mathematics

MDPI

Review

Towards Efficient HPC: Exploring Overlap Strategies Using MPI Non-Blocking Communication

Yuntian Zheng and Jianping Wu *

College of Meteorology and Oceanography, National University of Defense Technology, Changsha 410073, China; zhengyuntian18@nudt.edu.cn
* Correspondence: wjp@nudt.edu.cn

Abstract: As high-performance computing (HPC) platforms continue to scale up, communication costs have become a critical bottleneck affecting overall application performance. An effective strategy to overcome this limitation is to overlap communication with computation. The Message Passing Interface (MPI), as the de facto standard for communication in HPC, provides non-blocking communication primitives that make such overlapping feasible. By enabling asynchronous communication, non-blocking operations reduce idle time of cores caused by data transfer delays, thereby improving resource utilization. Overlapping communication with computation is particularly important for enhancing the performance of large-scale scientific applications, such as numerical simulations, climate modeling, and other data-intensive tasks. However, achieving efficient overlapping is non-trivial and depends not only on advances in hardware technologies such as Remote Direct Memory Access (RDMA), but also on well-designed and optimized MPI implementations. This paper presents a comprehensive survey on the principles of MPI non-blocking communication, the core techniques for achieving computation–communication overlap, and some representative applications in scientific computing. Alongside the survey, we include a preliminary experimental study evaluating the effectiveness of asynchronous progress mechanism on modern HPC platforms to support the development of parallel programs for HPC researchers and practitioners.

Keywords: HPC; MPI; non-blocking communication; computation–communication overlap; parallel computing; asynchronous communication

MSC: 65Y05; 68W10

check for updates

Academic Editor: Jonathan Blackledge

Received: 29 April 2025
Revised: 26 May 2025
Accepted: 31 May 2025
Published: 2 June 2025

Citation: Zheng, Y.; Wu, J. Towards Efficient HPC: Exploring Overlap Strategies Using MPI Non-Blocking Communication. *Mathematics* **2025**, *13*, 1848. https://doi.org/10.3390/math13111848

Copyright: © 2025 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

1. Introduction

With the growing demands of scientific computing, engineering simulation, and big data analytics, HPC has become a cornerstone of modern scientific research and industrial advancement. However, in large-scale applications, due to the overhead incurred by data transfer, communication often becomes a bottleneck to achieve high performance in massively parallel environments [1,2]. As a result, many researchers have focused on how to address this issue [3,4].

In parallel computing, MPI provides a crucial mechanism for addressing this challenge through non-blocking communication operations. While these operations allow computation and communication to be issued concurrently, effective overlapping can only be achieved when asynchronous communication mechanisms are available that ensure communication progresses independently in the background. This approach not only reduces idle time caused by communication but also improves overall utilization of computational

resources. With the continuous advancement of hardware, this technique is expanding rapidly [5–8], laying a solid foundation for the ongoing development of HPC.

Computation–communication overlap techniques have been widely adopted in various numerical weather prediction models to mitigate communication overhead. These include the Fifth Generation National Center for Atmospheric Research/Pennsylvania State University Mesoscale Model (MM5), the Advanced System for Urban Climate Applications (ASUCA), the Weather Research and Forecasting model (WRF), and the Integrated Forecasting System (IFS) dynamical core developed by the European Centre for Medium-Range Weather Forecasts (ECMWF) [9,10]. In China, the National University of Defense Technology has implemented two overlapping strategies for the semi-Lagrangian scheme in the Yin-He Global Spectral Model (YHGSM). The first involves partitioning the interpolated arrays into three independent sub-arrays and reorganizing the communication and interpolation procedures. The experiments show that, by employing non-blocking operations, the interpolation cost at each time step is reduced about 0.2 s, and the total execution time for a 10-day forecast is reduced by around 264 s for a 128-core configuration and 260 s for a 224-core configuration, respectively [11]. The second strategy involves grouping the vertical levels. The 93 levels involved in the semi-Lagrangian interpolation are divided into three groups, enabling the overlapping of communication in one group with the computation in another. This technique effectively hides communication overhead, resulting in a maximum execution time reduction of 12.5% in the semi-Lagrangian scheme, significantly enhancing the overall efficiency of YHGSM [12].

In recent years, the technique of computation–communication overlap has attracted significant global attention and research efforts. Researchers have proposed a variety of approaches, focusing on aspects such as algorithm design, programming model optimization, and hardware support [13–16]. Jichi Guo et al. introduced a systematic approach for automatically overlapping communication with computation in MPI applications. Their method involves performance modeling, optimization analysis, and program transformation, achieving speedups ranging from 3% to 88% across seven NPB benchmarks on clusters with different network configurations [17]. In 2020, C. R. Barbosa et al. proposed three methods for addressing overlap issues within recursive task-based runtime systems. By inserting dedicated progress tasks into the Intel TBB task graph, their approach achieved significant performance gains of up to 11% in matrix multiplication benchmarks [5]. Takashi Soga et al. explored the potential of MPI+OpenMP models for enabling overlapping on vector supercomputers such as SX-ACE and SX-Aurora TSUBASA. Their study focused on the Successive Over-Relaxation (SOR) method in thermal plasma flow simulations. Results published in [18] demonstrated that the proposed 'Manual' model could hide up to 90% and 80% of MPI communication time on SX-ACE and SX-Aurora TSUBASA, respectively.

However, achieving effective computation–communication overlap based on MPI non-blocking communication remains a complex and contentious challenge, with no universally accepted solution to date [19]. This survey aims to provide a comprehensive overview of the current research landscape, offering valuable insights that inform the development of future standards. In addition, by highlighting recent applications in scientific computing, this work serves as a reference for the design and optimization of similar applications in future.

2. MPI Communication Mechanisms

The MPI has become the de facto standard for inter-process communication in the HPC field due to its flexibility and portability. This section introduces the basic principles of MPI communication, including its underlying communication model and the distinction between blocking and non-blocking operations. Understanding these mechanisms is essen-

tial for optimizing parallel programs, particularly in terms of overlapping communication with computation.

2.1. The MPI Communication Model

MPI is a standard protocol used for inter-process communication in parallel computing and widely used in HPC. The core challenge of MPI applications is to efficiently reduce communication overhead between processes for scalability. MPI supports data transfer through various protocols (e.g., TCP, RDMA), each with distinct performance characteristics in terms of latency, bandwidth, and others [20].

In MPI, communication operations can be divided into two main types: point-to-point and collective. Point-to-point operation is the most basic form, involving direct data transfer between two processes, where one process sends a message and the other receives it. Collective operations involve data exchange among multiple processes such as broadcast, scatter, or all-to-all, and has a significant impact on performance of large-scale parallel systems. Although collective operations provide higher-level abstractions, they are typically built upon point-to-point communication mechanisms. Their implementations tend to be more complex and can vary across different MPI libraries. For a more detailed discussion of collective communication, we refer readers to [21]. Nevertheless, a solid understanding of point-to-point communication remains essential, as it forms the foundation for the MPI communication model.

For MPI point-to-point communication, it can be implemented through various mechanisms, but is generally categorized into two protocols based on message size: the eager protocol for small messages and the rendezvous protocol for large ones [22]. These protocols have various specific implementations depending on different scenarios.

When the eager protocol is used, for send, there are typically two cases as illustrated in Figure 1: for very small messages, the data is first buffered and then transmitted via the network interface card (NIC, a hardware component responsible for managing data transmission between nodes in a computer network); for somewhat larger messages, the data is sent directly to the destination process. For receive, in the eager protocol, the receive operation can again be divided into two cases as illustrated in Figure 2. One is the unexpected receive, where the data has already arrived and is stored in the system buffer before the operation is posted. The other is the expected receive, where the operation is posted before the message arrives.

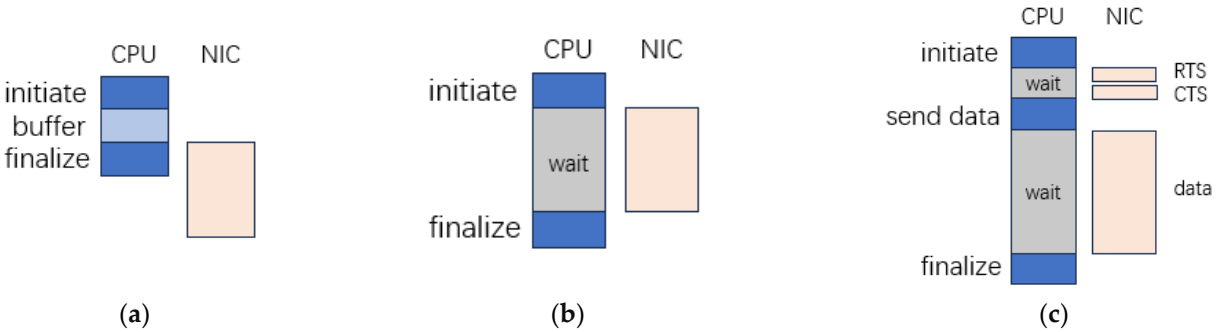


Figure 1. Three Message Sending Modes: (**a**) Buffered Eager Sending; (**b**) Standard Eager Sending; (**c**) Rendezvous Sending.

The rendezvous protocol requires a handshake between the sender and receiver before the actual data transfer can occur, ensuring that the receiver has posted a matching receive buffer. This avoids buffer overflows and is typically used for large messages. In contrast, the eager protocol transmits data immediately without waiting for the receiver to be ready, relying on pre-allocated buffers on the receiver side.

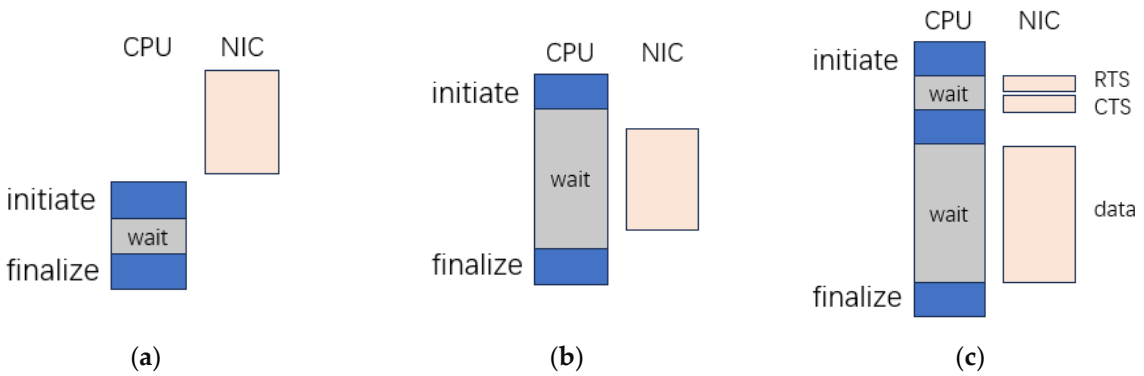


Figure 2. Three Message Receiving Modes in MPI point-to-point communication: (**a**) Eager Unexpected Receiving—the message arrives before the receive is posted and is temporarily stored in a system buffer; (**b**) Eager Expected Receiving—the receive is posted before the message arrives, and data is delivered directly to the user buffer; (**c**) Rendezvous Receiving—the receiver first posts the receive, the sender then performs a handshake (RTS/CTS) before sending the data.

The communication cost of data transmission over a network can be expressed as:

$$T_{comm} = \boldsymbol{Latency} + \boldsymbol{MsgSize}/\boldsymbol{Bandwidth} \quad (1)$$

This formula indicates that the communication cost consists of two components: the latency and a bandwidth-related term that depends on the message size. Latency is primarily influenced by the relative distance from the sender to the receiver and dominates the cost for small messages. Bandwidth, on the other hand, plays a more significant role for large messages. In MPI library implementations, the choice of communication protocols is closely tied to this cost model. Specifically, different protocols are adopted based on message size to optimize performance [23]. For small messages, the eager protocol is preferred as it assumes that the receiver has sufficient memory, allowing the sender to transfer messages immediately. For large messages, the rendezvous protocol is employed, using a "handshake" mechanism to ensure that the receiver has allocated adequate memory before transmission begins, thereby preventing buffer overflow and optimizing the utilization of bandwidth.

2.2. Non-Blocking Communication in MPI

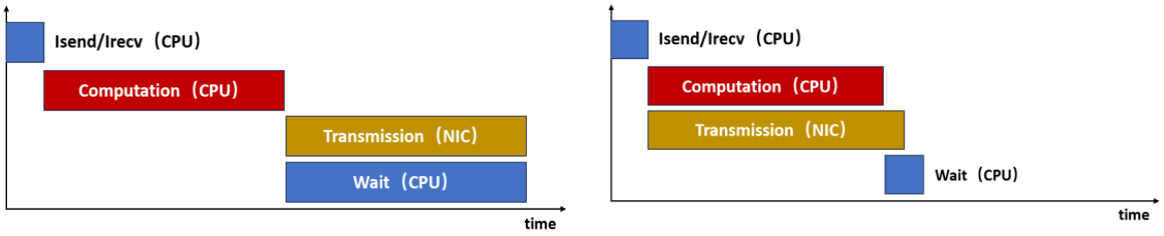
MPI communication can be categorized into blocking and non-blocking modes. In blocking mode, the sender blocks until the data is fully transmitted, and similarly, the receiver waits until the expected data arrives. In modern supercomputing systems, where dedicated hardware such as NIC is typically employed, blocking mode may lead to inefficient utilization of computational resources. To address this problem, non-blocking mode can be used to overlap communication with computation, where the communication operations can proceed in the background while the processes continue to compute. This asynchronous mechanism allows for more efficient utilization of system resources.

The concept of non-blocking communication was first introduced in the MPI-1 standard in 1994, which provided basic non-blocking point-to-point communication functions. The MPI-2 standard, introduced in 1997, extended this functionality by supporting non-blocking file I/O operation and one-sided communication. In 2012, the MPI-3 standard introduced non-blocking collective communication. These advances are aligned with the modern interconnects such as InfiniBand, RoCE (RDMA over Converged Ethernet), Cray's Gemini and Aries networks [24].

From a semantic perspective, non-blocking MPI communication requires users to explicitly manage the lifecycle of communication buffers, ensuring that these buffers are

not accessed until the operation has completed. MPI tracks the status of each non-blocking operation through request handles, which can be queried or synchronized using functions such as MPI_Wait and MPI_Test. The core idea to overlap communication with computation is to do useful computations concurrently when the communication is performed.

As illustrated by the grey regions in Figures 1 and 2, blocking communication introduces idle periods where the CPU remains inactive while waiting for data transfer to complete. With non-blocking communication, the CPU is free to compute immediately after initiating the communication operation, regardless of whether it is a send or receive. Non-blocking operations conceptually allow to overlap communication with computation, but the actual effectiveness depends on the availability of system-level or hardware-supported asynchronous communication mechanisms. Non-blocking operations do not guarantee actual progress in overlapping, as data transfers are often postponed until MPI_Wait is invoked—merely deferring, not avoiding, the blocking phase [25]. This behavior is illustrated in Figure 3. Effective overlapping can only be realized with proper asynchronous support, which may incur additional overhead and complexity.



(**a**) (**b**)

Figure 3. Non-blocking Communication Semantics: (**a**) No overlap; (**b**) Ideal computation–communication overlap.

3. Key Techniques to Overlap Communication with Computation

Achieving effective overlap between communication and computation is a key strategy for improving the scalability and performance of parallel applications. While non-blocking MPI operations provide a basic mechanism for enabling such overlap, their effectiveness heavily depends on both software and hardware support. This introduces core definitions and metrics for evaluating overlap, discusses mechanisms for ensuring communication progress with a focus on asynchronous threading, and reviews recent developments in hardware offloading and task scheduling that contribute to improved overlap efficiency.

3.1. Relevant Definitions and Evaluation Metrics

To achieve ideal computation–communication overlap as shown in Figure 3b, it is not sufficient to use MPI non-blocking communication functions only. A robust MPI progress mechanism—particularly an independent progress mechanism—as well as hardware-based communication offloading, is also essential.

Independent MPI progress refers to communicate asynchronously while the process is performing computation, without requiring explicit MPI calls such as MPI_Wait or MPI_Test to drive progress. This so-called strong progression can be implemented by the MPI library using background threads that periodically poll communication status. Different MPI implementations support varying progression models, including:

- Cooperative (weak progression);
- Coercive (strong progression via interrupt-driven mechanisms);
- Co-located (strong progression using multitasking);
- Concurrent (strong progression using dedicated CPU cores);

- Offloaded (strong progression using dedicated non-CPU hardware).

As discussed in [26], strong progression offers several advantages over weak progression. These include simplifying MPI specifications, improving performance portability, and supporting multi-language parallel programming environments. Although strong progression remains an optional feature in the MPI standard, it significantly enhances the potential to overlap communication with computation. Moreover, it is supported by most modern hardware platforms, rendering weak progression unnecessary in many scenarios.

Network offloading refers to the ability that MPI communication operations are partially or fully handled by network hardware, thereby minimizing the impact on CPU computation. A prominent example is RDMA, which allows data to be transferred directly between memory of different nodes without CPU intervention. InfiniBand is one of the most widely used interconnects that supporting RDMA. Additionally, certain high-performance NICs, such as the Mellanox ConnectX series, support offloading MPI operations [27].

Overlapping, in this context, is a capability of the network layer that enables data transmission without CPU intervention. This allows CPU to focus on computation while communication is handled by other dedicated hardware of the underlying network. The effectiveness of this overlap can be quantitatively evaluated using the Overlapping Ratio, which is defined as Equation (2), where T_overlap is the time period during which communication and computation are proceeding concurrently, and T_comm is the total communication time.

$$\textbf{\textit{Overlap Ratio}} = T_{overlap} / T_{comm} \tag{2}$$

The overlapping ratio ranges from 0 to 1, where a value closer to 1 indicates a higher degree of overlap. Measuring overlap typically relies on analyzing the application's execution timeline. Tools such as TAU or Intel Trace Analyzer can capture the time spent on communication and computation, enabling calculation of the overlapping ratio.

The MPI standard does not mandate asynchronous progress for non-blocking communication functions. Therefore, blindly modifying an application without verifying whether the underlying platform supports computation–communication overlap may result in little performance gain. It is advisable to evaluate this capability through targeted microbenchmarks before implementation. In 2022, Medvedev conducted a systematic analysis for existing benchmarks and proposed a new benchmark suite, IMB-ASYNC, to assess and compare the level of computation–communication overlap in various MPI implementations. This suite integrates the advantages of previous benchmarks, supports diverse communication patterns, and adopts a novel methodology to estimate the degree of overlap. Its effectiveness was validated on the Lomonosov-2 supercomputer, and future work aims to extend testing to broader hardware and software configurations [28].

The study in [29] investigates the combined impact of computation–communication overlap, independent progress, and hardware offloading on the performance of applications. Using different MPI implementations, experiments were conducted on Linux clusters and the Accelerated Strategic Computing Initiative(ASCI) Red system. A comprehensive evaluation using microbenchmarks, collective benchmarks, and NAS Parallel Benchmarks (NPB) revealed that the synergy of these three features can reduce the overall execution time by 2% to 20%. Among them, independent progress was found to be a critical factor for performance enhancement, though in certain scenarios it may also incur performance penalties.

3.2. Achieving Progress for Non-Blocking Communication

In the MPI communication model, progress refers to the continuous advancement of communication operations toward their completion. For blocking communication, achieving progress is relatively straightforward, as the CPU is fully engaged in communication

and not performing other computations during this phase. In contrast, non-blocking communication presents a more complex scenario, critically depending on the underlying progress mechanism.

The MPI standard specifies a general rule for progression in asynchronous operations, which has led to two different interpretations, both considered compliant. The strict interpretation—also referred to as strong progression, as discussed in Section 3.1—requires that once a non-blocking communication operation is issued, it must continue to make progress autonomously, without requiring any further MPI library calls. The weaker interpretation, by contrast, requires the application to invoke additional MPI functions to drive communication progress, as shown in Figure 4 [30].

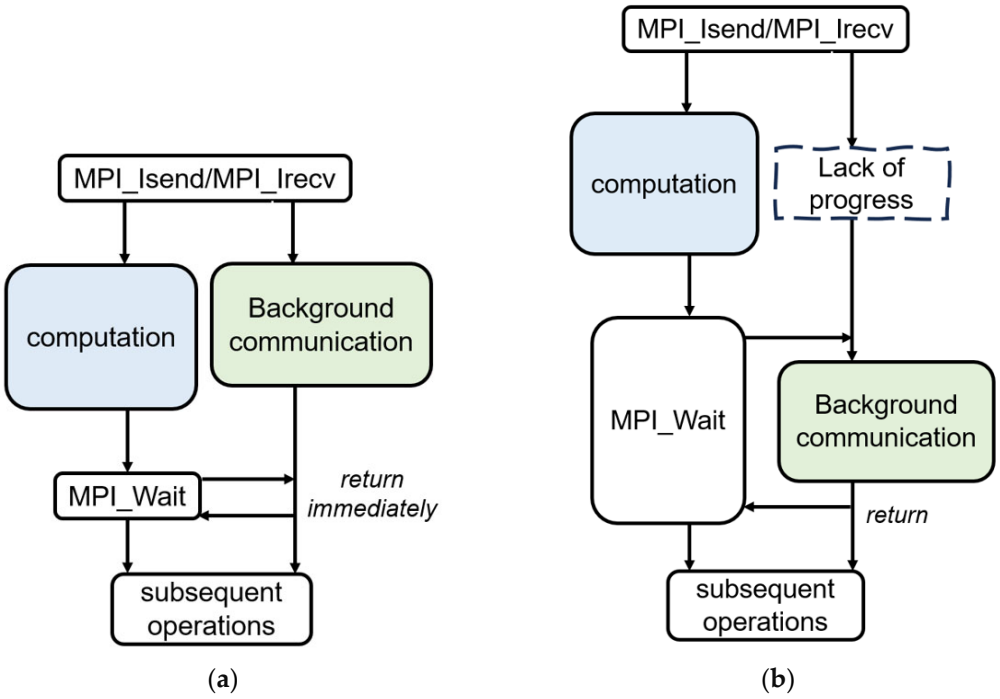


Figure 4. Non-blocking operation: (**a**) Strong progression. (**b**) Weak Progression.

While strong progression appears more conducive to overlapping communication with computation, it may introduce overhead due to background threads or hardware polling. In many real-world scenarios where overlapping opportunities are limited, weak progression implementation that minimizes latency may offer better performance [31].

Achieving strong progress is especially challenging for collective communication. Not only must the data transfers make progress, but the collective algorithms themselves must also advance, making hardware-based progress mechanisms much harder to implement. CPU intervention is often unavoidable. Due to the complexity and inefficiency of current implementations of non-blocking collective operations, their adoption in practice remains limited. This, in turn, reduces the incentive for MPI implementers to optimize such capabilities. Nevertheless, asynchronous progression remains a critical enabler for computation–communication overlap, and managing a diverse range of communication requirements efficiently in the background continues to be a major challenge in large-scale HPC applications [32].

To reduce the CPU intervention in communication, OS-bypass networking technologies such as Myrinet, Quadrics, and InfiniBand were developed, which allow NICs to perform communication functions asynchronously. However, these solutions often lack sufficient support for full MPI semantics. As a result, achieving such progress may still incur considerable CPU overhead [33].

From an application developer's perspective, there are currently three main approaches to achieve background progress in MPI communications.

1. Manual Progress via Explicit MPI Calls:

A common approach is to manually drive communication progress through explicit MPI calls, which is to significantly modify the application codes—such as repeatedly calling MPI_Test during the computation phases to achieve communication overlap, as illustrated in Figure 5.

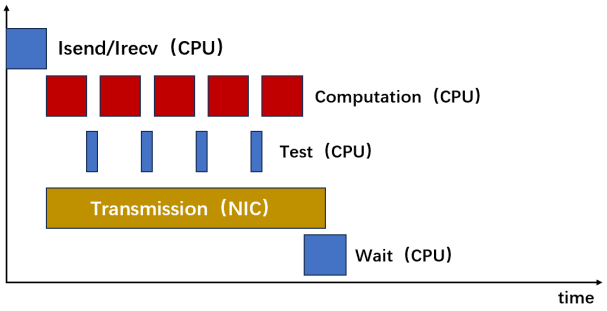


Figure 5. Inserting MPI_Test calls in a computation loop to explicitly advance communication progress (ideal case).

However, this approach has a critical drawback, that is, the placement and frequency of MPI_Test calls are entirely controlled by the programmer, while the actual state of communication progress is essentially a black box. Therefore, developers must resort to repeated trial-and-error to find appropriate call timings. Calling MPI_Test too early or too late often results in missed opportunities for overlapping. In real-world applications, particularly when computation is handled by external scientific libraries, it can be extremely difficult to insert MPI_Test calls at appropriate positions, making it even harder to achieve effective overlapping.

Under this manual progress mode, performance is often evaluated using Equation (3). Here, size denotes the scale of the computation or data, interval refers to the time elapsed between two successive MPI_Test calls, and N represents the number of calls. By tuning interval, one can experimentally determine a calling pattern to improve the performance.

$$N = \lfloor size/interval \rfloor + 1 \tag{3}$$

2. Offloading Progress to Dedicated Hardware:

The second approach is to offload progression to dedicated hardware, which involves offloading MPI communication to dedicated hardware, and is considered to be the ideal solution. It requires no additional intervention from the programmer and prevents resource wastage. However, its implementation heavily relies on hardware support and requires tight hardware–software coordination [34]. This approach faces practical challenges: the diversity of existing hardware often necessitates custom implementations, which often require significant investment with limited performance gains. Consequently, achieving optimal computation–communication overlap through this approach is very difficult. Furthermore, offloading protocol processing to NICs creates performance pressures, that is, as more network functions get offloaded, the NIC's embedded processors can quickly become performance bottlenecks [35].

3. Use of Asynchronous Progress Threads:

The third approach is to use asynchronous progress threads, which involves enabling asynchronous progress threads, which represents a compromise between portability and

performance. This approach is widely supported by mainstream MPI implementations such as MPICH, MVAPICH, and OpenMPI. For instance, in MPICH 3.2, asynchronous progress can be activated by setting the environment variable MPICH_ASYNC_PROGRESS = 1. MVAPICH further improves upon this design. With this method, MPI automatically spawns a dedicated progress thread for each process at runtime. Once the main thread initiates a communication operation, it continues computation, while a dedicated progress thread handles the communication progression [36].

However, this mechanism introduces challenges such as lock contention and competition for CPU cores [37]. As illustrated in Figure 6, assigning both the main thread and the progress thread to the same core can lead to resource contention and degraded performance. If resources permit, it is recommended to allocate a dedicated core for the progress thread, as shown in Figure 6a.

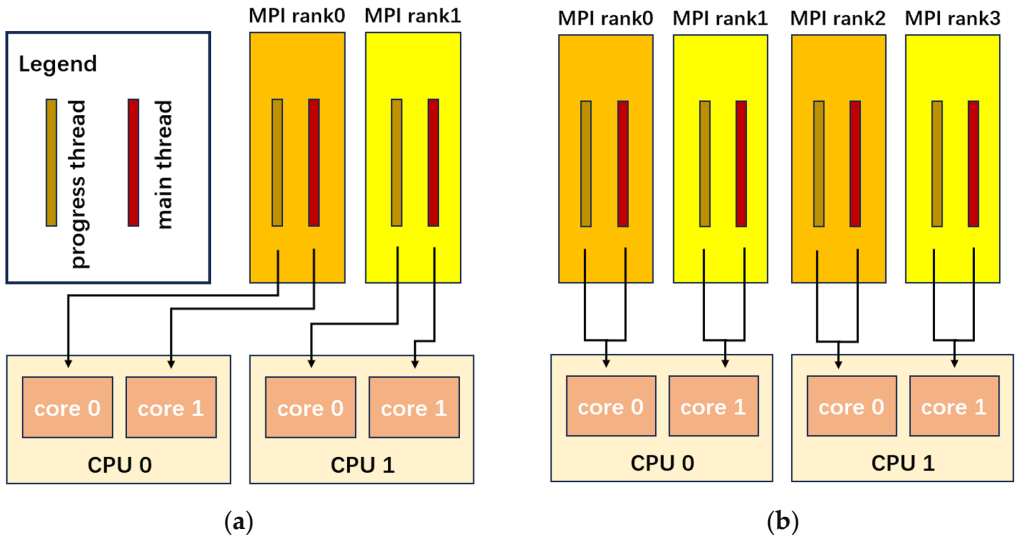


Figure 6. Two allocation strategies for asynchronous progress threads: (**a**) dedicated core; (**b**) oversubscribed (shared core).

There are generally two progression models based on threading: polling-based and interrupt-based. In the polling model, each MPI process has a dedicated thread that continuously polls the MPI engine to handle incoming messages, providing immediate data availability. In contrast, the interrupt-based approach uses hardware interrupts to awaken the thread only when progression is required. Although this method avoids constant polling and conserves CPU resources, it incurs overhead due to operating system involvement.

Reference [38] analyzes three types of progression strategies over InfiniBand networks—manual progression, hardware-assisted progression, and thread-based progression—and proposes mitigation techniques such as intelligent aggregation, signal mechanisms, kernel-level and hardware-level implementations to address the performance overheads of thread-based progression.

3.3. Advances in Supporting Hardware and Software Technologies

In recent years, optimizing computation–communication overlap based on MPI has become a prominent research focus in the HPC field. Researchers have achieved innovations from multiple aspects, including software optimization, hardware offloading, and algorithm design.

At the software level, a variety of methods have been proposed to enhance the progress mechanisms of MPI libraries and to improve the effectiveness of overlapping. For instance, Nguyen et al. [39] proposed an LLVM-based compiler optimization that transforms block-

ing MPI calls into non-blocking ones and extends the overlap window through code motion. By analyzing dependencies to safely reposition MPI calls, their method significantly increases overlap duration, reducing communication stalls and improving computation–communication overlap in real-world HPC applications.

Thread-based asynchronous progress techniques have also been studied extensively. As described in [24], an enhanced design allows the MPI library to autonomously detect the need for non-blocking communication progress, thereby reducing context switches and contention between the main and progress threads. This significantly improves independent progress capabilities and demonstrates considerable performance gains in both microbenchmarks and real-world applications across various hardware architectures. Nigay et al. [40] addressed communication and timing issues in MPI virtualization, which achieves implicit overlap and load balancing by mapping MPI processes to threads and oversubscribing CPU cores. They proposed low-overhead process- and core-level timers to correct wall-clock timing inaccuracies and found that full overlap under the rendezvous protocol requires at least three MPI processes per core. Their work offers practical insights for designing efficient virtualized MPI systems.

In 2024, the MPICH development team at Argonne National Laboratory emphasized that the MPI standard lacks a well-defined, interoperable progression mechanism, which poses challenges for performance and compatibility with modern programming paradigms such as task-based and event-driven models [41]. The team proposed a set of MPI extensions aimed at exposing internal progress-related mechanisms. This transparency empowers developers to construct and manage customized progress engines more effectively, thereby facilitating better support for computation–communication overlap.

Secondly, in terms of hardware offloading, researchers are actively exploring various approaches to offload the processing of MPI communication protocols to dedicated hardware. Bayatpour et al. [42] proposed CHAMPION, a hardware-assisted MPI engine that improves overlap efficiency by combining hardware tag matching with software prefetching and adaptive, history-based heuristics. On Frontera, it reduced collective latency by 41% and improved P3DFFT communication by 23%, showcasing the benefits of hardware–software co-design for scalable MPI performance.

In 2021, Sarkauskas et al. [43] investigated the use of the BlueField-2 DPU to offload non-blocking MPI_Ibcast and MPI_Iallgather operations. They proposed two algorithmic designs—Flat and Hierarchical—and demonstrated through experiments that the BlueField-2 DPU serves as an effective hardware for offloading. Their findings highlighted that, compared to traditional CPU-based approaches, the DPU-based design's most significant advantage lies in its ability to enable true computation–communication overlap.

In 2024, Liang et al. from the National University of Defense Technology addressed the performance limitations of traditional MPI broadcast algorithms on modern high-speed interconnects [44]. They optimized MPI broadcast operations for the TH-Express interconnect architecture used in Tianhe supercomputers. They completely offload MPI broadcast operations to NIC, while intelligently selecting different offloading strategies based on message size. By taking full advantage of the NIC's collective communication capabilities, their approach demonstrated measurable performance gains. Experiments on both Tianhe-2A and Tianhe-EP supercomputers showed significant performance benefits, achieving a speedup of 1.34× for the scientific application LAMMPS.

These studies demonstrate that effective hardware–software co-design can greatly enhance the development and performance of high-performance applications, especially in enabling communication–computation overlapping.

Finally, in terms of algorithm design, recent research has focused on optimizing MPI's integration with parallel programming models to better leverage computation–

communication overlap. As HPC systems increasingly adopt the many-core architecture, effectively combining MPI with shared-memory models like Intel TBB has become crucial. However, achieving overlapping in recursive task graph environments remains challenging. To address this challenge, researchers in [5] proposed three methods—Root, Non-leaves, and Colored tasks—which improve the degree of overlap by inserting dedicated progress tasks at optimal positions within the task graph. Benchmark results demonstrated that the performance improvements can be up to 11%.

With the rise of many-core and heterogeneous systems, Asynchronous Task-based Programming models (ATaP) have attracted increasing attention. These models can automatically exploit computation–communication overlap to improve performance. However, the degree of overlap is often limited, and its effectiveness when interacting with MPI still restricts the actual overlap achieved. To address this limitation, Castillo et al. [45] proposed an approach that integrates MPI's internal information into the ATaP runtime system to optimize task creation and scheduling decisions. Benchmark results in MPI and OmpSs environments showed significant performance improvements: up to 16.3% for point-to-point communication benchmarks and up to 34.5% for collective ones.

Moreover, for application developers, the performance of the MPI library is critical to achieving efficient overlap. Selecting a highly optimized MPI implementation and enabling advanced configuration options, such as asynchronous progress threads or tuning communication buffer sizes, can lead to significant performance improvements, particularly when tailored to the characteristics of the underlying hardware. On the hardware side, most network fabrics continue to rely on the host CPU for tasks such as communication matching and queue traversal. This reliance can hinder the effectiveness of overlap. However, certain interconnect technologies, including Quadrics, have introduced alternative mechanisms to address this limitation [46].

In summary, the synergy between software optimization, hardware capabilities, and programming model extensions provides a solid foundation for enabling effective computation–communication overlap in HPC environments, paving the way for more efficient architectural designs in future.

4. Applications of Computation-Communication Overlap

This chapter presents practical applications of computation–communication overlap in both controlled experiments and real-world HPC scenarios. We begin by analyzing the performance implications of enabling overlap using the MPICH3 implementation, based on tests conducted across different platforms. We then explore how overlap techniques have been successfully applied in large-scale scientific applications. These examples demonstrate the tangible benefits of overlap in improving computational efficiency and reducing communication bottlenecks in modern HPC workloads.

4.1. Comparative Analysis of the Performance Using MPICH3

To enable computation–communication overlap, the program structure should typically be transformed from the pattern depicted in Figure 7a to that in Figure 7b, utilizing MPI non-blocking communication.

To evaluate the performance of computation–communication overlap with MPICH3, we conducted comparative experiments using the workflows illustrated in Figure 7. Each test was performed separately under single-node and dual-node configurations, where two processes exchanged data of identical size.

The computation segment designed to overlap with communication was scaled proportionally to the communication volume and adjusted so that the computation time slightly exceeded the communication time, thereby enhancing the visibility of potential overlap.

Figure 7a serves as a baseline to measure the raw cost of communication and computation, while Figure 7b focuses on measuring the time spent in the MPI_Wait call, representing the communication time not overlapped by computation. Comparing these two measurements provides a clear indication of the effectiveness of computation–communication overlap. The results thus obtained reflect the practical overlap capabilities achievable under typical runtime conditions across different platforms.

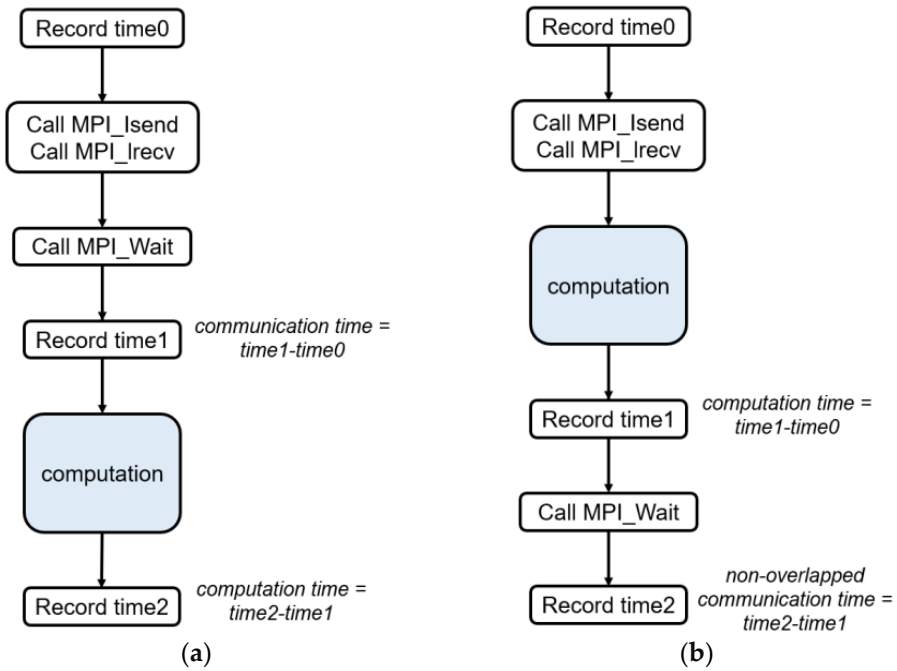


Figure 7. (**a**) Baseline workflow, where communication and computation are executed sequentially (to facilitate separate measurement of their respective durations); (**b**) Optimized workflow utilizing non-blocking communication to enable computation–communication overlap.

We first conducted measurements on single-node platforms, specifically using personal laptop computers with multi-core architectures. In single-node settings, due to the shared memory architecture with a unified address space, MPI communication is effectively carried out as memory copy operations rather than true network transfers. Two different laptop models were selected for this analysis, with their hardware configurations summarized in Table 1.

Table 1. Hardware Specifications of Two Single-Node Platforms.

Configuration Item	Platform A	Platform B
CPU	Intel(R) Core(TM) i5-10210U CPU @ 1.60GHz	13th Gen Intel(R) Core(TM) i9-13900H
Thread(s) per core	2	2
Core(s)	4	10
Cache	L1d: 128 KiB; L1i: 128 KiB; L2: 1 MiB; L3: 6 MiB	L1d: 480 KiB; L1i: 320 KiB; L2: 12.5 MiB; L3: 24 MiB
Memory	16 GB	32 GB
OS	Windows 11 + WSL2 + Ubuntu 20.04.4 LTS	Windows 11 + WSL2 + Ubuntu 20.04.6 LTS

The experimental results from Laptop A and Laptop B are similar, as illustrated in Figure 8. As in Figure 8a,c, simply using non-blocking communication without considering implementation details can fail to achieve actual computation–communication overlap. In the non-blocking version, the communication primarily occurs during the MPI_Wait

call, concentrating the communication overhead at that point rather than hiding it behind computation. This behavior aligns with the weak progression semantics of MPICH 3.2, which is the default. As a widely used MPI implementation, MPICH 3 prioritizes portability and broad compatibility over platform-specific performance optimizations, which explains this behavior.

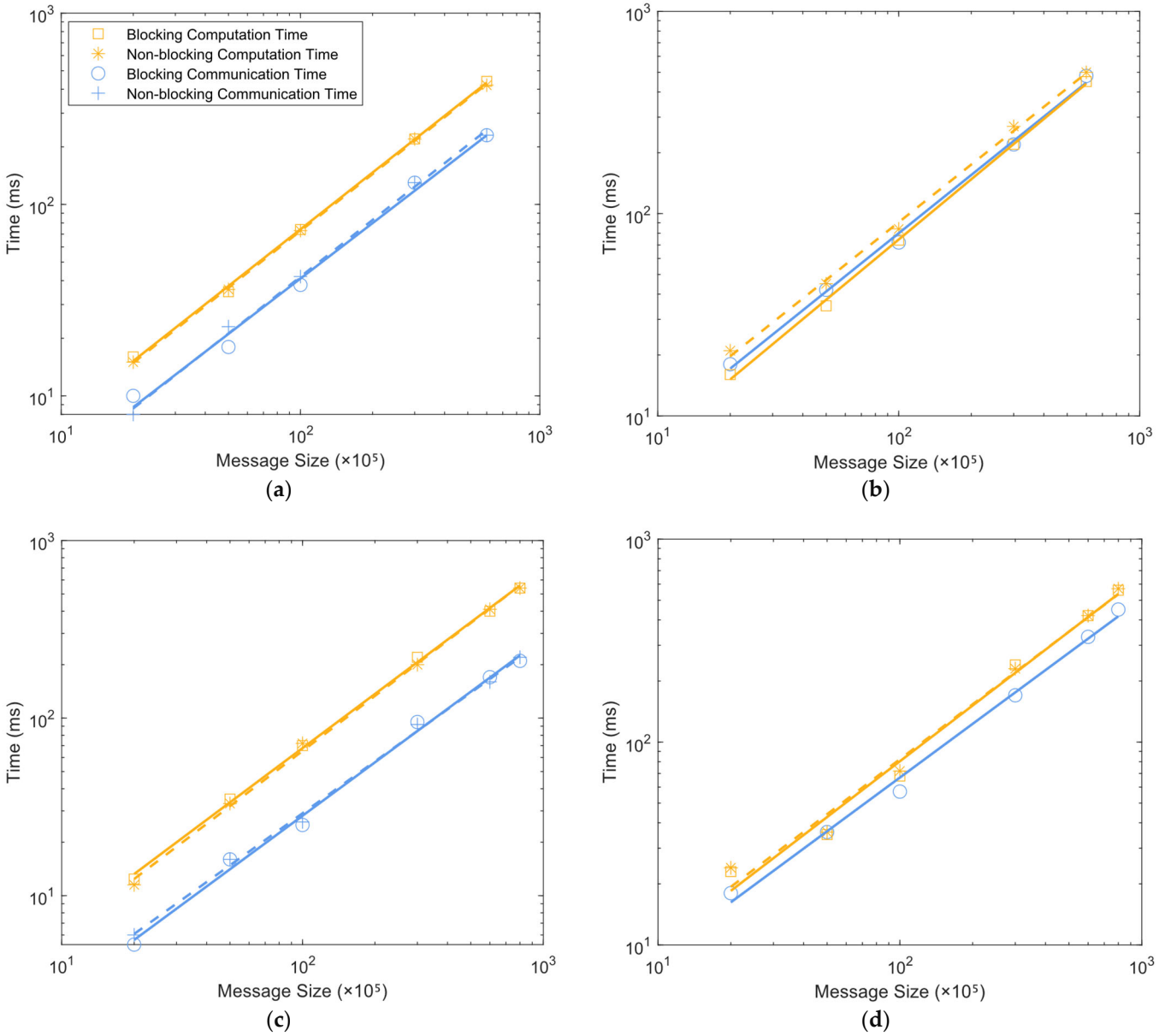


Figure 8. Performance results: (**a**) Laptop A—asynchronous progress thread disabled (default configuration); (**b**) Laptop A—asynchronous progress thread enabled with a dedicated physical core; (**c**) Laptop B—asynchronous progress thread disabled (default configuration); (**d**) Laptop B—asynchronous progress thread enabled with a dedicated physical core.

To facilitate computation–communication overlap, MPICH3 provides an alternative mechanism for achieving strong progress: the asynchronous progress thread, as discussed in Section 3.2. The effects of this mechanism are illustrated in Figure 8b,d, when asynchronous progress is enabled, and each MPI process is allocated two physical threads. Compared to Figure 8a,c, the computation time remains largely unchanged, while the communication time in the non-blocking version is effectively hidden—the overhead of the MPI_Wait call is negligible and thus not shown. However, we also found a drawback: the

blocking communication overhead increases relative to the default configuration. This is likely due to the additional interactions between the progress and main threads, introducing some overhead. This finding strongly suggests that for applications where blocking communication dominated the workload, employing asynchronous progress threads may not yield meaningful performance benefits.

Subsequently, we extend our evaluation to two multi-node HPC platforms, each equipped with two compute nodes and one MPI process is used for each node. In this configuration, MPI communication primarily relies on the interconnect network between nodes, introducing greater complexity compared to the intra-node memory-copy mechanism in the single-node scenario. The configurations of the two platforms are summarized in Table 2.

Table 2. Hardware Specifications of Two Multi-Node Platforms.

Configuration Item	Laptop A	Laptop B
CPU	Intel(R) Xeon(R) Platinum 9242 CPU @ 2.30GHz	Intel(R) Xeon(R) Gold 6150 CPU @ 2.70GHz
Thread(s) per core	1	1
Core(s)	96	36
Interconnect	InfiniBand	InfiniBand
Memory	384 GB	186 GB
OS	Linux	Linux

Note: On both platforms, the InfiniBand interconnect is switch-based, representing a typical production HPC environment. The reported memory size (186 GB) reflects the value given by the Slurm resource manager (RealMemory = 190,000 MB). This may not correspond to standard DIMM configurations but represents the usable memory detected by the system.

As shown in Figure 9a,c, the experiments from the two-node platform exhibits similar behavior to previous results, failing to achieve effective computation–communication overlap despite advanced hardware and system support. Figure 9b,d reveal that while the computation time for both blocking and non-blocking versions remain nearly identical, the non-blocking version significantly reduces communication time. This confirms that dedicating a physical core to asynchronous progress threads is an effective strategy. However, the increase in blocking communication overhead remains a critical observation. Comparing Figure 9a,b, we note that enabling asynchronous progress threads causes the communication time to increase from being shorter than the computation time to exceeding it. As a result, in the non-blocking version as illustrated in Figure 9b, the communication overhead can no longer be fully hidden within the computation phase.

The experimental results on Platform B are largely consistent with those observed previously. Notably, comparing Figure 9c,d, both the communication and computation overheads remain nearly unchanged. Unlike earlier observations, enabling asynchronous progress threads does not incur additional costs in this case. This indicates that, with more efficient system-level implementations and better hardware–software coordination, it is possible to benefit from computation–communication overlap without incurring noticeable additional overhead for blocking communication operations.

The comparison between Figures 10a and 8a, as well as between Figures 10b and 9a, indicates that when restricting each MPI process to a single physical core, both computation and communication costs will increase. Moreover, the system exhibits unstable computation and communication timings due to thread switching overhead. Figure 10a shows just one possible observation, as significant variability was observed in performance measurements across both laptops and platform B, with relatively stable results exhibited only on platform A. While non-blocking communication maintains its ability to effectively hide communication overhead in this configuration, the system fails to deliver any meaningful improvement in the performance of the overall application.

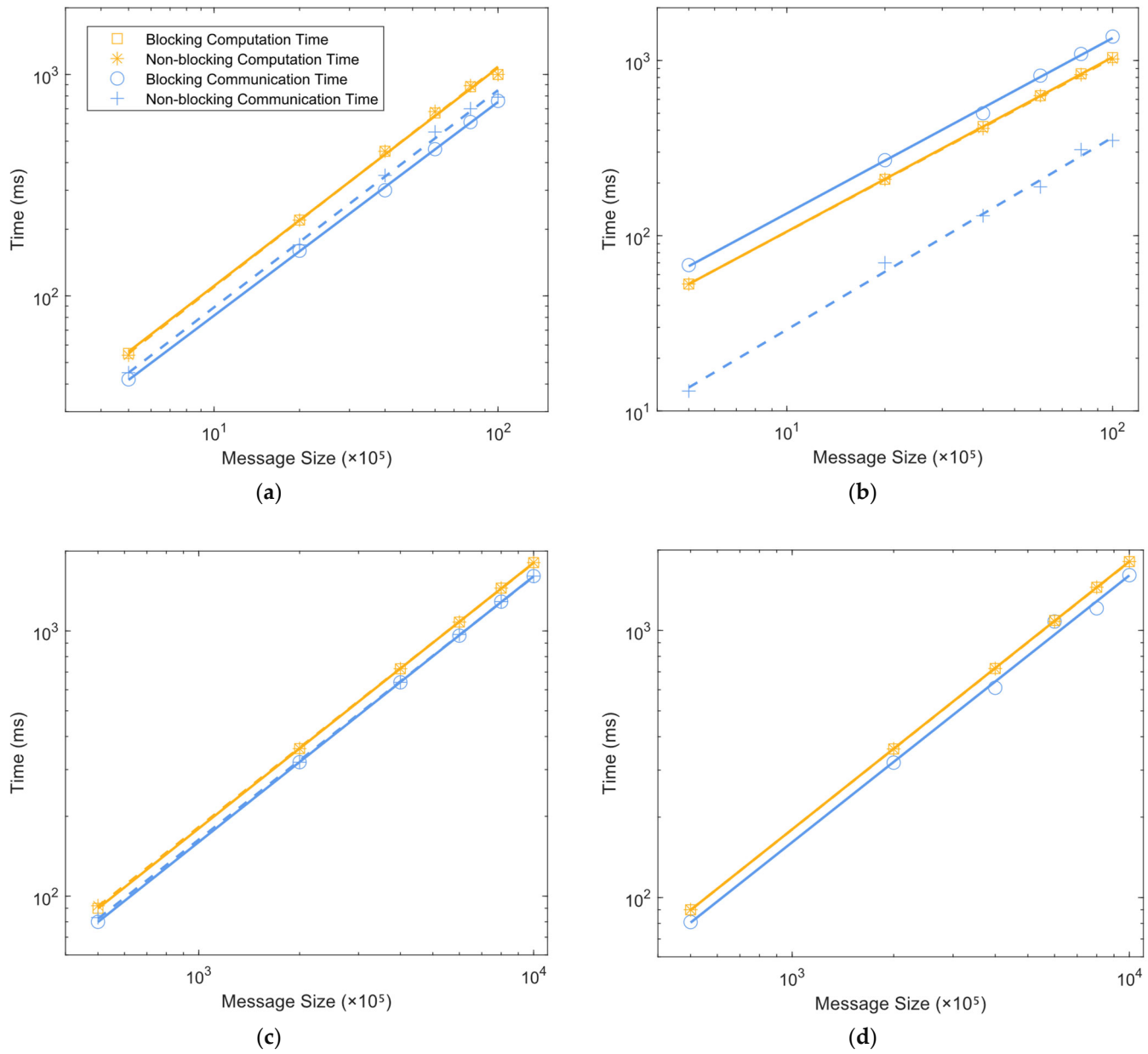


Figure 9. Performance results: (**a**) Platform A—asynchronous progress thread disabled (default configuration); (**b**) Platform A—asynchronous progress thread enabled with a dedicated physical core; (**c**) Platform B—asynchronous progress thread disabled (default configuration); (**d**) Platform B—asynchronous progress thread enabled with a dedicated physical core.

In our MPICH3 experiments, we also evaluated the strategy introduced in Section 3.2—periodically inserting MPI_Test calls within the computation phase. While MPI_Test can indeed advance communication progress, MPICH3's opaque progress mechanism makes optimal placement difficult to determine. Consequently, poorly placed calls may not only nullify potential benefits from overlap but could even introduce additional overhead.

Overall, achieving effective computation–communication overlap remains challenging with the widely used generic MPI implementation MPICH3 when relying solely on non-blocking communication across most platforms. Currently, the most viable approach involves enabling asynchronous progress threads while dedicating a physical core to support them. This strategy effectively leverages surplus CPU resources and can yield substantial performance gains for applications. However, developers must carefully assess potential overhead introduced in blocking communication sections and balance the trade-offs to

maximize performance benefits. Furthermore, a well-optimized system implementation can significantly simplify application-level tuning.

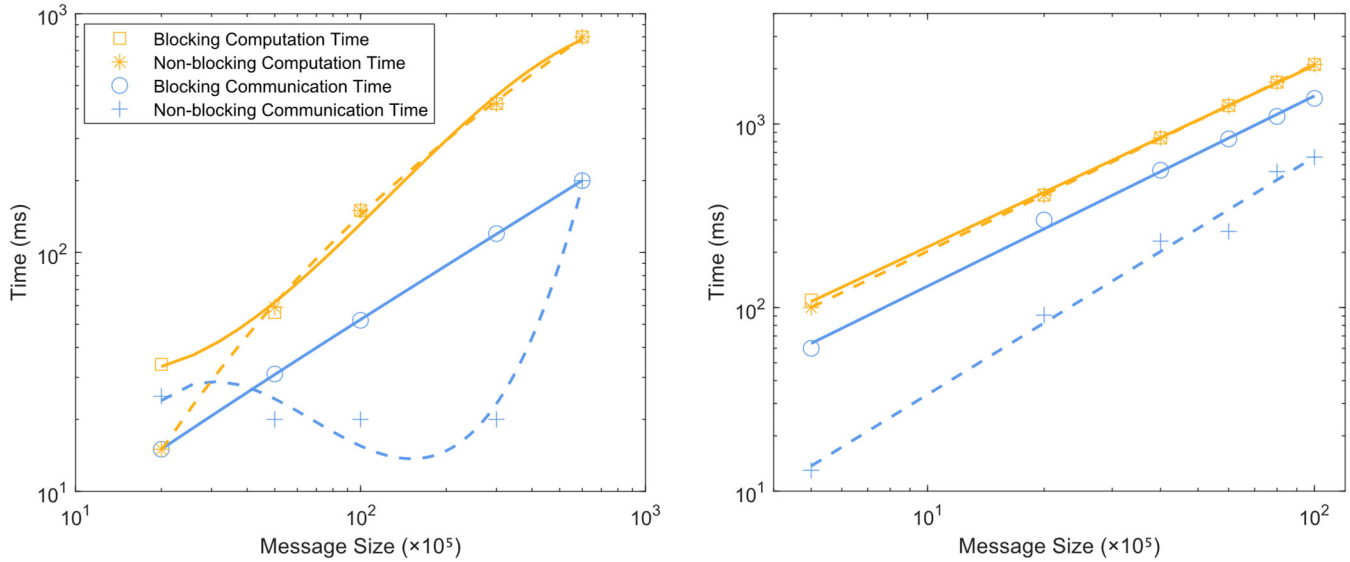


Figure 10. Performance with Asynchronous Progress Thread enabled while sharing a physical core with the main thread: (**a**) Results on Laptop A; (**b**) Results on Platform A.

4.2. Practical Implementations in Supercomputing Applications

In scientific computing applications such as computational fluid dynamics (CFD) simulations and numerical weather prediction, grid-based methods are widely used. These methods typically require extensive data exchanges of boundary information between neighboring processes. By implementing pipeline techniques combined with non-blocking MPI communication, the transfer of boundary data can be performed in parallel with the computation of interior grid points, significantly improving the overall parallel efficiency [47,48].

A notable advancement in this area was demonstrated by Guo et al. in 2020 through their optimization for OpenFOAM, a widely used open-source CFD platform [49]. Their work specifically targeted the performance of the bottlenecks in the Preconditioned Conjugate Gradient (PCG) method by carefully analyzing its communication dependencies. Through the adoption of non-blocking collective communication and loop unrolling, they restructured the operations to overlap communication with computation. Experimental results on a 2D lid-driven cavity flow with a 2048 $\times$ 1024 grid demonstrated that the simulation time can be reduced by 8.0–29.0% when scaling from 64 to 2048 cores.

In HPC applications, data distribution and aggregation are critical operations—for instance, during matrix factorization or linear system solving. Non-blocking communication enables asynchronous data transfer, which can significantly reduce overall runtime by overlapping communication with computation. This technique aligns with pipeline techniques naturally, which decomposes the workflow into independent stages, allowing the communication of one stage and the computation of another performed concurrently. A typical example of this kind of optimization was presented in 2024 by Liu et al. They addressed communication bottlenecks in the Atmospheric General Circulation Model (AGCM) used by the China Beijing Climate Center [50] and proposed a fine-grained pipelining method called Pipe-AGCM based on latitude-based grouping, which allowed overlapping one group's communication with another's computation. By tuning the number of groups, the

method reduced the communication time by up to 78.02% and the total runtime by up to 29.68%.

Krylov subspace methods, such as the Conjugate Gradient (CG) and Generalized Minimal Residual (GMRES) algorithms, are iterative solvers for large sparse linear systems and often involve frequent communication for dot products and vector updates. Non-blocking communication allows to overlap the communication of such operations with other computations like sparse matrix-vector multiplications, to reduce the iteration time. In [51], several computation–communication overlapped Krylov subspace methods were proposed to address the performance bottleneck on distributed memory systems. For instance, the PIPECG-OATI algorithm reduces global reductions to once every two iterations and overlaps them with two preconditioner applications and two matrix-vector multiplications. Similarly, the PIPE-sCG and PIPE-PsCG methods perform global reductions every s iterations, overlapping with s matrix-vector products or s preconditioner steps, respectively.

Effective implementation of overlapping relies heavily on task decomposition and scheduling. By dividing computations into small tasks and dynamically scheduling them according to the communication status, idle periods can be minimized. Load balancing strategies are also essential for evenly distributed computational and communication loads, and then for avoiding performance bottlenecks and enhancing overlap efficiency [52]. In 2024, Nakajima applied overlapping techniques to the forward and backward substitution phases of the Incomplete Cholesky (IC(0)) smoother within the Multigrid Conjugate Gradient (MGCG) method [53]. In parallel finite element (FEM) and finite volume (FVM) methods, internal nodes were categorized into boundary and interior nodes, with communication of the former overlapped by computation of the latter. Results demonstrated that over 40% performance improvement can be achieved on the 4096-node Odyssey system and over 20% improvement can be achieved on the 1024-node Oakbridge-CX system.

In the field of numerical weather forecasting, as models increasingly adopt higher resolutions for improved accuracy, I/O operations involving massive datasets have become critical performance bottlenecks. Traditional I/O mechanisms are inefficient at such scales and often become a bottleneck of the model performance. The use of MPI non-blocking communication for computation–communication overlap offers a promising solution, enabling computation to proceed concurrently with I/O. In 2022, the ECMWF integrated XIOS 2.0 into IFS CY43R3 to address the inefficiency of the original sequential I/O scheme. By allowing IFS processes to asynchronously send data to XIOS servers using MPI non-blocking communication, the model's computing performance was no longer blocked by I/O operations, significantly improving the overall efficiency [54].

Beyond traditional HPC applications, emerging AI workloads increasingly demand efficient computation–communication overlap to enhance scalability and performance. This is especially vital in operations like gradient aggregation and model broadcasting, where overlapping communication with local computation can significantly reduce training time and improve throughput. In 2021, Castelló et al. [55] proposed a pipelined non-blocking MPI_Iallreduce for TensorFlow with Horovod, enabling computation–communication overlap, achieving up to 60% speedup and performance comparable to NVIDIA's NCCL. Building on this, Jangda et al. [56] developed CoCoNet, a DSL and compiler fusing computation–communication operations, reordering tasks, and enabling fine-grained MatMul-AllReduce overlap. Breaking the abstraction barrier, it minimizes overhead and boosts GPT-3 pipeline-parallel inference by 1.77×. For edge AI, Guo et al. [57] introduced AutoDiCE, an automated toolchain for distributed CNN inference on heterogeneous devices, combining MPI and OpenMP with efficient partitioning to reduce memory usage and energy consumption.

In summary, as the scale of supercomputing platforms continues to grow, communication overhead is increasingly becoming a limiting factor for performance. Computation–communication overlap, based on MPI non-blocking communication, offers a viable and effective approach to mitigating this issue [58]. As the synergy among algorithms, software, and hardware continues to improve, this technique is expected to deliver substantial benefits across a growing range of scientific and engineering applications.

5. Conclusion and Outlook

This paper presents a comprehensive review of computation–communication overlap techniques based on MPI non-blocking communication, covering theoretical foundations, implementation methods, optimization strategies, and real-world applications. By analyzing representative methods and case studies, we demonstrate the technique's significant potential in HPC.

However, the practical application of this technique still faces several challenges. Some MPI implementations may not fully exploit the underlying hardware's Remote Memory Access (RMA) capabilities, instead relying on software-based mechanisms such as auxiliary progress threads. This inconsistency complicates efforts to ensure true asynchronous transfer and overlap across platforms. Future work will systematically compare MPI libraries to clarify differences in progress and background transfer handling. Moreover, different network architectures vary in their support and performance characteristics necessitate careful tuning of MPI libraries to accommodate. Developers often lack the guidance to fully exploit hardware capabilities.

As system scales continue to increase, achieving sufficient overlap presents increasing challenges, including dynamic load balancing for tasks of varying sizes, memory management optimized for diverse architectures and access patterns, and communication optimization.

Despite its promising potential, MPI implementations often do not enforce strict interpretations of progress semantics, primarily because most applications lack the opportunity to fully exploit asynchronous transfers. Achieving efficient overlap still requires stronger software–hardware co-design and clearer standardization. Furthermore, resource contention remains a major bottleneck for overlapping techniques, necessitating more robust and scalable solutions.

Looking forward, breakthroughs are expected through the development of novel programming models and the integration of AI techniques. Intelligent runtime systems that combine static code analysis with dynamic optimization, powered by machine learning and deep learning, can automatically adapt computation–communication scheduling, optimize resource allocation, and predict application behavior. Furthermore, deep learning enables automatic code decomposition and dependency analysis, facilitating fine-grained and highly efficient overlap. As hardware architectures continue to evolve and intelligent optimization techniques are further developed, computation–communication overlap will become an essential component for supporting next-generation scientific applications. Addressing current limitations through cross-layer innovations will be critical to fully exploiting the potential of large-scale parallel computing.

Author Contributions: Conceptualization, Y.Z. and J.W.; methodology, Y.Z. and J.W.; investigation, Y.Z. and J.W.; writing—original draft preparation, Y.Z.; writing—review and editing, J.W.; visualization, Y.Z.; supervision, J.W. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by National Natural Science Foundation of China grant number 42375158.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Betzel, F.; Khatamifard, K.; Suresh, H.; Lilja, D.J.; Sartori, J.; Karpuzcu, U. Approximate Communication: Techniques for Reducing Communication Bottlenecks in Large-Scale Parallel Systems. *ACM Comput. Surv.* **2019**, *51*, 1–32. [CrossRef]
2. Wei, X.; Cheng, R.; Yang, Y.; Chen, R.; Chen, H. Characterizing Off-path SmartNIC for Accelerating Distributed Systems. In Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation (OSDI '23), Boston, MA, USA, 10–12 July 2023; pp. 987–1004.
3. Wang, Z.; Wang, H.; Song, X.; Wu, J. Communication-Aware Energy Consumption Model in Heterogeneous Computing Systems. *Comput. J.* **2024**, *67*, 78–94. [CrossRef]
4. Pereira, R.; Roussel, A.; Carribault, P.; Gautier, T. Communication-Aware Task Scheduling Strategy in Hybrid MPI+OpenMP Applications. In *OpenMP: Enabling Massive Node-Level Parallelism, IWOMP 2021*; McIntosh-Smith, S., de Supinski, B.R., Klinkenberg, J., Eds.; Springer: Cham, Switzerland, 2021; Volume 12870, pp. 199–213. [CrossRef]
5. Barbosa, C.R.; Lemarinier, P.; Sergent, M.; Papauré, G.; Pérache, M. Overlapping MPI communications with Intel TBB computation. In Proceedings of the 2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), New Orleans, LA, USA, 18–22 May 2020; pp. 958–966. [CrossRef]
6. Ouyang, K.; Si, M.; Hori, A.; Chen, Z.; Balaji, P. Daps: A Dynamic Asynchronous Progress Stealing Model for MPI Communication. In Proceedings of the 2021 IEEE International Conference on Cluster Computing (CLUSTER), Portland, OR, USA, 7–10 September 2021; pp. 516–527. [CrossRef]
7. Temuçin, Y.H.; Sedeh, A.B.; Schonbein, W.; Grant, R.E.; Afsahi, A. Utilizing Network Hardware Parallelism for MPI Partitioned Collective Communication. In Proceedings of the 2025 32nd Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP), Turin, Italy, 12–14 March 2025.
8. Lescouet, A.; Brunet, É.; Trahay, F.; Thomas, G. Transparent Overlapping of Blocking Communication in MPI Applications. In Proceedings of the 2020 IEEE 22nd International Conference on High Performance Computing and Communications; IEEE 18th International Conference on Smart City, Yanuca Island, Fiji, 14–16 December 2020; pp. 744–749. [CrossRef]
9. Liu, D.; Wu, J.; Pan, X.; Wang, Y. The development of overlapping computation with communication and its application in numerical weather prediction models. In Proceedings of the International Conference on Electronic Information Technology (EIT 2022), Chengdu, China, 23 May 2022; Volume 12254. [CrossRef]
10. Liu, H.; Lei, K.; Yang, H.; Luan, Z.; Qian, D. Towards Optimized Hydrological Forecast Prediction of WRF-Hydro on GPU. In Proceedings of the 2023 IEEE International Conference on High Performance Computing & Communications, Melbourne, VIC, Australia, 17–21 December 2023; pp. 138–145. [CrossRef]
11. Jiang, T.; Wu, J.; Liu, Z.; Zhao, W.; Zhang, Y. Optimization of the parallel semi-Lagrangian scheme based on overlapping communication with computation in the YHGSM. *Q. J. R. Meteorol. Soc.* **2021**, *147*, 2293–2302. [CrossRef]
12. Liu, D.; Liu, W.; Pan, L.; Zhang, Y.; Li, C. Optimization of the Parallel Semi-Lagrangian Scheme to Overlap Computation with Communication Based on Grouping Levels in YHGSM. *CCF Trans. High Perform. Comput.* **2024**, *6*, 68–77. [CrossRef]
13. Punniyamurthy, K.; Hamidouche, K.; Beckmann, B.M. Optimizing Distributed ML Communication with Fused Computation-Collective Operations. In Proceedings of the SC24: International Conference for High Performance Computing, Networking, Storage and Analysis, Atlanta, GA, USA, 17–22 November 2024; pp. 1–17. [CrossRef]
14. Wahlgren, J. Using GPU-Aware Message Passing to Accelerate High-Fidelity Fluid Simulations. 2022. Available online: https://www.diva-portal.org/smash/record.jsf?pid=diva2:1710487&dswid=7070 (accessed on 16 November 2022).
15. Khalilov, M.; Timofeev, A.; Polyakov, D. Towards OpenUCX and GPUDirect Technology Support for the Angara Interconnect. In *Supercomputing. RuSCDays 2022*; Lecture Notes in Computer Science; Voevodin, V., Sobolev, S., Yakobovskiy, M., Shagaliev, R., Eds.; Springer: Cham, Switzerland, 2022; Volume 13708, pp. 514–526. [CrossRef]
16. Jammer, T.; Bischof, C. Compiler-enabled optimization of persistent MPI Operations. In Proceedings of the 2022 IEEE/ACM International Workshop on Exascale MPI (ExaMPI), Dallas, TX, USA, 13–18 November 2022; pp. 1–10. [CrossRef]
17. Guo, J.; Yi, Q.; Meng, J.; Zhang, J.; Balaji, P. Compiler-Assisted Overlapping of Communication and Computation in MPI Applications. In Proceedings of the 2016 IEEE International Conference on Cluster Computing (CLUSTER), Taipei, Taiwan, 26–29 September 2016; pp. 60–69. [CrossRef]
18. Soga, T.; Yamaguchi, K.; Mathur, R.; Watanabe, O.; Musa, A.; Egawa, R.; Kobayashi, H. Effects of Using a Memory Stalled Core for Handling MPI Communication Overlapping in the SOR Solver on SX-ACE and SX-Aurora TSUBASA. *Supercomput. Front. Innov.* **2020**, *7*, 4–15. [CrossRef]
19. Klemm, M.; Cownie, J. *High Performance Parallel Runtimes: Design and Implementation*; Walter de Gruyter GmbH & Co KG: Berlin, Germany, 2021.

20. Katragadda, S. Optimizing High-Speed Data Transfers Using RDMA in Distributed Computing Environments. Available online: https://www.researchgate.net/profile/Santhosh-Katragadda/publication/388618653_OPTIMIZING_HIGH-SPEED_DATA_TRANSFERS_USING_RDMA_IN_DISTRIBUTED_COMPUTING_ENVIRONMENTS/links/679f16b6645ef274a45da115/OPTIMIZING-HIGH-SPEED-DATA-TRANSFERS-USING-RDMA-IN-DISTRIBUTED-COMPUTING-ENVIRONMENTS.pdf (accessed on 12 December 2022).
21. Luo, X. Optimization of MPI Collective Communication Operations. Ph.D. Dissertation, University of Tennessee, Knoxville, TN, USA, 2020. Available online: https://trace.tennessee.edu/utk_graddiss/5818 (accessed on 5 May 2020).
22. Wang, J.; Zhuang, Y.; Zeng, Y. A Transmission Optimization Method for MPI Communications. *J. Supercomput.* **2024**, *80*, 6240–6263. [CrossRef]
23. Shafie Khorassani, K.; Hashmi, J.; Chu, C.H.; Chen, C.C.; Subramoni, H.; Panda, D.K. Designing a ROCm-Aware MPI Library for AMD GPUs: Early Experiences. In *High Performance Computing. ISC High Performance 2021*; Lecture Notes in Computer Science; Chamberlain, B.L., Varbanescu, A.L., Ltaief, H., Luszczek, P., Eds.; Springer: Cham, Switzerland, 2021; Volume 12728. [CrossRef]
24. Ruhela, A.; Subramoni, H.; Chakraborty, S.; Bayatpour, M.; Kousha, P.; Panda, D.K. Efficient Design for MPI Asynchronous Progress without Dedicated Resources. *Parallel Comput.* **2019**, *85*, 13–26. [CrossRef]
25. Denis, A.; Jaeger, J.; Jeannot, E.; Reynier, F. One Core Dedicated to MPI Nonblocking Communication Progression? A Model to Assess Whether It Is Worth It. In Proceedings of the 2022 IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid), Taormina, Italy, 16–19 May 2022; pp. 736–746. [CrossRef]
26. Holmes, D.J.; Skjellum, A.; Schafer, D. Why Is MPI (Perceived to Be) So Complex? Part 1—Does Strong Progress Simplify MPI? In Proceedings of the 27th European MPI Users' Group Meeting, 21–24 September 2020; pp. 21–30. [CrossRef]
27. Bayatpour, M.; Sarkauskas, N.; Subramoni, H.; Hashmi, J.M.; Panda, D.K. BluesMPI: Efficient MPI Non-Blocking Alltoall Offloading Designs on Modern BlueField Smart NICs. In *High Performance Computing. ISC High Performance 2021*; Lecture Notes in Computer Science; Chamberlain, B.L., Varbanescu, A.L., Ltaief, H., Luszczek, P., Eds.; Springer: Cham, Switzerland, 2021; Volume 12728. [CrossRef]
28. Medvedev, A.V. IMB-ASYNC: A Revised Method and Benchmark to Estimate MPI-3 Asynchronous Progress Efficiency. *Clust. Comput.* **2022**, *25*, 2683–2697. [CrossRef]
29. Brightwell, R.; Underwood, K.D. An Analysis of the Impact of MPI Overlap and Independent Progress. In Proceedings of the 18th Annual International Conference on Supercomputing (ICS '04), Saint Malo, France, 26 June–1 July 2004; pp. 298–305. [CrossRef]
30. Holmes, D.J.; Skjellum, A.; Jaeger, J.; Grant, R.E.; Bangalore, P.V.; Dosanjh, M.G.; Bienz, A.; Schafer, D. Partitioned Collective Communication. In Proceedings of the 2021 Workshop on Exascale MPI (ExaMPI), St. Louis, MO, USA, 14 November 2021; pp. 9–17. [CrossRef]
31. Reynier, F. A Study on Progression of MPI Communications Using Dedicated Resources. Ph.D. Thesis, Université de Bordeaux, Pessac, France, 2022.
32. Dongarra, J.; Tourancheau, B.; Denis, A.; Jaeger, J.; Jeannot, E.; Pérache, M.; Taboada, H. Study on Progress Threads Placement and Dedicated Cores for Overlapping MPI Nonblocking Collectives on Manycore Processor. *Int. J. High Perform. Comput. Appl.* **2019**, *33*, 1240–1254. [CrossRef]
33. Bayatpour, M.; Ghazimirsaeed, S.M.; Xu, S.; Subramoni, H.; Panda, D.K. Design and Characterization of InfiniBand Hardware Tag Matching in MPI. In Proceedings of the 2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID), Melbourne, VIC, Australia, 11–14 May 2020; pp. 101–110. [CrossRef]
34. Levy, S.; Schonbein, W.; Ulmer, C. Leveraging High-Performance Data Transfer to Offload Data Management Tasks to SmartNICs. In Proceedings of the 2024 IEEE International Conference on Cluster Computing (CLUSTER), Kobe, Japan, 24–27 September 2024; pp. 346–356. [CrossRef]
35. Cardellini, V.; Fanfarillo, A.; Filippone, S. Overlap Communication with Computation in MPI Applications. 2016. Available online: https://art.torvergata.it/handle/2108/140530 (accessed on 1 February 2016).
36. Horikoshi, M.; Gerofi, B.; Ishikawa, Y.; Nakajima, K. Exploring Communication-Computation Overlap in Parallel Iterative Solvers on Manycore CPUs Using Asynchronous Progress Control. In Proceedings of the HPCAsia '22 Workshops, Virtual, 11–14 January 2022; pp. 29–39. [CrossRef]
37. Denis, A.; Jaeger, J.; Jeannot, E.; Reynier, F. A Methodology for Assessing Computation/Communication Overlap of MPI Nonblocking Collectives. *Concurr. Comput. Pract. Exp.* **2022**, *34*, e7168. [CrossRef]
38. Hoefler, T.; Lumsdaine, A. Message Progression in Parallel Computing—To Thread or Not to Thread? In Proceedings of the 2008 IEEE International Conference on Cluster Computing, Tsukuba, Japan, 29 September–1 October 2008; pp. 213–222. [CrossRef]
39. Nguyen, V.M.; Saillard, E.; Jaeger, J.; Barthou, D.; Carribault, P. Automatic Code Motion to Extend MPI Nonblocking Overlap Window. In *High Performance Computing. ISC High Performance 2020*; Lecture Notes in Computer Science; Jagode, H., Anzt, H., Juckeland, G., Ltaief, H., Eds.; Springer: Cham, Switzerland, 2020; Volume 12321. [CrossRef]

40. Nigay, A.; Mosimann, L.; Schneider, T.; Hoefler, T. Communication and Timing Issues with MPI Virtualization. In Proceedings of the 27th European MPI Users' Group Meeting (EuroMPI/USA '20). Association for Computing Machinery, New York, NY, USA, 11–20 September 2020. [CrossRef]
41. Zhou, H.; Latham, R.; Raffenetti, K.; Guo, Y.; Thakur, R. MPI Progress for All. In Proceedings of the SC24 Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis, Atlanta, GA, USA, 17–22 November 2024; pp. 425–435. [CrossRef]
42. Bayatpour, M.; Hashmi Maqbool, J.; Chakraborty, S.; Kandadi Suresh, K.; Ghazimirsaeed, S.M.; Ramesh, B.; Subramoni, H.; Panda, D.K. Communication-Aware Hardware-Assisted MPI Overlap Engine. In *High Performance Computing. ISC High Performance 2020*; Lecture Notes in Computer Science; Sadayappan, P., Chamberlain, B.L., Juckeland, G., Ltaief, H., Eds.; Springer: Cham, Switzerland, 2020; Volume 12151. [CrossRef]
43. Sarkauskas, N.; Bayatpour, M.; Tran, T.; Ramesh, B.; Subramoni, H.; Panda, D.K. Large-Message Nonblocking MPI_Iallgather and MPI_Ibcast Offload via BlueField-2 DPU. In Proceedings of the 2021 IEEE 28th International Conference on High Performance Computing, Data, and Analytics (HiPC), Bengaluru, India, 17–20 December 2021; pp. 388–393. [CrossRef]
44. Liang, C.; Dai, Y.; Xia, J.; Xu, J.; Peng, J.; Xu, W.; Xie, M.; Liu, J.; Lai, Z.; Ma, S.; et al. The Self-Adaptive and Topology-Aware MPI_Bcast Leveraging Collective Offload on Tianhe Express Interconnect. In Proceedings of the 2024 IEEE International Parallel and Distributed Processing Symposium (IPDPS), San Francisco, CA, USA, 27–31 May 2024; pp. 791–801. [CrossRef]
45. Castillo, E.; Jain, N.; Casas, M.; Moreto, M.; Schulz, M.; Beivide, R.; Valero, M.; Bhatele, A. Optimizing Computation-Communication Overlap in Asynchronous Task-Based Programs. In Proceedings of the ACM International Conference on Supercomputing (ICS '19), Phoenix, AZ, USA, 26–28 June 2019; pp. 380–391. [CrossRef]
46. Michalowicz, B.; Suresh, K.K.; Subramoni, H.; Abduljabbar, M.; Panda, D.K.; Poole, S. Effective and Efficient Offloading Designs for One-Sided Communication to SmartNICs. In Proceedings of the 2024 IEEE 31st International Conference on High Performance Computing, Data, and Analytics (HiPC), Bangalore, India, 18–21 December 2024; pp. 23–33. [CrossRef]
47. Dreier, N. Hardware-Oriented Krylov Methods for High-Performance Computing. Ph.D. Thesis. ProQuest Dissertations & Theses Global, 2021. Order No. 28946756. Available online: https://www.proquest.com/dissertations-theses/hardware-oriented-krylov-methods-high-performance/docview/2607316034/se-2 (accessed on 3 August 2021).
48. Thomadakis, P.; Tsolakis, C.; Chrisochoides, N. Multithreaded Runtime Framework for Parallel and Adaptive Applications. *Eng. Comput.* **2022**, *38*, 4675–4695. [CrossRef]
49. Guo, X.W.; Li, C.; Li, W.; Cao, Y.; Liu, Y.; Zhao, R.; Zhang, S.; Yang, C. Improving Performance for Simulating Complex Fluids on Massively Parallel Computers by Component Loop-Unrolling and Communication Hiding. In Proceedings of the 2020 IEEE 22nd International Conference on High Performance Computing and Communications; IEEE 18th International Conference on Smart City, Yanuca Island, Fiji, 14–16 December 2020; pp. 130–137. [CrossRef]
50. Liu, D.; Ren, X.; Wu, J.; Liu, W.; Zhao, J.; Peng, S. Pipe-AGCM: A Fine-Grain Pipelining Scheme for Optimizing the Parallel Atmospheric General Circulation Model. In *Euro-Par 2024: Parallel Processing. Euro-Par 2024*; Lecture Notes in Computer Science; Carretero, J., Shende, S., Garcia-Blas, J., Brandic, I., Olcoz, K., Schreiber, M., Eds.; Springer: Cham, Switzerland, 2024; Volume 14803. [CrossRef]
51. Tiwari, M. Communication Overlap Krylov Subspace Methods for Distributed Memory Systems. Ph.D. Thesis, Indian Institute of Science, Bangalore, India, 2022.
52. Xiong, H.; Li, K.; Liu, X.; Li, K. A Multi-Block Grids Load Balancing Algorithm with Improved Block Partitioning Strategy. In Proceedings of the 2019 IEEE 21st International Conference on High Performance Computing and Communications (HPCC), Zhangjiajie, China, 10–12 August 2019; pp. 37–44. [CrossRef]
53. Nakajima, K. Communication-Computation Overlapping for Parallel Multigrid Methods. In Proceedings of the 2024 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), San Francisco, CA, USA, 27–31 May 2024; pp. 751–760. [CrossRef]
54. Yepes-Arbós, X.; van den Oord, G.; Acosta, M.C.; Carver, G.D. Evaluation and Optimisation of the I/O Scalability for the Next Generation of Earth System Models: IFS CY43R3 and XIOS 2.0 Integration as a Case Study. *Geosci. Model Dev.* **2022**, *15*, 379–394. [CrossRef]
55. Castelló, A.; Quintana-Ortí, E.S.; Duato, J. Accelerating distributed deep neural network training with pipelined MPI allreduce. *Clust. Comput.* **2021**, *24*, 3797–3813. [CrossRef]
56. Jangda, A.; Huang, J.; Liu, G.; Sabet, A.H.N.; Maleki, S.; Miao, Y.; Musuvathi, M.; Mytkowicz, T.; Saarikivi, O. Breaking the computation and communication abstraction barrier in distributed machine learning workloads. In Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '22), Lausanne, Switzerland, 28 February–4 March 2022; Association for Computing Machinery: New York, NY, USA, 2022; pp. 402–416. [CrossRef]

57. Guo, X.; Pimentel, A.D.; Stefanov, T. Automated Exploration and Implementation of Distributed CNN Inference at the Edge. *IEEE Internet Things J.* **2023**, *10*, 5843–5858. [CrossRef]
58. Tran, T.; Kuncham, G.K.R.; Ramesh, B.; Xu, S.; Subramoni, H.; Abduljabbar, M.; Panda, D.K.D. OHIO: Improving RDMA Network Scalability in MPI_Alltoall Through Optimized Hierarchical and Intra/Inter-Node Communication Overlap Design. In Proceedings of the 2024 IEEE Symposium on High-Performance Interconnects (HOTI), Albuquerque, NM, USA, 21–23 August 2024; pp. 47–56. [CrossRef]

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.