by N-body simulation are massive; we choose the Barnes–Hut approximation algorithm [10], which is a widely used hierarchical algorithm in the real world.

The key input of N-body simulation is the number of particles N to be calculated and their initial status including position, velocity, mass, and radius. For simplicity, these

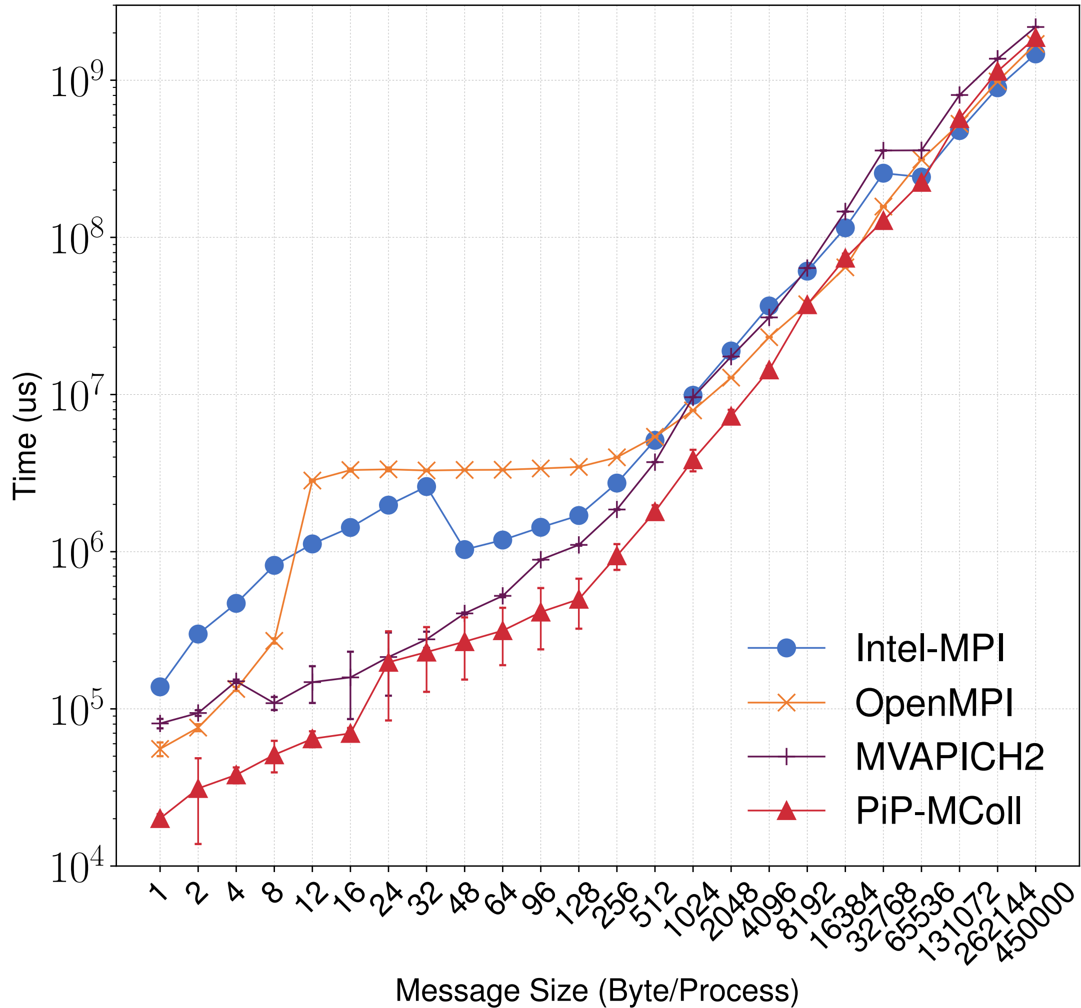


Figure 4.9: MPI_Allgather performance comparison among Intel-MPI, OpenMPI, MVAPICH2, and PiP-MColl with different message sizes on 256 Xeon Broadwell nodes.

are randomly initialized in our test in the beginning, and we only set the input parameter N to define the problem size. The simulation iterates force and position computation many times. The major MPI routine adopted by the simulation is `MPI_Allgather`, and multiple iterations are executed.

We perform the strong-scaling experiment with nodes ranging from 4 to 256, and we set $N = 10^5$. Figure 4.10 shows the average runtime of one simulation iteration of

baseline and PiP-MColl with runtime breakdown. We notice that when the number of nodes is 4, the speedup is the smallest; this is because the ratio of communication to computation is small, which results in negligible runtime improvement from communication for the whole application. When we increase the execution scale, however, the overall speedup increases rapidly. This is because PiP-MColl benefits predominate and bring about higher message rate and network throughput.

The runtime and speedup trend comply with our microbenchmark results in Figure 4.7. In N-body simulation, when the execution scale is increased, the workload per process of MPI_Allgather decreases, and PiP-MColl switches to the small-message algorithm, which provides maximal 2.5 speedup at 256 nodes.

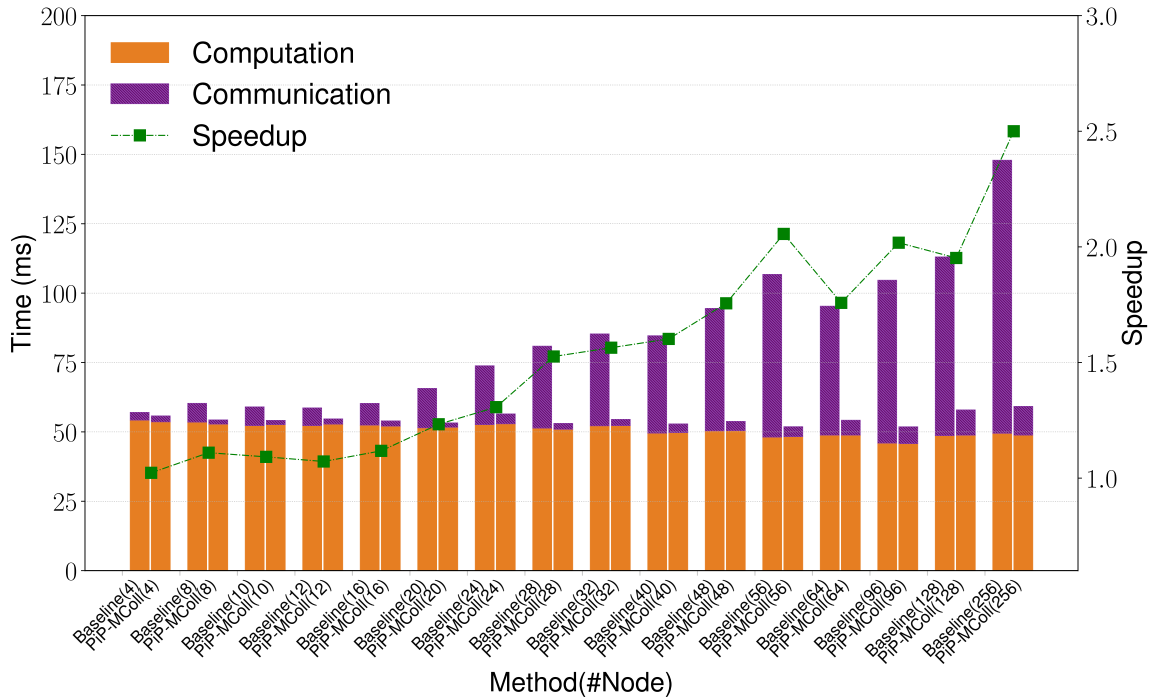


Figure 4.10: Strong scaling N-body simulation performance from 4 to 256 nodes. The particle is set as 10^5 and average runtime of multiple iterations is presented.

4.5 Summary

MPI collective performance is a popular research topic that has been studied for years. The state-of-the-art works adopt various methods to improve intra- and internode collective communication performance; among them, XPMEM, CMA, KNEM, and POSIX shared-memory techniques are widely used for efficient MPI collective design. However, these designs involve heavy system overhead or double copy overhead, which results in suboptimal performance of collective algorithms.

In this thesis we propose PiP-MColl, a PiP-based multiobject interprocess MPI collective design, to improve performance for small- and medium-message collectives without causing extra overhead. PiP-MColl utilizes a PiP shared-memory technique to load MPI processes into the same virtual memory space and allows us to perform data copy at the userspace and avoid system and double copy overhead. Multiobject design at large scale for small- and medium-message communication in the PiP environment enables us to obtain maximal message rate and network throughput. We apply PiP-MColl to three widely used MPI collectives: `MPI_Scatter`, `MPI_Allgather`, and `MPI_Allreduce`. The experimental results show that PiP-MColl performs much better than the baseline PiP-MPICH in all cases and also beats the widely used MPI libraries Intel MPI, OpenMPI, and MVAPICH2. In addition, the real-world N-body application obtains better performance than the baseline, providing further proof of the effectiveness of PiP-MColl.

In summary, PiP-MColl is an efficient method that utilizes multiobject and the PiP shared-memory technique and is able to maximize performance at a large scale for small- and medium-message MPI collectives.

Chapter 5

Conclusions

In this thesis, we focus on three problems in MPI communication and propose the corresponding solutions.

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. In addition, traditional software-level asynchronous progress mechanisms have to statically configure progress resources (i.e., CPU cores). Such a method may even perform poorly for multiple-stage applications where each stage often forms a different communication and computation pattern.

To solve these issues, 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.

At the same time, 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.

To further improve MPI collectives performance, we propose PiP-MColl, a PiP-based multiobject interprocess MPI collective design, to improve performance for small- and medium-message collectives without causing extra overhead. PiP-MColl utilizes a PiP shared-memory technique to load MPI processes into the same virtual memory space and allows us to perform data copy at the userspace and avoid system and double copy overhead. Multiobject design at large scale for small- and medium-message communication in the PiP

environment enables us to obtain maximal message rate and network throughput. The experimental results show that PiP-MColl performs much better than the baseline PiP-MPICH in all cases and also beats the widely used MPI libraries Intel MPI, OpenMPI, and MVAPICH2. In addition, the real-world N-body application obtains better performance than the baseline, providing further proof of the effectiveness of PiP-MColl.

Bibliography

[1] OpenFabrics Interfaces (OFI).

[2] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, pages 265–283, 2016.

[3] Umut A Acar, Guy E Blelloch, and Robert D Blumofe. The Data Locality of Work Stealing. In *Proceedings of the twelfth annual ACM symposium on Parallel algorithms and architectures*, pages 1–12. ACM, 2000.

[4] Umut A Acar, Arthur Chargu'eraud, and Mike 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*, pages 219–228, 2013.

[5] S. Arnautov, P. Felber, C. Fetzer, and B. Trach. FFQ: A Fast Single-Producer/Multiple-Consumer Concurrent FIFO Queue. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 907–916, Los Alamitos, CA, USA, June 2017. IEEE Computer Society.

[6] Nimar S Arora, Robert D Blumofe, and C Greg Plaxton. Thread Scheduling for Multiprogrammed multiprocessors. *Theory of computing systems*, 34(2):115–144, 2001.

[7] Ammar Ahmad Awan, Khaled Hamidouche, Jahanzeb Maqbool Hashmi, and Dhabaleswar K Panda. S-caffe: Co-designing mpi runtimes and caffe for scalable deep learning on modern gpu clusters. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 193–205, 2017.

[8] Alan Ayala, Stanimire Tomov, Xi Luo, Hejer Shaeik, Azzam Haidar, George Bosilca, and Jack Dongarra. Impacts of multi-gpu mpi collective communications on large fft computation. In *2019 IEEE/ACM Workshop on Exascale MPI (ExaMPI)*, pages 12–18. IEEE, 2019.

[9] Saman Barghi and Martin Karsten. Work-Stealing, Locality-Aware Actor Scheduling. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 484–494. IEEE, 2018.

[10] Josh Barnes and Piet Hut. A hierarchical o (n log n) force-calculation algorithm. *nature*, 324(6096):446–449, 1986.

[11] Richard F Barrett, Courtenay T Vaughan, and Michael A Heroux. MiniGhost: A Miniapp for Exploring Boundary Exchange Strategies using Stencil Computations in Scientific parallel computing. *Sandia National Laboratories, Tech. Rep. SAND*, 5294832:2011, 2011.

[12] Milind Bhandarkar, Laxmikant V Kalé, Eric de Sturler, and Jay Hoeflinger. Adaptive Load Balancing for MPI Programs. In *International Conference on Computational Science*, pages 108–117. Springer, 2001.

[13] Mark S. Birrittella, Mark Debbage, Ram Huggahalli, James Kunz, Tom Lovett, Todd Rimmer, Keith D. Underwood, and Robert C. Zak. Intel Omni-path Architecture: Enabling Scalable, High Performance Fabrics. In *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*, pages 1–9, 2015.

[14] Mark S Birrittella, Mark Debbage, Ram Huggahalli, James Kunz, Tom Lovett, Todd Rimmer, Keith D Underwood, and Robert C Zak. Intel® omni-path architecture: Enabling scalable, high performance fabrics. In *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*, pages 1–9. IEEE, 2015.

[15] Rupak Biswas, Sajal K Das, Daniel J Harvey, and Leonid Oliker. Parallel Dynamic Load Balancing Strategies for Adaptive Irregular Applications. *Applied Mathematical Modelling*, 25(2):109–122, 2000.

[16] Robert D Blumofe and Charles E Leiserson. Scheduling Multithreaded Computations by Work Stealing. *Journal of the ACM (JACM)*, 46(5):720–748, 1999.

[17] Ron Brightwell, Rolf Riesen, and Keith D Underwood. Analyzing the impact of overlap, offload, and independent progress for message passing interface applications. *The International Journal of High Performance Computing Applications*, 19(2):103–117, 2005.

[18] Jehoshua Bruck, Ching-Tien Ho, Shlomo Kipnis, Eli Upfal, and Derrick Weathersby. Efficient algorithms for all-to-all communications in multiport message-passing systems. *IEEE Transactions on parallel and distributed systems*, 8(11):1143–1156, 1997.

[19] Darius Buntinas, Guillaume Mercier, and William Gropp. Implementation and Evaluation of Shared-Memory Communication and Synchronization Operations in MPICH2 using the Nemesis Communication Subsystem. *Parallel Computing*, 33(9):634–644, 2007.

[20] José María Cecilia, José Manuel García, and Manuel Ujaldón. CUDA 2D stencil computations for the jacobi method. In *International Workshop on Applied Parallel Computing*, pages 173–183. Springer, 2010.

[21] Sourav Chakraborty, Mohammadreza Bayatpour, J Hashmi, Hari Subramoni, and Dhabaleswar K Panda. Cooperative Rendezvous Protocols for Improved Performance and Overlap. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 361–373. IEEE, 2018.

[22] Sourav Chakraborty, Hari Subramoni, and Dhabaleswar K Panda. Contention-aware kernel-assisted mpi collectives for multi/many-core systems. In *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 13–24. IEEE, 2017.

[23] David Chase and Yossi Lev. Dynamic Circular Work-Stealing Deque. In *Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures*, pages 21–28. ACM, 2005.

[24] Quan Chen and Minyi Guo. Contention and locality-aware work-stealing for iterative applications in multi-socket computers. *IEEE Transactions on Computers*, 67(6):784–798, 2017.

[25] Quan Chen, Minyi Guo, and Haibing Guan. LAWS: Locality-Aware Work-Stealing for Multi-Socket Multi-Core Architectures. In *Proceedings of the 28th ACM international conference on Supercomputing*, pages 3–12. ACM, 2014.

[26] Quan Chen, Minyi Guo, and Zhiyi Huang. Adaptive Cache Aware Bitier Work-Stealing in Multisocket Multicore Architectures. *IEEE Transactions on Parallel and Distributed Systems*, 24(12):2334–2343, 2012.

[27] Quan Chen, Minyi Guo, and Zhiyi 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*, pages 163–172. ACM, 2012.

[28] Quan Chen, Zhiyi Huang, Minyi Guo, and Jingyu Zhou. Cab: Cache Aware Bitier Task-Stealing in Multi-Socket Multi-Core Architecture. In *2011 International Conference on Parallel Processing*, pages 722–732. IEEE, 2011.

[29] Sudheer Chunduri, Scott Parker, Pavan Balaji, Kevin Harms, and Kalyan Kumaran. Characterization of mpi usage on a production supercomputer. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 386–400. IEEE, 2018.

[30] Intel Corporation. Intel MPI Library, 2021.

[31] Alexandre Denis. pioman: a Pthread-Based Multithreaded Communication Engine. In *2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 155–162. IEEE, 2015.

[32] Justin Deters, Jiaye Wu, Yifan Xu, and I-Ting Angelina Lee. A NUMA-aware provably-efficient task-parallel platform based on the work-first principle. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*, pages 59–70. IEEE, 2018.

[33] Karen D. Devine, Erik G. Boman, and George Karypis. *Partitioning and Load Balancing for Emerging Parallel Applications and Architectures*, pages 99–126. 2006.

[34] Vijay Dhanraj. *Enhancement of LiMIC-Based Collectives for Multi-core Clusters.* PhD thesis, The Ohio State University, 2012.

[35] James S. Dinan, Pavan Balaji, Jeffrey R. Hammond, Sriram Krishnamoorthy, and Vinod Tipparaju. Supporting the Global Arrays PGAS Model using MPI One-Sided Communication. In *2012 IEEE International Parallel and Distributed Processing Symposium*, May 2012.

[36] Andi Drebes, Antoniu Pop, Karine Heydemann, Nathalie Drach, and Albert 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*, pages 1–2, 2016.

[37] Ulrich Drepper. ELF handling for thread-local storage, version 0.20. *Red Hat*, 2005.

[38] James W. Lottes Paul F. Fischer and Stefan G. Kerkemeier. Nek5000 Web Page, 2008.

[39] P Fischer and K Heisey. NEKBONE: Thermal Hydraulics Mini-Application, 2013.

[40] P Fischer, J Kruse, J Mullen, H Tufo, J Lottes, and S Kerkemeier. Nek5000: Open Source Spectral Element CFD Solver. *Argonne National Laboratory, Mathematics and Computer Science Division, Argonne, IL, see https://nek5000. mcs. anl. gov/index. php/MainPage*, 2008.

[41] 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*, 26(1–2):241–263, 1998.

[42] Mark Gates, Jakub Kurzak, Ali Charara, Asim YarKhan, and Jack Dongarra. SLATE: Design of a Modern Distributed and Accelerated Linear Algebra Library. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 26. ACM, 2019.

[43] Allen Gersho and Robert M Gray. *Vector quantization and signal compression*, volume 159. Springer Science & Business Media, 2012.

[44] Robert Gerstenberger, Maciej Besta, and Torsten Hoefler. Enabling Highly-Scalable Remote Memory Access Programming with MPI-3 One Sided. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 53:1–53:12. ACM, November 2013.

[45] Brice Goglin and Stephanie Moreaud. KNEM: A generic and scalable kernel-assisted intra-node MPI communication framework. *Journal of Parallel and Distributed Computing*, 73(2):176–188, 2013.

[46] Brice Goglin and Stephanie Moreaud. KNEM: A generic and scalable kernel-assisted intra-node MPI communication framework. *Journal of Parallel and Distributed Computing*, 73(2):176–188, 2013.

[47] William D Gropp. Learning from the success of MPI. In *International Conference on High-Performance Computing*, pages 81–92. Springer, 2001.

[48] Paul Grun, Sean Hefty, Sayantan Sur, David Goodell, Robert D Russell, Howard Pritchard, and Jeffrey M Squyres. A brief introduction to the openfabrics interfaces-a new network api for maximizing high performance application efficiency. In *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*, pages 34–39. IEEE, 2015.

[49] Yi Guo, Jisheng Zhao, Vincent Cave, and Vivek Sarkar. SLAW: A Scalable Locality-Aware Adaptive Work-Stealing Scheduler. In *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, pages 1–12. IEEE, 2010.

[50] S. K. Gutierrez, N. T. Hjelm, M. G. Venkata, and R. L. Graham. Performance Evaluation of Open MPI on Cray XE/XK Systems. In *2012 IEEE 20th Annual Symposium on High-Performance Interconnects*, pages 40–47, 2012.

[51] Per Hammarlund, Alberto J Martinez, Atiq A Bajwa, David L Hill, Erik Hallnor, Hong Jiang, Martin Dixon, Michael Derr, Mikal Hunsaker, Rajesh Kumar, et al. Haswell: The Fourth-Generation Intel Core Processor. *IEEE Micro*, 34(2):6–20, 2014.

[52] Jeff R Hammond, Sriram Krishnamoorthy, Sameer Shende, Nichols A Romero, and Allen D Malony. Performance characterization of global address space applications: a case study with NWChem. *Concurrency and Computation: Practice and Experience*, 24(2):135–154, 2012.

[53] Jahanzeb Maqbool Hashmi, Sourav Chakraborty, Mohammadreza Bayatpour, Hari Subramoni, and Dhabaleswar K Panda. Designing efficient shared address space reduction collectives for multi-/many-cores. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1020–1029. IEEE, 2018.

[54] Danny Hendler and Nir Shavit. Non-Blocking Steal-Half Work Queues. In *Proceedings of the twenty-first annual symposium on Principles of distributed computing*, pages 280–289. ACM, 2002.

[55] Bruce Hendrickson and Karen Devine. Dynamic Load Balancing in Computational Mechanics. *Computer Methods in Applied Mechanics and Engineering*, 184(2):485–500, 2000.

[56] Tasuku Hiraishi, Masahiro Yasugi, Seiji Umatani, and Taiichi Yuasa. Backtracking-Based Load Balancing. *ACM Sigplan Notices*, 44(4):55–64, 2009.

[57] Nathan Hjelm, Pavel Shamis, and Jeff Squyres. XPMEM Linux Kernel Module, 2018.

[58] Nathan Hjelm, Pavel Shamis, and Jeff Squyres. XPMEM Linux Kernel Module, 2018.

[59] Roger W Hockney. The communication challenge for mpp: Intel paragon and meiko cs-2. *Parallel computing*, 20(3):389–398, 1994.

[60] Torsten Hoefler and Andrew Lumsdaine. Message progression in parallel computing-to thread or not to thread? In *2008 IEEE International Conference on Cluster Computing*, pages 213–222. IEEE, 2008.

[61] Torsten Hoefler, Andrew Lumsdaine, and Wolfgang Rehm. Implementation and performance analysis of non-blocking collective operations for MPI. In *SC'07: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, pages 1–10. IEEE, 2007.

[62] Atsushi Hori, Balazs Gerofi, and Yutaka Ishikawa. An implementation of user-level processes using address space sharing. In *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 976–984, 2020.

[63] Atsushi Hori, Min Si, Balazs Gerofi, Masamichi Takagi, Jai Dayal, Pavan Balaji, and Yutaka Ishikawa. Process-in-Process: Techniques for Practical Address-Space Sharing. In *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*, pages 131–143. ACM, 2018.

[64] Chao Huang, Orion Lawlor, and Laxmikant V Kale. Adaptive mpi. In *International workshop on languages and compilers for parallel computing*, pages 306–322. Springer, 2003.

[65] Intel Corporation. OPA-PSM2 Github repository.

[66] Arpan Jain, Ammar Ahmad Awan, Hari Subramoni, and Dhabaleswar K Panda. Scaling tensorflow, pytorch, and mxnet using mvapich2 for high-performance deep learning on frontera. In *2019 IEEE/ACM Third Workshop on Deep Learning on Supercomputers (DLS)*, pages 76–83. IEEE, 2019.

[67] Surabhi Jain, Rashid Kaleem, Marc Gamell Balmana, Akhil Langer, Dmitry Durnov, Alexander Sannikov, and Maria Garzaran. Framework for scalable intra-node collective operations using shared memory. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 374–385. IEEE, 2018.

[68] Surabhi Jain, Rashid Kaleem, Marc Gamell Balmana, Akhil Langer, Dmitry Durnov, Alexander Sannikov, and Maria 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*, SC '18. IEEE Press, 2018.

[69] Weihang Jiang, Jiuxing Liu, Hyun-Wook Jin, Dhabaleswar K Panda, Darius Buntinas, Rajeev Thakur, and William 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*, pages 68–76. Springer, 2004.

[70] H-W Jin, Sayantan Sur, Lei Chai, and Dhabaleswar K Panda. Limic: Support for high-performance mpi intra-node communication on linux cluster. In *2005 International Conference on Parallel Processing (ICPP'05)*, pages 184–191. IEEE, 2005.

[71] Laxmikant V Kale and Sanjeev Krishnan. CHARM++: A Portable Concurrent Object Oriented System Based on C++. In *OOPSLA*, volume 93, pages 91–108. Citeseer, 1993.

[72] Krishna Kandalla, Hari Subramoni, Gopal Santhanaraman, Matthew Koop, and Dhabaleswar K Panda. Designing multi-leader-based allgather algorithms for multi-core clusters. In *2009 IEEE International Symposium on Parallel & Distributed Processing*, pages 1–8. IEEE, 2009.

[73] Michael Kerrisk. Overview of POSIX Shared Memory, 2008.

[74] Michael Kerrisk. Linux Programmer's Manual, 2020.

[75] Michael Kerrisk. Overview of POSIX Shared Memory, 2021.

[76] Manojkumar Krishnan, Jarek Nieplocha, Michael Blocksome, and Brian Smith. Evaluation of remote memory access communication on the IBM blue gene/P supercomputer. In *2008 International Conference on Parallel Processing-Workshops*, pages 109–115. IEEE, 2008.

[77] Sameer Kumar, Gabor Dozsa, Gheorghe Almasi, Philip Heidelberger, Dong Chen, Mark E Giampapa, Michael Blocksome, Ahmad Faraj, Jeff Parker, Joseph 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*, pages 94–103, 2008.

[78] Sameer Kumar, Amith R Mamidala, Daniel A Faraj, Brian Smith, Michael Blocksome, Bob Cernohous, Douglas Miller, Jeff Parker, Joseph Ratterman, Philip 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*, pages 763–773. IEEE, 2012.

[79] Sameer Kumar, Yanhua Sun, and Laximant 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*, pages 689–699. IEEE, 2013.

[80] Argonne National Laboratory. MPICH — High-Performance Portable MPI, 2021.

[81] Ping Lai, Pavan Balaji, Rajeev Thakur, and Dhabaleswar K Panda. ProOnE: a General-Purpose Protocol Onload Engine for Multi- and Many-Core Architectures. *Computer Science-Research and Development*, 23(3-4):133–142, 2009.

[82] Shigang Li, Torsten Hoefler, and Marc Snir. Numa-aware shared-memory collective communication for mpi. In *Proceedings of the 22nd international symposium on High-performance parallel and distributed computing*, pages 85–96, 2013.

[83] Jonathan Lifflander, Sriram Krishnamoorthy, and Laxmikant 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*, pages 857–868. IEEE, 2014.

[84] Jiuxing Liu, Jiesheng Wu, and Dhabaleswar K Panda. High performance RDMA-based MPI implementation over infiniband. *International Journal of Parallel Programming*, 32(3):167–198, 2004.

[85] Stuart Lloyd. Least squares quantization in pcm. *IEEE transactions on information theory*, 28(2):129–137, 1982.

[86] Teng Ma, George Bosilca, Aurelien Bouteiller, Brice Goglin, Jeffrey M Squyres, and Jack J Dongarra. Kernel assisted collective intra-node mpi communication among multi-core and many-core cpus. In *2011 International Conference on Parallel Processing*, pages 532–541. IEEE, 2011.

[87] Seung-Jai Min, Costin Iancu, and Katherine Yelick. Hierarchical Work Stealing on Manycore Clusters. In *Fifth Conference on Partitioned Global Address Space Programming Models (PGAS11)*, volume 625, 2011.

[88] Jarek Nieplocha, Bruce Palmer, Vinod Tipparaju, Manojkumar Krishnan, Harold Trease, and Edoardo Apra. Advances, Applications and Performance of the Global Arrays Shared Memory Programming Toolkit. *The International Journal of High Performance Computing Applications*, 20(2):203–231, 2006.

[89] Kaiming Ouyang, Yanfei Guo, Min Si, Astushi Hori, Hui Zhou, Ken Raffenetti, Zizhong Chen, and Rajeev Thakur. Efficient process-in-process based multiobject interprocess mpi collectives for large-scale applications. In *36th ACM International Conference on Supercomputing (ICS'22)*, pages 516–527. IEEE, 2021.

[90] Kaiming Ouyang, Min Si, Astushi Hori, Zizhong Chen, and Pavan Balaji. Daps: A dynamic asynchronous progress stealing model for mpi communication. In *2021 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 516–527. IEEE, 2021.

[91] Kaiming Ouyang, Min Si, Atsushi Hori, Zizhong Chen, and Pavan Balaji. CAB-MPI: Exploring interprocess work-stealing towards balanced MPI communication. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15. IEEE, 2020.

[92] Benjamin S Parsons and Vijay S Pai. Accelerating mpi collective communications through hierarchical algorithms without sacrificing inter-node communication flexibility. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pages 208–218. IEEE, 2014.

[93] Jeeva Paudel, Olivier Tardieu, and José Nelson Amaral. On the Merits of Distributed Work-Stealing on Selective Locality-Aware Tasks. In *2013 42nd International Conference on Parallel Processing*, pages 100–109. IEEE, 2013.

[94] Marc Pérache, Patrick Carribault, and Hervé Jourdren. Mpc-mpi: An mpi implementation reducing the overall memory consumption. In *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*, pages 94–103. Springer, 2009.

[95] Marc Pérache, Hervé Jourdren, and Raymond Namyst. MPC: A Unified Parallel Runtime for Clusters of NUMA Machines. In *Proceedings of the 14th International Euro-Par Conference on Parallel Processing*, Euro-Par'08, pages 78–88, Berlin, Heidelberg, 2008. Springer-Verlag.

[96] Antoine Petitet, RC Whaley, J Dongarra, and A Cleary. HPL–A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers (2004). *Available from Internet:¡ http://www. netlib. org/benchmark/hpl*, 2016.

[97] Howard Pritchard, Duncan Roweth, David Henseler, and Paul 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*, volume 79, page 130, 2012.

[98] Rolf Rabenseifner. Optimization of collective reduction operations. In *International Conference on Computational Science*, pages 1–9. Springer, 2004.

[99] Amit Ruhela, Hari Subramoni, Sourav Chakraborty, Mohammadreza Bayatpour, Pouya Kousha, and Dhabaleswar K Panda. Efficient asynchronous communication progress for MPI without dedicated resources. In *Proceedings of the 25th European MPI Users' Group Meeting*, pages 1–11, 2018.

[100] Marc Sergent, Mario Dagrada, Patrick Carribault, Julien Jaeger, Marc Pérache, and Guillaume Papauré. Efficient Communication/Computation Overlap with MPI+OpenMP Runtimes Collaboration. In *European Conference on Parallel Processing*, pages 560–572. Springer, 2018.

[101] Mohammed Shaheen and Robert Strzodka. NUMA Aware Iterative Stencil Computations on Many-Core Systems. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium*, pages 461–473. IEEE, 2012.

[102] Pavel Shamis, Manjunath Gorentla Venkata, M Graham Lopez, Matthew B Baker, Oscar Hernandez, Yossi Itigin, Mike Dubman, Gilad Shainer, Richard L Graham, Liran Liss, et al. UCX: An Open Source Framework for HPC Network APIs and Beyond. In *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*, pages 40–43. IEEE, 2015.

[103] Brendan Sheridan and Jeremy T Fineman. A Case for Distributed Work-Stealing in Regular Applications. In *Proceedings of the 6th ACM SIGPLAN Workshop on X10*, pages 32–33, 2016.

[104] Shumpei Shiina and Kenjiro Taura. Almost deterministic work stealing. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–16, 2019.

[105] Mohak Shroff and Robert A Van De Geijn. Collmark: Mpi collective communication benchmark. In *International Conference on Supercomputing*, page 10. Citeseer, 2000.

[106] Min Si and Pavan 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)*, pages 206–214. IEEE, 2017.

[107] Min Si, Antonio J Peña, Pavan Balaji, Masamichi Takagi, and Yutaka Ishikawa. MT-MPI: Multithreaded MPI for Many-Core Environments. In *Proceedings of the 28th ACM international conference on Supercomputing*, pages 125–134, 2014.

[108] Min Si, Antonio J Pena, Jeff Hammond, Pavan Balaji, and Yutaka Ishikawa. Scaling NWChem with Efficient and Portable Asynchronous Communication in MPI RMA. In *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 811–816. IEEE, 2015.

[109] Min Si, Antonio J Pena, Jeff Hammond, Pavan Balaji, Masamichi Takagi, and Yutaka Ishikawa. Casper: An asynchronous progress model for MPI RMA on many-core architectures. In *2015 IEEE International Parallel and Distributed Processing Symposium*, pages 665–676. IEEE, 2015.

[110] Min Si, Antonio J Pena, Jeff Hammond, Pavan Balaji, Masamichi Takagi, and Yutaka Ishikawa. Dynamic adaptable asynchronous progress model for MPI RMA multiphase applications. volume 29, pages 1975–1989. IEEE, 2018.

[111] Warut Suksompong, Charles E Leiserson, and Tao B Schardl. On the Efficiency of Localized Work Stealing. *Information Processing Letters*, 116(2):100–106, 2016.

[112] Sayantan Sur, Hyun-Wook Jin, Lei Chai, and Dhabaleswar 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*, pages 32–39, 2006.

[113] Hong Tang and Tao Yang. Optimizing threaded mpi execution on smp clusters. In *Proceedings of the 15th international conference on Supercomputing*, pages 381–392, 2001.

[114] Olivier Tardieu, Haichuan Wang, and Haibo Lin. A Work-Stealing Scheduler for X10's Task Parallelism with Suspension. *ACM SIGPLAN Notices*, 47(8):267–276, 2012.

[115] Rajeev Thakur and William D Gropp. Improving the Performance of Collective Operations in MPICH. In *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*, pages 257–267. Springer, 2003.

[116] Rajeev Thakur, Rolf Rabenseifner, and William Gropp. Optimization of collective communication operations in mpich. *The International Journal of High Performance Computing Applications*, 19(1):49–66, 2005.

[117] E Thommes, M Nagasawa, and DNC Lin. Dynamical shake-up of planetary systems. ii. n-body simulations of solar system terrestrial planet formation induced by secular resonance sweeping. *The Astrophysical Journal*, 676(1):728, 2008.

[118] François Trahay and Alexandre Denis. A scalable and generic task scheduling system for communication libraries. In *2009 IEEE International Conference on Cluster Computing and Workshops*, pages 1–8. IEEE, 2009.

[119] Michele Trenti and Piet Hut. N-body simulations (gravitational). *Scholarpedia*, 3(5):3930, 2008.

[120] Alexandros Tzannes, George C Caragea, Rajeev Barua, and Uzi Vishkin. Lazy Binary-Splitting: A Run-Time Adaptive Work-Stealing Scheduler. In *ACM Sigplan Notices*, volume 45, pages 179–190. ACM, 2010.

[121] The Ohio State University. MVAPICH: MPI over InfiniBand, 10GigE/iWARP and RoCE, 2021.

[122] M Usama. Advances in knowledge discovery and data mining. 1996.

[123] Karthikeyan Vaidyanathan, Dhiraj D Kalamkar, Kiran Pamnany, Jeff R Hammond, Pavan Balaji, Dipankar Das, Jongsoo Park, and Bálint 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*, pages 1–12. IEEE, 2015.

[124] Marat Valiev, Eric J Bylaska, Niranjan Govind, Karol Kowalski, Tjerk P Straatsma, Hubertus JJ Van Dam, Dunyou Wang, Jarek Nieplocha, Edoardo Apra, Theresa L Windus, et al. NWChem: A comprehensive and scalable open-source solution for large scale molecular simulations. *Computer Physics Communications*, 181(9):1477–1489, 2010.

[125] Jerome 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*, page 33. ACM, 2014.

[126] David W Walker and Jack J Dongarra. Mpi: a standard message passing interface. *Supercomputer*, 12:56–68, 1996.

[127] Yusong Wang and Michael Borland. Pelegant: A parallel accelerator simulation code for electron generation and tracking. In *AIP Conference Proceedings*, volume 877, pages 241–247. American Institute of Physics, 2006.

[128] Richard M Yoo, Christopher J Hughes, Changkyu Kim, Yen-Kuang Chen, and Christos 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*, pages 315–325. ACM, 2013.

[129] Chenhan D Yu, Jianyu Huang, Woody Austin, Bo Xiao, and George Biros. Performance optimization for the k-nearest neighbors kernel on x86 architectures. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12, 2015.

[130] Han Zhao, Quan Chen, Yuxian Qiu, Ming Wu, Yao Shen, Jingwen Leng, Chao Li, and Minyi Guo. Bandwidth and Locality Aware Task-stealing for Manycore Architectures with Bandwidth-Asymmetric Memory. *ACM Transactions on Architecture and Code Optimization (TACO)*, 15(4):55, 2018.