Received 26 April 2024, accepted 1 June 2024, date of publication 11 June 2024, date of current version 19 June 2024.

Digital Object Identifier 10.1109/ACCESS.2024.3412878

RESEARCH ARTICLE

Inter-Node Message Passing Through Optical Reconfigurable Memory Channel

MAURICIO G. PALMA[1], JORGE GONZALEZ[2], MARTIN CARRASCO[2,3], RUTH RUBIO-NORIEGA[4], KEREN BERGMAN[5], (Fellow, IEEE), AND RODOLFO AZEVEDO[1], (Member, IEEE)

[1]Institute of Computing, UNICAMP, Campinas 13083-872, Brazil
[2]Departamento de Ciencia de la Computación, Universidad de Ingeniería y Tecnología, Barranco 15063, Peru
[3]Department of Computer Science, Vrije Universiteit Amsterdam, 1081 HV Amsterdam, The Netherlands
[4]Instituto Nacional de Investigación y Capacitación de Telecomunicaciones, Universidad Nacional de Ingeniería, Rímac 15333, Peru
[5]Department of Electrical Engineering, Columbia University, New York City, NY 10027, USA

Corresponding author: Mauricio G. Palma (mauricio.palma@ic.unicamp.br)

The work of Jorge Gonzalez was supported in part by Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES) under Grant 88882.329108/2019-01, in part by Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq) under Grant 142016/2020-9 and Grant 316067/2023-7, in part by the Sao Paulo Research Foundation (FAPESP) under Grant 2013/08293-7, in part by Consejo Nacional de Ciencia y Tecnología-Programa Nacional de Investigación Científica y Estudios avanzados (CONCYTEC-PROCIENCIA) under Contract 064-2021-FONDECYT, and in part by Universidad de Ingeniería y Tecnología (UTEC) Fondo Semilla 2023.

ABSTRACT Efficient data movement between nodes in a data center is essential for optimal performance of distributed workloads. With advancements in computing interconnection and memory, new opportunities have emerged. We propose a novel inter-node architecture and protocol called Flexible Memory Units (FMU) that uses optically disaggregated memory. FMUs can be dynamically allocated to different nodes during runtime using optical switches. The primary objective of FMUs is to use the disaggregated memory as temporary buffers during inter-node communication. We have implemented Simplecomm, an open-source simulator, to evaluate real MPI benchmarks using FMU. Our evaluation demonstrates significant speedups of up to 5.18× in communication-bound applications and 1.22× on computing-intensive applications, compared to a 100 Gbps InfiniBand interconnect.

INDEX TERMS Inter-node message passing, MPI, message passing, optical interconnection, photonic, disaggregated memory.

I. INTRODUCTION

Memory is a complex resource to manage in a data center. For example, users might not know exactly the memory requirement for a job, making memory overprovision a common practice [1], [2]. Furthermore, the memory space available to a process is limited to a single node's memory space, as the process cannot require memory space that resides on other nodes. Memory disaggregation [3] appears as a solution to flexible allocation of memory resources in the cluster. Optical interconnects are the leading candidate for the implementation of scalable memory disaggregation in data centers due to their inherent high bandwidth and low energy consumption [4], [5], [6], [7]. Some memory disaggregation works proposed the creation of a memory pool that is accessible by multiple nodes [8], [9], [10], [11], [12], [13]. Such nodes can request additional memory from this pool to increase their memory space and reduce overprovisioning.

Message Passing Interface (MPI) [14] is a standard programming model that simplifies data sharing between inter-node processes and increases software portability. Inter-node communication performance on a computer cluster directly impacts the execution of distributed programs because a program can have a relevant portion of its execution time spent on communication, and therefore several proposals aim to reduce this communication overhead. Previous works [15], [16], [17], [18], [19], [20], [21] considered the conventional architecture of a computer cluster where the Network Interface Card (NIC) is the only option

The associate editor coordinating the review of this manuscript and approving it for publication was Stanley Cheung.

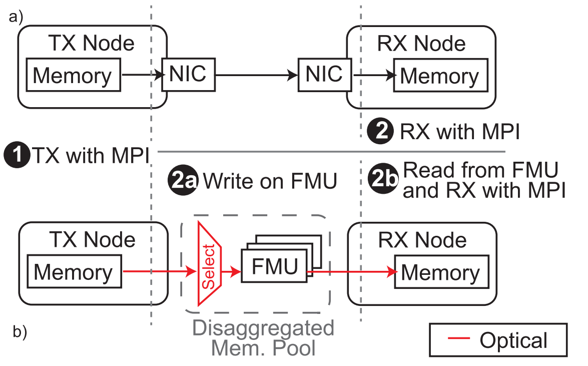


FIGURE 1. **Inter-node communication through FMU: TX_{node} writes data on the FMU, that is read afterwards by the RX_{node}.**

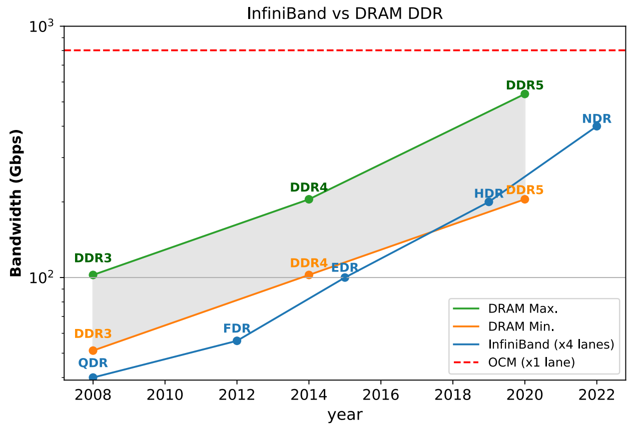


FIGURE 2. **Bandwidth comparison between DDR [23], [24], [25], Infiniband [26] and OCM [6].**

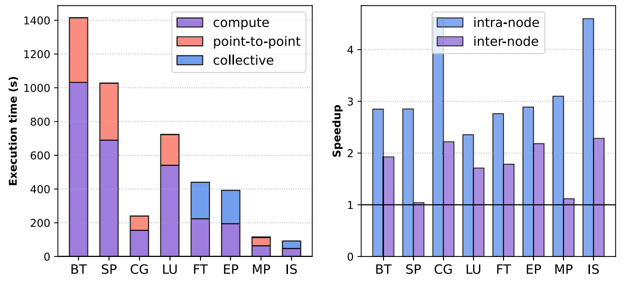


FIGURE 3. **NPB benchmark characterization with class D input. (Left) Exec. time. (Right) Speedup comparing intra-node and inter-node execution using 4 ranks and FDR Infiniband.**

to perform inter-node communication. The primary and foremost solution is to use RDMA [15], [16] while increasing bandwidth and reducing communication latency from network specifications, such as Ethernet and InfiniBand. Other MPI-related solutions include refining the MPI standard [17], improving the communication algorithms for better overlap of concurrent communications [18], developing specific accelerators for it [20], [22], or even relying on a hybrid approach (e.g., MPI+OpenMP [19]) aiming to improve the performance of intra-node communication and reduce the occurrences of inter-node communication.

Our goal in this work is to accelerate MPI inter-node communication by utilizing an optically connected memory pool as a shared memory space among nodes. We divide the memory pool into units, referred to as Flexible Memory Units (FMUs), and use FMUs as an intermediate buffer in the inter-node communication process. We named this novel inter-node communication scheme the FMU protocol. Figure 1 illustrates the operation of our novel method, which consists of two steps. First, the node that wants to send data (TX_{node}) writes them in the FMU. Notice that the FMU selection is done through an optically reconfigurable path (see Section III-C). Second, the target node (RX_{node}) reads it from the FMU, completing the inter-node communication process. During its design, we identified three main challenges in using the FMU protocol.

Challenge 1: For efficient memory access, access to FMU must have a performance similar to that of the DDR protocol. The main memory channel with the DDR interface performs better than the network protocols. Figure 2 compares DDR channels and InfiniBand, where the peak bandwidth performance from the DDR interface is at least twice the performance of an InfiniBand link. FMU accesses are made through Optically Connected Memory (OCM) [6], which is a custom optical point-to-point interconnect.

As depicted in Figure 2, OCM has two main characteristics: i) achieves Tbps per optical link (similar to DDR channels) outperforming commercial interconnects such as InfiniBand, and ii) enables interconnection in the order of meters, an order of magnitude higher (meters, rack-to-rack) than traditional memory channels (few centimeters, on board).

Challenge 2: We need to evaluate the FMU protocol on the execution of point-to-point communication and collective operations, as both are presented in MPI programs [27], [28]. Additionally, we needed to consider the computation and communication overlap on these kinds of workloads. Figure 3 (left) shows the total execution time of eight MPI applications from the NPB benchmark with the class D input. We observe that the applications spent 39% on average in communication. The highest communication times are 50% from `FT` and 50% from `EP`, both with collective routines. The lowest communication time is 25% from `LU` using point-to-point directives.

Challenge 3: The performance gap between the inter-node and intra-node execution caused by the communication overhead can increase by the optical devices overhead. Figure 3 (right) presents the speedup of six MPI applications from the NPB benchmark. Intra-node execution used four ranks in the same node, while inter-node used four ranks on four different nodes. The baseline is a single-rank execution on a single node. We observe that the inter-node communication causes a slowdown. In addition, we need to consider the intrinsic overhead of the optical devices (such as electro-optical conversion). This raises the fact that the FMU protocol requires low switching times (up to microseconds) and high port number capacity, allowing the nodes to connect to a higher number of FMUs.

Our novel optical FMU architecture aims to reduce the performance gap between the MPI intra- and inter-node communication. We focus on improving the inter-node scenario using reconfigurable optical interconnects. This work makes the following major contributions:

- We introduce the FMU architecture and its protocol. To our knowledge, this is the first work to use optically disaggregated memory as a buffer for inter-node message passing.
- Our FMU protocol allows the MPI runtime to select between conventional network interconnects (i.e., Infiniband) for small messages (in KB order) or our FMU interconnect for large messages (in MB order).
- We implemented the FMU timing and system model on our in-house simulator SimpleComm to study MPI systems with real workloads. To our knowledge, no open-source MPI simulator incorporates models of optically disaggregated memory to buffer MPI messages. Our simulator is available as open-source in [29].
- We extensively evaluated and compared our FMU architecture with current 100 Gbps Infiniband interconnects. FMU can achieve a speedup of 3× on messages larger than 8 MB. Communication-bound workloads show a speedup up to 5.18×; while workloads that present a relevant computation part show speedup up to 1.22×.

This paper is organized into nine main sections. Section II presents fundamental concepts on memory disaggregation, photonics, and MPI. Section III overviews the FMU architecture, while Section IV presents its timing model. On Section V we present details of our in-house simulator SimpleComm. On Section VI we present the experimental setup, and on Section VII we evaluate a cluster with our proposed architecture showing our results. Finally, in Section VIII we analyze related works, and on Section IX we discuss our final remarks.

II. BACKGROUND

This section briefly introduces three background topics for our work: memory disaggregation, optical switching, and MPI protocols. For further information on these topics, refer to [30], [31], and [32].

A. MEMORY DISAGGREGATION

Main memory disaggregation allows a computer node to obtain more memory space than initially assigned [3]. This is performed with two goals: 1) increasing the amount of memory that a node can use, as it can access memory modules that are placed over greater distances, and 2) improving the usage of the overall memory space of the cluster, claiming external memory as much as needed by its workload, reducing over-provisioning of memory space. Two distinct architectures achieve these goals [3]:

TABLE 1. Optical switch latency latency from state-of-the-art devices.

Work	Year	Switching Time
[49]	2016	17μs
[43]	2016	1μs
[50]	2017	20μs
[51]	2019	20μs
[52]	2020	5μs
[44]	2020	4ns
[46]	2021	4ns-10μs
[45]	2021	1μs
[48]	2023	100ps-10ms

1) SPLIT

A *split architecture* [8], [9], [13], [33], [34], [35], [36] allows a complete software solution to implement disaggregated memory using network interfaces (NIC). Memory disaggregation can be achieved using conventional electrical devices, e.g., PCIe interfaces with RDMA calls or custom protocols for resource disaggregation.

2) POOL

It aggregates memory in an external pool [8], [9], [11], which is connected to the computing nodes as an extra memory space. Recent work [6], [37], [38] shows photonics to be an appealing solution for attaching disaggregated memory directly to the memory channel. Integrated electronic and photonic circuits can provide optical interconnection in the memory channel. Furthermore, memory allocation becomes a solvable problem using reconfigurable optical circuit switches.

This **work** focuses on a *pool architecture*, where a memory pool is organized as a shared resource for the computer cluster (please refer to Section III-B).

B. OPTICAL SWITCHES FOR DATA CENTERS

Optical switch fabrics emerge as the basic block for next-generation interconnects in data centers. Optical switch fabrics have three main characteristics: i) exhibit better scalability compared to electronic switch fabrics while having a high port number [39], [40], ii) provide high bandwidth and low energy consumption [41], and iii) are compatible with CMOS while using photonic PDKs (Process Desing Kits) [42]. Nanosecond operation has already been demonstrated with photonic switches [43], [44], [45], [46], [47], [48]. However, reducing its system-level latencies remains a challenge. There are two main overhead latencies: i) the recovery and synchronization time whenever a lightpath is established because of the circuit switching nature of the optical links and ii) electro-optic devices driving and control related overheads to enable routing.

In general, the latencies of the optical switch for data communication are of the order of μs [43], [48], [49], [50], [51], [52], as shown in Table 1.

C. MESSAGE PASSING INTERFACE

The Message Passing Interface (MPI) standard [14] simplifies the development of parallel programs using distributed memory. An MPI implementation, such as MPICH [53] or OpenMPI [54], is required to code and execute MPI programs. These implementations have runtime processes that run alongside application processes, known as process manager (PM) [55]. PM provides inter-node communication schemes to the MPI programs. Each computing node has a running instance of the PM, which communicates to other nodes to coordinate and perform data transfers between processes.

Two key information required by the PM are: i) message source, provided by the process sending data, and ii) destination address where the data should be written, provided by the receiving process.

Additional information that needs to be equal is also sent from both processes, such as a signature and a tag. It enables the PM to compare the information from a sending (TX_{node}) to a receiving request (RX_{node}), a.k.a. tag matching [56], [57]. After performing the tag matching, the PM has all the information necessary to complete the message-passing process.

Inter-node communication with MPI can use two protocols [58]:

1) EAGER

Figure 4a presents the eager protocol. First, it uses previously allocated buffers 1b to allow TX_{node} to transfer the message 1a as soon as requested by the process. Afterward, RX_{node} performs the tag matching 2, copying the data to the process's local buffer.

2) RENDEZVOUS

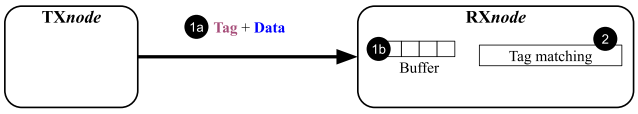
Figure 4b shows the rendezvous protocol. After RX_{node} receives the tag 1, it performs the tag matching 2a. Before transferring the message 3, TX_{node} receives a confirmation from the RX_{node} PM 2b.

The eager protocol has an advantage over the rendezvous because the message transfer between nodes starts before the tag matching and can conclude earlier, potentially reducing each process's time spent on communication. Despite this advantage, the eager protocol is limited to transferring messages that can fit the buffer size provided by the PM.

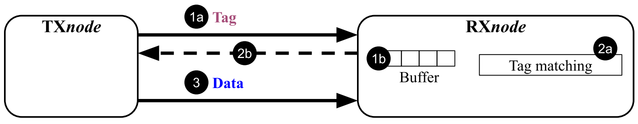
A defined message size threshold permits selecting the protocol. The eager protocol is used for messages smaller than the buffer, whereas the rendezvous protocol is chosen for larger messages. This motivates us to develop a new MPI protocol that allows for early transfers of large messages (see Section III-A), even before the tag matching.

III. FLEXIBLE MEMORY UNIT ARCHITECTURE

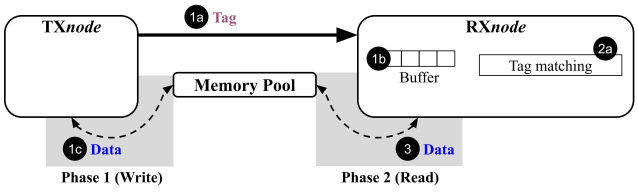
In this section, we present our novel inter-node reconfigurable architecture. The main goal is to use the memory space of a disaggregated memory pool as a dedicated buffer for storing MPI messages. We elaborate on the arbitration, mapping methods and timing model for reconfigurable memory channels.



(a) Eager protocol.



(b) Rendezvous protocol.



(c) FMU protocol.

FIGURE 4. Inter-node message passing protocols: Eager (a), Rendezvous (b) and FMU (c).

A. THE FMU PROTOCOL

FMU protocol consists of two sub-protocols: *read* and *write*. In the *write* protocol, TX_{node} transfers the message to the memory pool. In the *read* protocol, the memory pool transfers the message to the RX_{node}. Figure 4c shows that all actions performed by TX_{node} occur during writing, while all actions performed by RX_{node} occur during reading.

On the occurrence of a sending call, TX_{node} sends the tag for RX_{node} through the network 1a, similar to the rendezvous protocol. At the same time, TX_{node} sends the data to the memory pool 1c and ends its participation in the transfer. On a receiving call, RX_{node} requires tag matching 2a. Then RX_{node} can only initiate the reading request on the memory pool once TX_{node} has already issued the request to write the message on it. After the tag matching, RX_{node} begins reading the message from the memory pool 3, completing the inter-node communication process. Note that in conventional protocols and FMU, a local buffer 1b on RX_{node} is used to store the tag.

B. ARCHITECTURE OVERVIEW

Figure 5 presents an overview of our FMU architecture within a cluster. It has a set of N computing nodes. Each node (left) has a local memory, a Network Interface Card (NIC), and an FMU controller (CFMU) 1. It can access the FMU memory pool (right) to use it as an additional memory

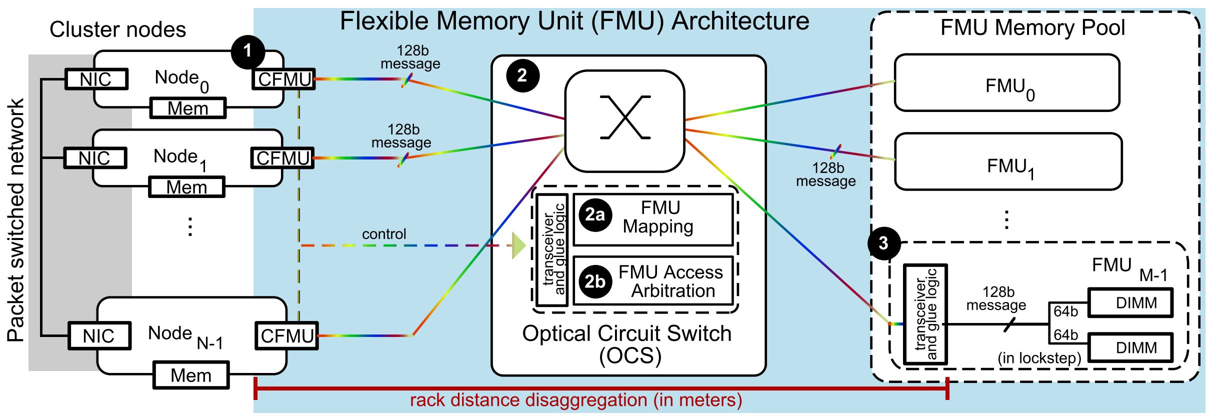


FIGURE 5. Overview of our computer cluster, where we have the nodes connected to a memory pool through an optical interconnection, which is managed using an Optical Circuit Switch (OCS).

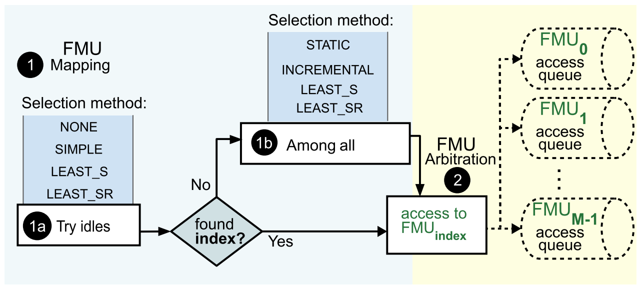


FIGURE 6. FMU access arbitration provided by the OCS, composed of the FMU mapping schemes and the accessing queues.

space. FMU relies on the memory disaggregation solution proposed in [6], which uses a silicon photonic link to establish memory disaggregation at the rack distance (on the order of meters). This optical interconnect has endpoints to translate the information from the optical domain to the electrical domain and vice versa. Over this optical interconnect, we add an optical circuit switch (OCS) 2 to enable reallocation in the memory pool. In addition to switching, OCS also performs a mapping 2a and arbitration 2b mechanism (Section III-C). Mapping consists in choosing which FMU will be used on the protocol, while arbitration controls the access to the FMUs. The memory pool 3 is a set of M FMU single units. A single FMU consists of an array of DIMMs in lock-step operation. Figure 5 (right) illustrates an array of two DIMMs that use a 64b interface each, resulting in a channel of 128b. There is extra hardware for the transceiver and its glue logic to modulate (transmit) and demodulate (receive) data, placed inside the CFMU, OCS, and FMU single units.

C. FMU ACCESS ARBITRATION AND MAPPING

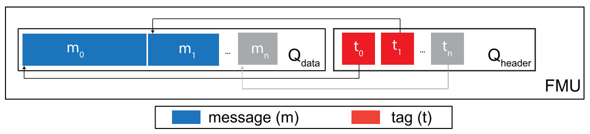
Figure 5 2 depicts an optical circuit switch (OCS) which has two main functions: 1) enables reconfiguration and 2) acts as a mapper and arbiter to access the FMUs. Each FMU can be connected only to a single CFMU, and vice versa. OCS arbitration works using a FIFO queue per FMU. The CFMU interacts with the OCS using two commands: 1) $FMU_{acquire}$ to request access to an FMU and 2) $FMU_{release}$ to release the FMU previously allocated. If the OCS controller receives an $FMU_{acquire}$, it pushes the request into the queue of the requested FMU. Each time a $FMU_{release}$ is issued, the OCS pops the queue of the FMU allocated to the CFMU.

OCS controller chooses one FMU based on our mapping policies if $FMU_{acquire}$ does not specify one. Figure 6 shows the OCS controller's steps on an incoming $FMU_{acquire}$, detailing the mapping scheme and its interaction with the FMU queues. As shown in Figure 5, the memory pool is a set of M FMUs. FMU mapping process (1) selects an index i from the set. It has two consecutive selection methods: 1) try idle and 2) among all. First, the OCS controller (1a) evaluates a mapping scheme based only on idle FMUs in which the accessing queue is empty and no CFMU is connected. Second, if no index is found because there is no empty queue (1b), then the index is selected from the entire set of M FMUs.

The node that sends the data selects the FMU during inter-node communication. To choose the FMU, the sending node (TX_{node}) issues an $FMU_{acquire}$ to obtain an FMU index. On the other hand, when the receiving node (RX_{node}) issues an $FMU_{acquire}$, it returns the previous index value.

TRY IDLE: this FMU index selection method has five algorithms.

- *NONE:* returns no index i and goes to **AMONG ALL**.
- *RANDOM:* selects and returns a random index i.
- *SIMPLE:* chooses the lowest index i;
- *LEAST_S:* chooses the index of the least used FMU accounting previous send requests. The memory pool contains M FMUs represented by an array of M, where each position holds a counter of the amount of data written into an FMU (in bytes). Each position i in the array represents an FMU, and the size of the message increments the counter. Then, the OCS controller selects the FMU with the lowest counter. If two or more positions have the same value, the controller selects the one with the lowest index.



```
Write(t,m)
1.FMUacquire
2.Write tag $t_n$ on $Q_{header}$
3.Write message $m_n$ on $Q_{data}$
4.FMUrelease
```

```
Read(m,t)
1.FMUacquire
2.Tag matching $t_n$ on $Q_{header}$
3.Read message $m_n$ on $Q_{data}$
4.Invalidate tag $t_n$ on $Q_{header}$
5.Remove invalid entries from
  FMU content, then FMUrelease
```

FIGURE 7. FMU content (top), composed of two data structures: Q_{header} and Q_{data}, and the protocols steps (bottom) followed by the nodes for interacting with these structures.

- *LEAST_SR:* chooses the index of the least used FMU accounting previous send and receive requests. Similarly to LEAST_S, the message size increments the FMU counter with send requests. However, it decreases with the receiving requests.

AMONG ALL: This selection method has five algorithms including *RANDOM*, *LEAST_S* and *LEAST_SR*, which are the same as in the **TRY IDLE** selection method.

- **STATIC**: chooses based on the index of RX_{node}, using $FMU_{index} = r \pmod M$, where r is the index of RX_{node} and M is the total number of FMUs;
- **INCREMENTAL**: chooses based on a wrapping global counter, which initiates at zero and increments on each send request. We use the formula $FMU_{index} = gi \pmod M$, where gi is the global counter and M is the total number of FMUs;

We evaluated our mapping methods in Section VI-D to observe its impact on protocol contention and system performance.

D. FMU READ AND WRITE PROTOCOLS

Figure 7 (top) presents the content of an FMU, and it consists of two circular queues: (1) Q_{data}, which is used to store the content of a message (m), and (2) Q_{header}, which is used as an index structure to store the tag and a pointer (t) to the address of the message content stored on Q_{data}. Both queues are similar to the circular queue of pending messages used for buffered sends in the MPI standard, e.g., when using *MPI_Buffer_attach*.

The FMU address space is a set of contiguous addresses limited by physical capacity. This memory space limits the maximum size of both Q_{header} and Q_{data}. Both queues have the same number of messages and tags, namely $m_{0,1,\ldots,n}$ and $t_{0,1,\ldots,n}$, respectively. However, while t_i entries in Q_{header} have the same size, m_i in Q_{data} can be of varying sizes. We define the size of Q_{data} and Q_{header} experimentally under different workloads; please refer to our results in Section VII-B.

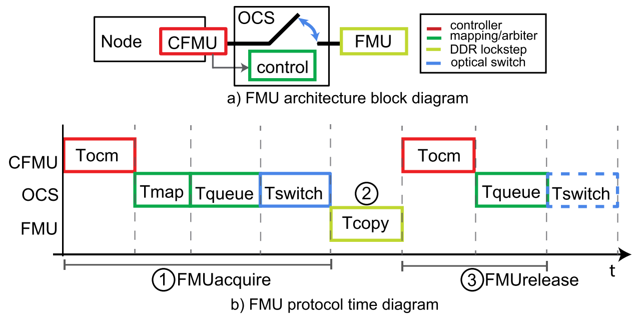


FIGURE 8. Timing diagram showing the interaction between CFMU, OCS and FMU.

Figure 7 (bottom) depicts the FMU protocol and its two main parts: i) read protocol for the RX_{node}, and ii) write protocol for the TX_{node}. It shows that the main steps to read or write a message or tag on an FMU are performed between $FMU_{acquire}$ and $FMU_{release}$. FMU protocol supports coalescing for read and write operations, allowing multiple message transfers simultaneously when using non-blocking operations.

IV. TIMING MODEL FOR THE FMU PROTOCOL

In this section, we defined a timing model to identify latencies that can impact the overall FMU performance. The model is based on two main definitions: the access time, T_{access} (from TX_{node} to FMU) and the transfer time, $T_{transfer}$ (from TX_{node} to FMU and to RX_{node}).

A. FMU ACCESS TIME, T_{ACCESS}

The FMU access process starts with an $FMU_{acquire}$ command issued by the node, and ends with an $FMU_{release}$. The process is depicted in Figure 8. For better understanding, we divide it into three steps:

Step 1: The node sends an $FMU_{acquire}$ to the OCS. The OCS takes a total time of T_{ocm} to acknowledge this request. T_{ocm} represents the delay due to the disaggregating solution. Every interaction between CFMU and the OCS, or between CFMU and an FMU delays a system by T_{ocm}. Thus, the first step has a total delay time of $T_{acquire} = 2T_{ocm} + T_{map} + T_{switch}$, where T_{map} is the time the OCS takes to select an FMU (by executing an FMU mapping scheme) and to place the request in the FMU queue, T_{queue} is the time the request stays in the queue, and T_{switch} is the switching time. After the switch has been completed, OCS informs the node about it. The node acknowledges this switch after T_{ocm} seconds.

Step 2: Here the local memory of the node and the selected FMU exchange data. Each interaction has a disaggregation delay of T_{ocm} and employs T_{copy} to complete. We define $T_{copy} = \frac{S}{BW}$, where S is the data exchanged in bytes and BW is the real bandwidth of the memory channel (in bytes per second).

Step 3: After data exchange is completed, the node issues an $FMU_{release}$ to the OCS, removes the request from the front of the queue, and terminates the interaction. The OCS takes

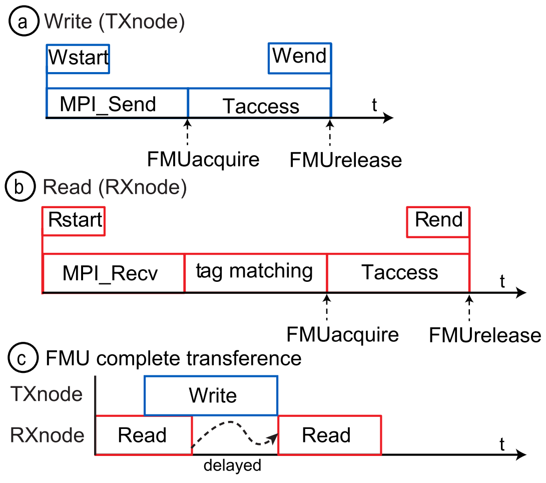


FIGURE 9. FMU transfer time from the TX_{node}, RX_{node} and during the complete transfer process.

T_{ocm} to acknowledge $FMU_{release}$ and terminates the access process.

The FMU access process has a total access time T_{access} defined by Equation 1. We can simplify this equation by neglecting some latencies. Namely, T_{ocm} and T_{map} are on the order of a few nanoseconds, with calculated T_{ocm} as low as 3.2 ns [6], and T_{map} depends only on a fast non-iterative algorithm (please refer to Section III-C). Additionally, our experiments let us consider T_{switch} in the order of a few microseconds, and T_{queue} a variable determined by the number of requests on the accessing queue. Thus, T_{access} can be reduced to Equation 2. Note that the realized bandwidth BW, the message size, S and T_{switch} are parameters that can be obtained by analyzing the memory channel, the MPI operation, and the OCS. Furthermore, T_{queue} represents the contention during execution time of the FMU protocol.

$$\begin{aligned} T_{access} &= 4 \times T_{ocm} + T_{map} + T_{queue} \\ &\quad + T_{switch} + T_{copy} \end{aligned} \tag{1}$$

$$\begin{aligned} T_{access} &\approx T_{queue} + T_{switch} + T_{copy} \\ &\approx T_{queue} + T_{switch} + \frac{S}{BW} \end{aligned} \tag{2}$$

B. FMU TRANSFER TIME, $T_{TRANSFER}$

$T_{transfer}$ is the total transfer time during the FMU protocol execution. We calculate three cases of $T_{transfer}$ according to Figure 9: (a) the transmission node, TX_{node}, (b) the reception node, RX_{node}, and (c) the whole transference process.

The write protocol (a) starts with an *MPI_Send*, starting an $FMU_{acquire}$ process that completes after T_{access} seconds. Thus, in TX_{node}, $T_{transfer} = T_{access}$ (Equation 3). Note that this behavior is similar to the eager protocol.

$$T_{transfer\ (TX_{node})} = T_{access} \tag{3}$$

The read protocol (b) starts with a receiving call, *MPI_Recv*, which issues an $FMU_{acquire}$. $T_{transfer}$ in the reading phase can take two values. If the reading phase starts after the end of a writing phase, RX_{node} issues $FMU_{acquire}$ immediately, so $T_{transfer} = T_{access}$. However, if the read protocol starts before the end of a write protocol, there is an additional delay difference between the end of the write phase, W_{end}, and the start of the receiving phase, R_{start}, thus $T_{transfer} = (W_{end} - R_{start}) + T_{access}$. Note that RX_{node} is able to issue $FMU_{acquire}$ before W_{end}, however, the request will remain in the accessing queue at least until the end of the write protocol (Figure 9c).

$$T_{transfer}(RX_{node}) = \begin{cases} T_{access}, & R_{start} \geq W_{end} \\ (W_{end} - R_{start}) + T_{access}, & R_{start} < W_{end} \end{cases} \tag{4}$$

Lastly, **the complete transfer process** (c) begins when the sending call is issued, which starts the writing phase. The transfer is completed when the message is located in local memory RX_{node}, which occurs at the end of the reading phase, after R_{end}. Hence, the delay in the whole transfer process can be calculated by $T_{transfer} = R_{end} - W_{start}$.

The three definitions of $T_{transfer}$ provide correct timing but use variables whose values are only known during execution. To create an analytical model for the transmission time of a message of size S, we considered the minimum time that $T_{transfer}$ can take. Equation 5 is a simplified model that uses only interconnection characteristics and message size. It highlights that the message is transmitted twice in the FMU protocol, once from TX_{node} to FMU and again from FMU to RX_{node}.

$$\begin{aligned} T_{transfer(simplified)} &= T_{access(TX_{node})} + T_{access(RX_{node})} \\ &= T_{switch} + \frac{S}{BW} + T_{switch} + \frac{S}{BW} \\ &= 2 \times (T_{switch} + \frac{S}{BW}) \end{aligned} \tag{5}$$

C. FMU COMPARED TO PACKET-SWITCHED NETWORKS

High-performance computing systems mainly use packet-switched networks with interconnects like InfiniBand and Ethernet. Equation 6 shows a simplified communication timing model, $\hat{T}_{transfer}$, for a packet-switched network, where T_L is the minimum time to transmit a single package, S is the message size and BW is the maximum bandwidth achieved by the interconnect.

$$\hat{T}_{transfer} = T_{packet-switched} = T_L + \frac{S}{BW} \tag{6}$$

FMU and packet-switched networks have three key differences, which are outlined in Table 2. In packet-switched networks, the bottleneck is the NIC to NIC performance, while in FMU, it is the DDR interface of the memory channel. T_{queue} is affected by the contention that occurs before a node obtains access to the FMU. Additionally, packet-switched networks share the communication channel among all nodes, whereas FMU offers an exclusive channel. In terms of available protocols, packet-switched networks use eager

TABLE 2. Summary of characteristics from packet-switched networks and FMU for inter-node message passing.

	Network	FMU
BW bottleneck	NIC	Memory Channel
Contention	Packet Switching	Circuit Switching
Protocols	Eager or Rendezvous	Read and Write

or rendezvous protocols, while FMU uses a two-protocol scheme that benefits larger messages similar to the eager protocol.

Equation 6 models the packet-switched network transfer, while Equation 5 models the FMU protocol transfer time. Although both models have components representing latency and bandwidth, the FMU protocol requires transmitting the message twice. Therefore, the memory channel should provide a bandwidth that is at least twice as high as that of the packet-switched network to maintain throughput. Note that in the FMU protocol, each phase of the transmission can utilize the full bandwidth of the memory channel, as shown in Equation 5.

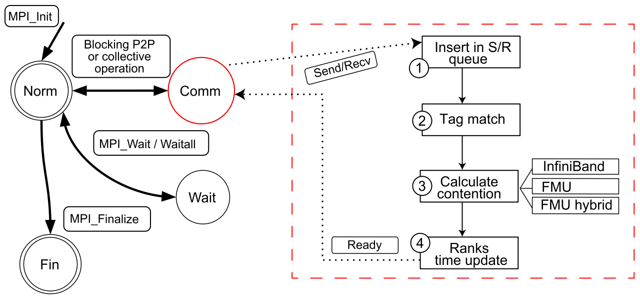
$$T_L + \frac{\hat{S}}{BW} = 2 \times T_{switch} + 2 \times \frac{\hat{S}}{BW} \qquad (7)$$

The FMU and the packet-switched network protocols can be used to transfer messages based on their size, S. Comparing the two transfer timing models helps to decide which method provides the lowest transfer time. The FMU protocol is associated with the higher bandwidth provided by the memory channel and the memory space provided by the FMU, so it is more suited for transferring large messages. However, smaller messages may have to rely on the packet-switched network. The pivot value, denoted by $\hat{S}$, represents the threshold value at which the transfer time on both approaches is equal, as given by Equation 7. This hybrid approach enables automatic use of the FMU protocol, without prior knowledge of distributed workloads.

V. FMU SIMULATOR

We developed SimpleComm, a trace-oriented MPI simulator that calculates the communication contention and the overlap of communication and computation. It works using time-independent traces produced by SimGrid [59], which are composed of MPI calls and approximations of the computing time expressed in floating-point operations per second (Flops). Currently, SimpleComm executes real MPI workloads using three models: i) FMU, ii) hybrid FMU (see Section VI-D), and iii) Infiniband.

We modeled InfiniBand communication as an ideal remote direct memory access (RDMA) transfer with no protocol overhead from protocol, package processing, or buffer contention. We model the message or payload transfer time in the simulator. We use state-of-the-art switching latency (see Section II-B) and a maximum bandwidth from two lockstep



(a) SimpleComm rank state machine (b) Communication state flow diagram

FIGURE 10. SimpleComm implementation overview.

DDR5 4800 MHz devices [25] in both FMU and hybrid FMU models. All FMU accesses are managed by a request queue manager while the total time accounts for the queueing and transfer time.

SimpleComm simulates each rank individually, starting at *MPI_Init*, using a four-state machine to control its progress:

- ***Norm:*** This is the initial state where any MPI operation does not block the rank. Computation progress and non-blocking MPI operations keep the rank in this state.
- ***Comm:*** The rank is blocked by a P2P or collective MPI operation.
- ***Wait:*** The rank is blocked by an *MPI_Wait* or *MPI_Waitall* operation.
- ***Fin:*** The rank finished its simulation with *MPI_Finalize*.

Comm and *Wait* are blocking states where computation estimation holds until the communication operation ends. The time spent in the blocking states counts as *idleness*.

SimpleComm's rank communication time estimation is presented in Figure 10b. An MPI operation can generate both send and receive (S/R) calls, which are included in the S/R queue (①). For each inclusion, SimpleComm performs a tag matching. If successful, it takes the issue time of the S/R call (②) and adds the transfer time calculated using Equation 5 for the FMU protocol and Equation 6 for packet-switched networks. Subsequently, SimpleComm accounts for contention between the transfers (③), adjusting the transfer times accordingly.

While the contention model for InfiniBand is based on a packet switch topology, the contention model for FMU is based on a circuit switch. The main difference compared to Infiniband is that FMU can only send data between nodes once a communication path is set by the first request in the queue. The output of this contention analysis is the final time annotation related to the transfer time. Finally, this value returns to the rank (④). Ranks return to the *Norm* state after the required time information is updated. In P2P operations, one other rank is involved in communication (two updates), and in collective operations, there are multiple ranks involved (more than two updates).

SimpleComm is written in Python and was designed to simplify the implementation of new contention models. A contention model is an abstract class that can be used

```python
from abc import ABC, abstractmethod
#Every new communication method should implement
#its own version of class Topology
class Topology(ABC):
    ...
    @abstractmethod
    def processContention(self, matchQ) -> MQ_Match:
        #Implement the contention model
        #in this method
        pass;
```

CODS LISTING 1. Definition of topology and its contention method `processContention`

```
...
[TOPOLOGY]
topology = FMU //select between FMU, InfiniBand and Hybrid
number_of_fmus = 4
fmu_seek_idle_kind = NONE // TRY IDLE
fmu_mapping = INCREMENTAL // AMONG ALL
fmu_bandwidth = 76800000000 // in Bps
fmu_latency = 0.000005 // in Seconds
processing_speed = 12000000000 // in Flops
...
```

CODS LISTING 2. Example configuration file for FMU

as a template. Listing 1 presents a template code snipet showing the signature of the *processContention* method. A new communication model is defined in the *Topology* class and has its definition of the *processContention* method.

SimpleComm supports several parameters in a configuration file. For example, the FMU model has the following parameters: latency, bandwidth, number of FMUs, CPU processing speed, and mapping methods. Listing 2 presents a code snippet of the configuration file, setting the simulator to the FMU protocol, with 4 FMUs, using the FMU mapping methods NONE (try idle) and INCREMENTAL (among all), bandwidth to 76.8 GBs, 5μs latency, and a node processing speed of 12 GFlops. SimpleComm is available as open-source in [29].

VI. EVALUATION METHODOLOGY

A. EXPERIMENTAL SETUP

We evaluated our proposal using the aforementioned simulator. Table 3 presents our simulation parameters. Our baseline system comprises up to 64 nodes, each with a CPU with a frequency of 3 GHz and a peak processing speed of 12 GFlops. These nodes connect through an InfiniBand network, and each node has a single CFMU that enables it to access the external pool of memories composed of up to 64 FMUs. The MPI benchmarks use up to 64 ranks, allocating one rank per node in all cases. The InfiniBand communication cost is estimated using its theoretical bandwidth [26] and an experimental latency measured with the *OSU_Latency* application from [60], running on a local system with Mellanox ConnectX-5 NICs. FMU communication is circuit switching, and its cost is estimated to be the maximum bandwidth of two lockstep DDR5-4800 MHz devices. We considered a 5 μs switching time based on [6].

B. BENCHMARKS

We evaluated our methods using 3 synthetic and 4 real-world benchmarks:

TABLE 3. Simulated system and interconnect parameters for Infiniband and FMU models.

Cluster	
CPU Frequency	3 GHz 12 GFlops (Freq. x AVX-256)
Max. number of Nodes	64
Max. number of FMUs	64
Max. number of MPI ranks	64 (1 rank per node)
Interconnect parameters	
InfiniBand	
Latency (*OSU_Latency*)	8 μs
Bandwidth	12.5 GB/s (Mellanox Connect X-5 NIC)
Communication cost	$8\ \mu s + \frac{size}{12.5GB/s}$ (from Eq. 6)
Eager Threshold	65536 Bytes
FMU	
Latency (T_{switch})	5 μs
Bandwidth	76.8 GB/s (DDR5-4800 MHz)
Communication cost	$10\ \mu s + \frac{size}{38.4GB/s}$ (from Eq. 5)

- **Synthetic benchmarks,** we used three synthetic benchmarks that aim to stress a specific MPI collective operation: i) *Ring_Bcast* for *MPI_Bcast*, ii) *Ring_Reduce* for *MPI_Reduce* and iii) *Ring_Allreduce* for *MPI_Allreduce*. They provide a scenario where several messages of the same size are transmitted simultaneously. These benchmarks are communication-bound, as their computation parts are negligible. Their iterative execution repeats 100 hundred times. A benchmark calls the collective operation once per rank each time. For example, executing them with 64 ranks results in the collective operation being called 64 times per iteration, varying the root rank on each call. The benchmarks *Ring_Allreduce* and *MPI_Allreduce* do not have a root rank. In our experiments (see Section VII-A), *MPI_Bcast* and *MPI_Reduce* were tuned to use the MPI binomial tree algorithm [61], [62], and *MPI_Allreduce* is composed of an *MPI_Reduce* followed by an *MPI_Bcast*.
- **NPB benchmarks,** we selected four benchmarks from the NAS Parallel Benchmark suite (NPB) [63]: `CG`,`FT`, `IS`, and `LU`. We used two input workloads, class C and class D. Class D provides a heavier workload than class C in computation and communication.

C. MESSAGE SIZE ANALYSIS AND HYBRID FMU VARIANT

We used the communication cost from Table 3 to estimate the impact on performance. We executed the ping-pong type program *OSU_latency* [60] and measured the transfer time between two nodes using Infiniband and FMU without contention. Although FMU is not intended to be the main interconnect between nodes, this evaluation helped us validate our simulator behavior and identify the message size where FMU can outperform Infiniband during the execution of an ideal communication. Figure 11 presents the transfer time of messages up to 10 MB (large messages). With large messages, the transfer time increases for both Infiniband and FMU. FMU obtains a maximum speedup of 2.99× for

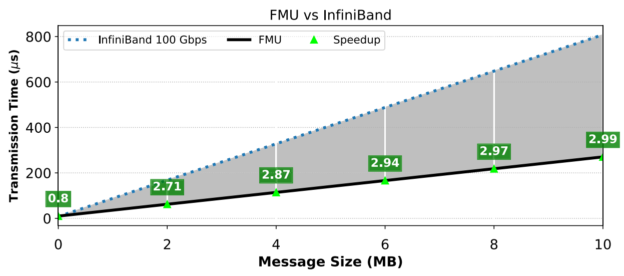


FIGURE 11. Comparison of transmission time between Infiniband and FMU with large messages in the order of MB (lower is better).

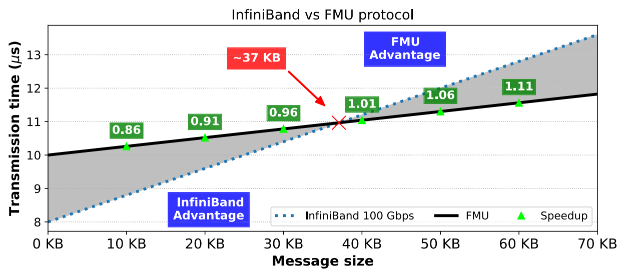


FIGURE 12. Comparison of transmission time between Infiniband and FMU with short messages in the order of KB (lower is better).

messages of 10 MB over Infiniband. This result matches the expected bandwidth ratio $\frac{BW_{FMU}}{BW_{InfiniBand}}$ between FMU and Infiniband.

Figure 12 presents the transfer time of messages smaller than 70 KB. The pivot value of 37 KB represents the minimum message size where the Infiniband communication cost is higher than the FMU communication cost.

In our experiments, we evaluated the FMU protocol by executing the benchmarks in two scenarios: i) using the FMU protocol and ii) using a variant called hybrid FMU where messages smaller than 37 KB are transferred using the Infiniband interconnect.

D. MAPPING METHOD ANALYSIS

We measured speedup with FMU using all mapping methods (see Section III-C) using the NPB benchmarks, comparing the results to a conventional Infiniband interconnect. We observe that the combination of NONE (try idle) and INCREMENTAL (among all) mapping methods obtained the highest speedup when using 16, 32, and 64 FMUs. The combination of LEAST_S (try idle) and STATIC (among all) showed the highest speedup when using 1, 4, and 8 FMUs. However, the speedup difference with less than 8 FMUs is below 3%. The maximum speedup of FMU with NONE and INCREMENTAL is 1.22×. For this reason, we selected the NONE and INCREMENTAL mapping methods to show our detailed performance evaluation.

VII. FMU EVALUATION

A. PERFORMANCE EVALUATION WITH SYNTHETIC BENCHMARKS

We evaluated a scenario in which synthetic benchmarks generate messages of identical sizes that are simultaneously transmitted. These benchmarks do not require computation on the nodes. Table 4 shows the total number of messages per synthetic benchmark and its message size. The average message size during execution is equal to the input size of the benchmark. As expected, the nodes remain idle during all executions (100% idleness) due to the benchmark implementation only containing the MPI collective for communication.

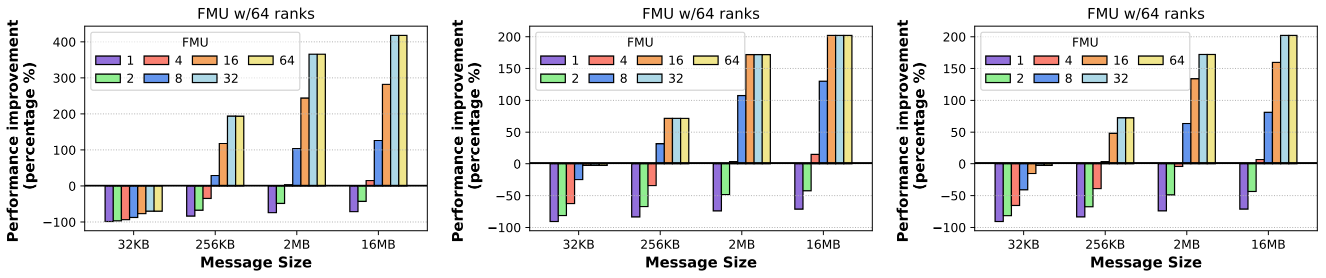
Figure 13 presents the speedup of the FMU protocol over InfiniBand when running the synthetic benchmarks with 64 ranks, one rank per node, with up to 64 FMUs. We evaluated message sizes from 32 KB to 16 MB. The results with 16 MB messages and 64 FMUs show a maximum speedup of 5.18× with *Ring_Bcast*, and an average speedup of 3.02× with the Reduce and Allreduce benchmarks. We make three key observations. First, increasing the number of FMUs to eight or more with messages larger than 2MB can lead to improved performance. In an ideal evaluation scenario without contention, a pivot value of 37 KB message size is observed (see Section VI-C). We observe a similar behavior in a contention scenario, with 32 KB messages exhibiting slowdown. FMU architecture with messages larger than 2 MB can help improve performance in contention scenarios.

Second, the ratio between the number of ranks and FMUs ($\frac{\#Ranks}{\#FMUs}$) need to be greater than 4 to obtain a similar performance to the baseline. However, with 32 FMUs, the performance improvement is similar with 64 FMUs. This can be caused by the waiting time to first access an FMU. The average waiting time with 16 FMUs is 6 ns, with 32 FMUs is 1.9 ns, and with 64 FMUs it is on the order of ps.

Third, the binomial tree broadcast algorithm provides negligible performance impact when used with the FMU protocol due to its circuit-switching operation. Using a naive algorithm, we tested a small benchmark using *MPI_Reduction* where all ranks send the message straight to the root rank. The results with the naive and the binomial tree algorithm were the same, while Infiniband shows a speedup

TABLE 4. Message characteristics of the synthetic benchmarks aimed to stress MPI collective operations.

Benchmark	Total messages	Msg. dize	MPI idleness source
Ring_Bcast	403K	32KB	Bcast
		256KB	
		2MB	
		16MB	
Ring_Reduce	403K	32KB	Reduce
		256KB	
		2MB	
		16MB	
Ring_Allreduce	806K	32KB	Allreduce
		256KB	
		2MB	
		16MB	



(a) Bcast (b) Reduce (c) Allreduce

FIGURE 13. Performance improvement of FMU over InfiniBand when executing synthetic benchmarks and 64 ranks (higher is better).

TABLE 5. NPB benchmarks: 4 benchmarks from NPB using classes C and D inputs.

Name	Input	Ranks	Total messages	Av. Msg. size (B)	Messages >37 KB (%)	MPI Idleness Source	Aggr. Idle. (%)
LU	C	4	323 K	9 K	1	P2P	1.99
		16	1.9 M	4 K	1	P2P	8.59
		64	9 M	2 K	1	P2P	19.01
	D	4	980 K	24 K	1	P2P	0.69
		16	5.8 M	12 K	1	P2P	2.45
		64	27 M	6 K	1	P2P	5.53
CG	C	4	31 K	296 K	50	P2P	10.94
		16	223 K	127 K	43	P2P	28.91
		64	1.2 M	59 K	40	P2P	55.79
	D	4	42 K	2.9 M	50	P2P	3.61
		16	297 K	1.2 M	43	P2P	10.07
		64	1.7 M	593 K	40	P2P	23.31
FT	C	4	351	100 M	76	Alltoall	21.19
		16	5 K	7.7 M	93	Alltoall	26.74
		64	90 K	513 K	98	Alltoall	29.21
	D	4	426	1.6 G	76	Alltoall	17.72
		16	6 K	124 M	93	Alltoall	20.86
		64	111 K	8.2 M	98	Alltoall	22.09
IS	C	4	348	12 M	38	Alltoallv	17.21
		16	5 K	971 K	47	Alltoallv	23.01
		64	90 K	64 K	49	Alltoallv	30.86
	D	4	348	203 M	38	Alltoallv	16.42
		16	5 K	15 M	47	Alltoallv	25.13
		64	90 K	1 M	49	Bcast	41.68

of up to 3× with a binomial tree compared to the naive algorithm.

B. PERFORMANCE EVALUATION WITH NPB BENCHMARKS

Table 5 shows the results using 4, 16, and 64 ranks with NPB benchmarks using input classes C and D. We measured three values: i) the total number of messages transmitted, ii) average message size, and iii) idleness as the percentage of time from the total execution time the ranks were idle due to communication.

We observe that idleness can increase with the number of ranks on all-to-all MPI communication patterns. As shown in Table 5, idleness increases with `FT` and `IS` benchmarks, while it reduces with `CG` and remains the same with `LU` being both P2P MPI communication. As expected, a higher number of ranks allows more messages to be on-fly in the system. The `FT` benchmark with class D input shows an average message size of 1.6 GB with 4 ranks and 8.2 MB with 64 ranks.

As shown in Figure 14, we evaluated the performance of the FMU protocol compared to InfiniBand by executing the NPB workloads using FMU and hybrid FMU (see Section VI-C).

We considered 64 FMUs using 4, 16, and 64 ranks, allocating one rank per node. Table 5 shows that only 1% of the messages produced with `LU` and class C input are larger than 37 KB. The maximum speed-up in both scenarios is 22%, that is, execute `FT` with class C input, 16 ranks, and 16 FMUs. We make two key observations. First, the hybrid FMU approach reduces the slowdown when transferring messages of smaller sizes (37KB). As depicted in Figure 14c, the hybrid FMU approach with `CG` benchmark and class C input shows improvement from 11% to 16% using 16 ranks and from 5% to 20% with 64 ranks. Second, a system with fewer FMUs can benefit from the hybrid approach if the workload can transfer more than one message when accessing an FMU.

Assuming DDR5 DIMMs of 8 GB, an FMU provides a minimum of 16 GB of storage space. We tracked the number of messages and their size during our experiments. Both define the size of Q_{data} and Q_{header}. The largest size of Q_{data} is 8.5 GB with the `FT` benchmark using class D inputs. We measured a maximum of 3938 messages in a single FMU with the `IS` benchmark, in both the class C and class D inputs. The estimated size of Q_{header} is $\approx$ 32 KB considering 8B blocks for indexing.

C. INTRA-NODE COMMUNICATION

FMU interconnects can achieve the bandwidth of local DDR memory because optical interconnects have a high bandwidth. We evaluated a scenario where we placed all ranks in the same node, making all communication intra-node. The FMU bandwidth is 76.8 GB/s (from FMU in Table 3), and the switch latency is zero because it does not require path reconfiguration. We executed four NPB benchmarks with class C input and the *Ring_Bcast* benchmark using a small (32 KB) and large (16 MB) message size. Figure 16a shows the speedup results of inter-node communication with InfiniBand and FMU compared to a conventional DDR interconnect using 4 ranks. The performance in NPB benchmarks is similar to the baseline in

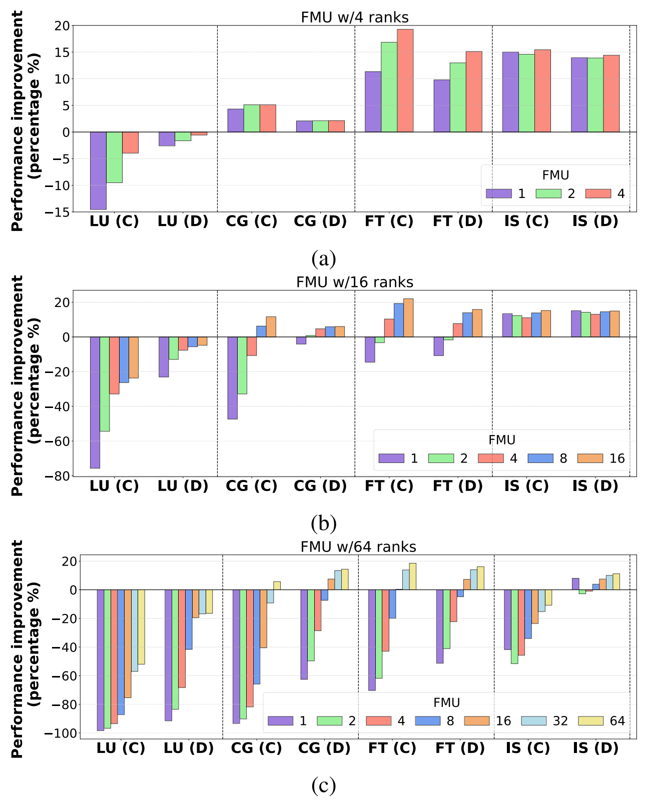


FIGURE 14. FMU performance improvement compared to InfiniBand with NPB benchmarks (higher is better).

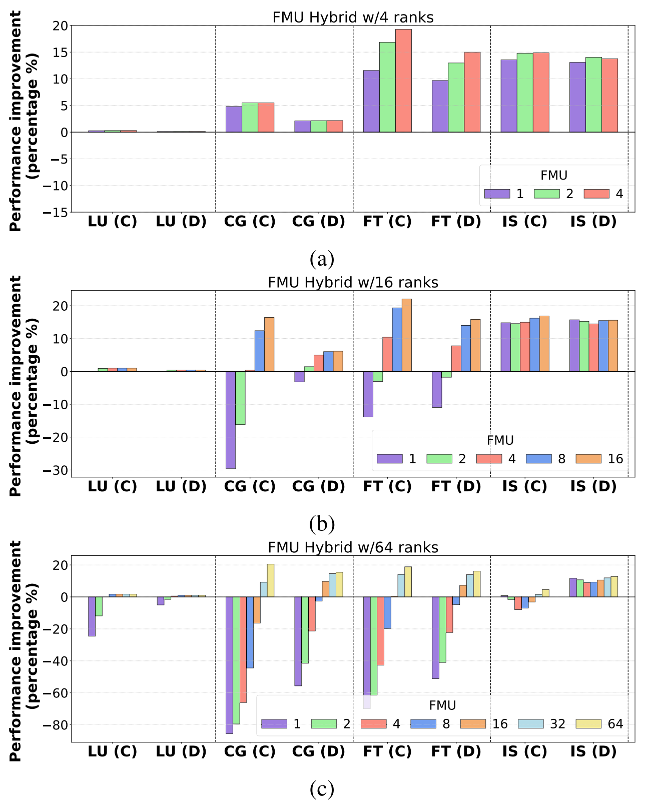


FIGURE 15. FMU hybrid performance improvement compared to InfiniBand with NPB benchmarks (higher is better).

Infiniband and FMU. FMU shows an average 0.99× speedup, and Infiniband shows an average speedup of 0.93×. FMU and Infiniband presented slowdown with Ring benchmark using 32 KB. With 16 MB messages, Infiniband slowed down while FMU presented a similar performance to the baseline. Performance improves as we increase the number of ranks to 64. As depicted in Figure 16b, on NPB benchmarks both Infiniband and FMU presented speedups with CG, FT and IS benchmarks. With *Ring_Bcast* using 16 MB messages, FMU achieved a speedup of 8.8×. This significant improvement compared to the baseline can be caused by the contention reduction provided by FMU. Multiple memory requests on the baseline highly degrade the performance of intra-node communication. We make two observations about this experiment: i) intra-node communication exhibits a better performance on scenarios of low contention and small message sizes, such as LU and Ring with 32 KB; ii) FMU shows an average speedup of 2.81× and Infiniband an average speedup of 1.81× on benchmarks that uses large message sizes, i.e. CG, FT, IS and Ring (16 MB).

VIII. RELATED WORK

To our knowledge, this is the first work to implement and extensively evaluate an MPI protocol with optically disaggregated memory for inter-node message buffering. This section presents related works on MPI inter-node optimization, memory disaggregation, and optical disaggregation.

A. MPI INTER-NODE OPTIMIZATION

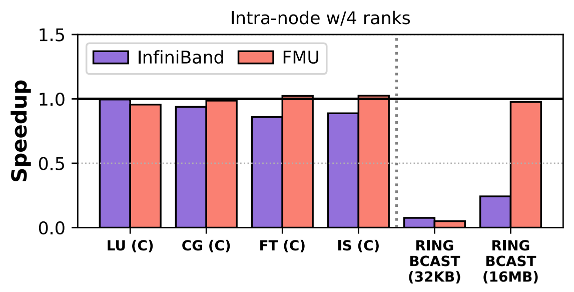
In [17] the authors characterize the MPI standard to understand the performance challenges. Previous works on MPI inter-node communication aimed to use RDMA [15], [16]. Previous works focus on improving collective operations with specific algorithms [18] and developing hardware accelerators [20], [22]. Other works study the usage of threads on endpoints [19], [21]; tag matching offloading [56], [57]. None of the previous work evaluates optical disaggregated memory as an alternative to overcome MPI performance challenges.

B. MEMORY DISAGGREGATION

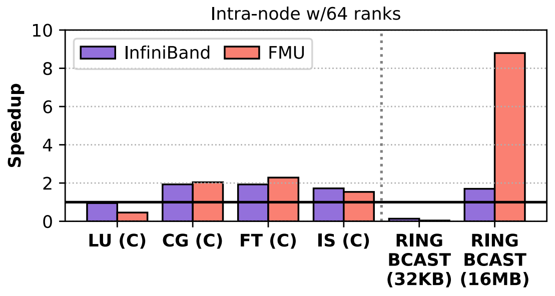
There are several works on disaggregated memory pool [8], [9], [13], [35], [36]. Many prior works used the NIC-to-NIC interface to share node local memory [33], [34] or implement an additional level in the memory hierarchy [11], [12]. However, none of these works focused on MPI-disaggregated memory usage for point-to-point and collective operations. In [3], the authors discussed the need for more work on message passing optimization in the disaggregated memory scenario.

C. PHOTONICS FOR DATACENTER

Recent works explore reconfigurable optical links inside the data center for efficient data movement [51], [64], [65], [66]. Other works propose new devices and systems for



(a) 4 ranks with NPB (left) and Ring (right) benchmarks.



(b) 64 ranks with NPB (left) and Ring (right) benchmarks.

FIGURE 16. Speedup of Infiniband and FMU compared to intra-node communication.

optical disaggregation [67], [68]. In [6] the authors show a detailed implementation of the optical interconnect for memory disaggregation.

IX. CONCLUSION

We have explored a novel approach for a disaggregated memory architecture and system-level design through optical reconfigurable channels. A comprehensive analysis combining protocol definition, timing models, and system-level simulation was performed to evaluate our memory architecture. Our FMU architecture is designed to work as an additional disaggregated buffer in cooperation with conventional interconnects such as Infiniband. Our evaluation shows performance gains of up to 5.18× in communication-bound applications and up to 1.22× in computing-intensive applications. Our new protocol is also implemented on a custom simulator based on SimGrid traces. The combined results in this work provide a detailed analysis of MPI inter-node communication. Our results characterize scenarios associated with small (37 KB) and large (> 2 MB) message sizes where MPI collectives can take advantage of our FMU architecture. This study empowers the usage of optical interconnects for disaggregated memory, paving the way for the future implementation of experimental systems.

REFERENCES

[1] J. Guo, Z. Chang, S. Wang, H. Ding, Y. Feng, L. Mao, and Y. Bao, "Who limits the resource efficiency of my datacenter: An analysis of Alibaba datacenter traces," in *Proc. IEEE/ACM 27th Int. Symp. Quality Service (IWQoS)*, Jun. 2019, pp. 1–10.

[2] I. Peng, R. Pearce, and M. Gokhale, "On the memory underutilization: Exploring disaggregated memory on HPC systems," in *Proc. IEEE 32nd Int. Symp. Comput. Archit. High Perform. Comput. (SBAC-PAD)*, Sep. 2020, pp. 183–190.

[3] M. Ewais and P. Chow, "Disaggregated memory in the datacenter: A survey," *IEEE Access*, vol. 11, pp. 20688–20712, 2023.

[4] G. Michelogiannakis, B. Klenk, B. Cook, M. Y. Teh, M. Glick, L. Dennison, K. Bergman, and J. Shalf, "A case for intra-rack resource disaggregation in HPC," *ACM Trans. Archit. Code Optim.*, vol. 19, no. 2, pp. 1–26, Mar. 2022.

[5] Z. Zhu, G. Di Guglielmo, Q. Cheng, M. Glick, J. Kwon, H. Guan, L. P. Carloni, and K. Bergman, "Photonic switched optically connected memory: An approach to address memory challenges in deep learning," *J. Lightw. Technol.*, vol. 38, no. 10, pp. 2815–2825, May 15, 2020.

[6] J. Gonzalez, M. G. Palma, M. Hattink, R. Rubio-Noriega, L. Orosa, O. Mutlu, K. Bergman, and R. Azevedo, "Optically connected memory for disaggregated data centers," *J. Parallel Distrib. Comput.*, vol. 163, pp. 300–312, May 2022.

[7] R. Lin, Y. Cheng, M. D. Andrade, L. Wosinska, and J. Chen, "Disaggregated data centers: Challenges and trade-offs," *IEEE Commun. Mag.*, vol. 58, no. 2, pp. 20–26, Feb. 2020.

[8] K. Lim, J. Chang, T. Mudge, P. Ranganathan, S. K. Reinhardt, and T. F. Wenisch, "Disaggregated memory for expansion and sharing in blade servers," *ACM SIGARCH Comput. Archit. News*, vol. 37, no. 3, pp. 267–278, Jun. 2009.

[9] K. Lim, Y. Turner, J. R. Santos, A. AuYoung, J. Chang, P. Ranganathan, and T. F. Wenisch, "System-level implications of disaggregated memory," in *Proc. IEEE Int. Symp. High-Performance Comp Architecture*, Feb. 2012, pp. 1–12.

[10] P. Faraboschi, K. Keeton, T. Marsland, and D. Milojicic, "Beyond processor-centric operating systems," in *Proc. 15th Workshop Hot Topics Operating Syst.*, 2015, pp. 1–7.

[11] S. Bergman, P. Faldu, B. Grot, L. Vilanova, and M. Silberstein, "Reconsidering OS memory optimizations in the presence of disaggregated memory," in *Proc. ACM SIGPLAN Int. Symp. Memory Manag.*, Jun. 2022, pp. 1–14.

[12] C. Giannoula, K. Huang, J. Tang, N. Koziris, G. Goumas, Z. Chishti, and N. Vijaykumar, "DaeMon: Architectural support for efficient data movement in fully disaggregated systems," *Proc. ACM Meas. Anal. Comput. Syst.*, vol. 7, no. 1, pp. 1–36, 2023.

[13] H. Li, D. S. Berger, L. Hsu, D. Ernst, P. Zardoshti, S. Novakovic, M. Shah, S. Rajadnya, S. Lee, I. Agarwal, M. D. Hill, M. Fontoura, and R. Bianchini, "Pond: CXL-based memory pooling systems for cloud platforms," in *Proc. 28th ACM Int. Conf. Architectural Support for Program. Lang. Operating Syst.*, Jan. 2023, pp. 574–587.

[14] Message Passing Interface Forum, *MPI: A Message-Passing Interface Standard Version 4.0*, Jun. 2021. [Online]. Available: https://www.mpi-forum.org/

[15] J. Liu, J. Wu, S. P. Kini, P. Wyckoff, and D. K. Panda, "High performance RDMA-based MPI implementation over InfiniBand," in *Proc. 17th Annu. Int. Conf. Supercomputing*, Jun. 2003, pp. 295–304.

[16] N. Hjelm, M. G. F. Dosanjh, R. E. Grant, T. Groves, P. Bridges, and D. Arnold, "Improving MPI multi-threaded RMA communication performance," in *Proc. 47th Int. Conf. Parallel Process.*, Aug. 2018, pp. 1–11.

[17] K. Raffenetti, A. Amer, L. Oden, C. Archer, W. Bland, H. Fujita, Y. Guo, T. Janjusic, D. Durnov, M. Blocksome, and M. Si, "Why is MPI so slow? Analyzing the fundamental limits in implementing MPI-3.1," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, Nov. 2017, pp. 1–12.

[18] E. Chan, M. Heimlich, A. Purkayastha, and R. van de Geijn, "Collective communication: Theory, practice, and experience," *Concurrency Comput., Pract. Exper.*, vol. 19, no. 13, pp. 1749–1783, Sep. 2007.

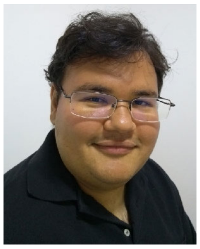
[19] D. E. Bernholdt, S. Boehm, G. Bosilca, M. Gorentla Venkata, R. E. Grant, T. Naughton, H. P. Pritchard, M. Schulz, and G. R. Vallee, "A survey of MPI usage in the US exascale computing project," *Concurrency Comput., Pract. Exper.*, vol. 32, no. 3, Feb. 2020.

[20] P. Haghi, A. Guo, Q. Xiong, R. Patel, C. Yang, T. Geng, J. T. Broaddus, R. Marshall, A. Skjellum, and M. C. Herbordt, "FPGAs in the network and novel communicator support accelerate MPI collectives," in *Proc. IEEE High Perform. Extreme Comput. Conf. (HPEC)*, Sep. 2020, pp. 1–10.
[21] R. Zambre, A. Chandramowliswharan, and P. Balaji, "How I learned to stop worrying about user-visible endpoints and love MPI," in *Proc. 34th ACM Int. Conf. Supercomputing*, Jun. 2020, pp. 1–13.
[22] C. Heinz and A. Koch, "Near-data FPGA-accelerated processing of collective and inference operations in disaggregated memory systems," in *Proc. IEEE/ACM Int. Workshop Heterogeneous High-Performance Reconfigurable Comput. (HRC)*, Nov. 2021, pp. 44–51.
[23] JEDEC. *DDR3 SDRAM*, Standard JESD79-3F, Jul. 2012.
[24] JEDEC. *DDR4 SDRAM*, Standard JESD79-4D, Jul. 2021.
[25] JEDEC. *DDR5 SDRAM*, Standard JESD79-5B, Aug. 2022.
[26] InfiniBand Trade Association, *InfiniBand Architecture Specification*, Release 1.4, Beaverton, OR, USA, Apr. 2020. [Online]. Available: https://www.infinibandta.org/
[27] E. Nuriyev, J.-A. Rico-Gallego, and A. Lastovetsky, "Model-based selection of optimal MPI broadcast algorithms for multi-core clusters," *J. Parallel Distrib. Comput.*, vol. 165, pp. 1–16, Jul. 2022.
[28] J.-Y. Cho, P.-R. Seo, and H.-W. Jin, "Exploiting copy engines for intra-node MPI collective communication," *J. Supercomput.*, vol. 79, no. 16, pp. 17962–17982, Nov. 2023.
[29] M. G. Palma. (2024). *Simplecomm Source Code*. [Online]. Available: https://github.com/Dragonslair5/SimpleComm
[30] M. Y. Teh, Z. Wu, M. Glick, S. Rumley, M. Ghobadi, and K. Bergman, "Performance trade-offs in reconfigurable networks for HPC," *J. Opt. Commun. Netw.*, vol. 14, no. 6, pp. 454–468, Jun. 2022.
[31] G. Michelogiannakis, Y. Arafa, B. Cook, L. Yuan Dai, A. Hameed Badawy, M. Glick, Y. Wang, K. Bergman, and J. Shalf, "Efficient intra-rack resource disaggregation for HPC using co-packaged DWDM photonics," 2023, *arXiv:2301.03592*.
[32] A. Ottino, J. Benjamin, and G. Zervas, "RAMP: A flat nanosecond optical network and MPI operations for distributed deep learning systems," *Opt. Switching Netw.*, vol. 51, Feb. 2024, Art. no. 100761.
[33] J. Gu, Y. Lee, Y. Zhang, M. Chowdhury, and K. G. Shin, "Efficient memory disaggregation with infiniswap," in *Proc. 14th USENIX Symp. Netw. Syst. Design Implement. (NSDI)*, 2017, pp. 649–667.
[34] E. Amaro, C. Branner-Augmon, Z. Luo, A. Ousterhout, M. K. Aguilera, A. Panda, S. Ratnasamy, and S. Shenker, "Can far memory improve job throughput?" in *Proc. 15th Eur. Conf. Comput. Syst.*, Apr. 2020, pp. 1–16.
[35] Compute Express Link Consortium, *Compute Express Link (CXL) Specification, 1.0 Edition*, Rev. 3.0, Aug. 2022. [Online]. Available: https://computeexpresslink.org/
[36] D. Gouk, S. Lee, M. Kwon, and M. Jung, "Direct access, high-performance memory disaggregation with directcxl," in *Proc. USENIX Annu. Tech. Conf.*, 2022, pp. 287–294.
[37] G. Zervas, H. Yuan, A. Saljoghei, Q. Chen, and V. Mishra, "Optically disaggregated data centers with minimal remote memory latency: Technologies, architectures, and resource allocation [invited]," *J. Opt. Commun. Netw.*, vol. 10, no. 2, pp. 270–285, Feb. 2018.
[38] M. Eppenberger, M. Bonomi, D. Moor, M. Mueller, B. I. Bitachon, T. Burger, and L. Alloatti, "Compact optical TX and RX macros for computercom monolithically integrated in 45 nm CMOS," *J. Lightw. Technol.*, vol. 39, no. 21, pp. 6869–6879, Nov. 1, 2021.
[39] Y. Lu and H. Gu, "Flexible and scalable optical interconnects for data centers: Trends and challenges," *IEEE Commun. Mag.*, vol. 57, no. 10, pp. 27–33, Oct. 2019.
[40] K. Suzuki, R. Konoike, H. Matsuura, R. Matsumoto, T. Inoue, S. Namiki, H. Kawashima, and K. Ikeda, "Recent advances in large-scale optical switches based on silicon photonics," in *Proc. Opt. Fiber Commun. Conf. Exhib. (OFC)*, Mar. 2022, pp. 1–3.
[41] J. Xia, M. Chen, A. Rizzo, B. Sun, Z. Wang, M. Glick, Q. Cheng, K. Bergman, and R. Penty, "O-band microring resonator based switch-and-select silicon photonic switch fabric," in *Proc. Conf. Lasers Electro-Optics (CLEO)*, May 2022, pp. 1–2.
[42] S. Y. Siew, B. Li, F. Gao, H. Y. Zheng, W. Zhang, P. Guo, S. W. Xie, A. Song, B. Dong, L. W. Luo, C. Li, X. Luo, and G.-Q. Lo, "Review of silicon photonics technology and platform development," *J. Lightw. Technol.*, vol. 39, no. 13, pp. 4374–4389, Jul. 1, 2021.
[43] T. J. Seok, N. Quack, S. Han, R. S. Müller, and M. C. Wu, "Large-scale broadband digital silicon photonic switches with vertical adiabatic couplers," *Optica*, vol. 3, no. 1, pp. 64–70, 2016.
[44] H. Ballani, P. Costa, R. Behrendt, D. Cletheroe, I. Haller, K. Jozwik, F. Karinou, S. Lange, K. Shi, B. Thomsen, and H. Williams, "Sirius: A flat datacenter network with nanosecond optical switching," in *Proc. Annu. Conf. ACM Special Interest Group Data Commun. Appl., Technol., Architectures, Protocols Comput. Commun.*, Jul. 2020, pp. 782–797.
[45] A. S. Raja, S. Lange, M. Karpov, K. Shi, X. Fu, R. Behrendt, D. Cletheroe, A. Lukashchuk, I. Haller, F. Karinou, B. Thomsen, K. Jozwik, J. Liu, P. Costa, T. J. Kippenberg, and H. Ballani, "Ultrafast optical circuit switching for data centers using integrated soliton microcombs," *Nature Commun.*, vol. 12, no. 1, p. 5867, Oct. 2021.
[46] M. Nance Hall, K.-T. Foerster, S. Schmid, and R. Durairajan, "A survey of reconfigurable optical networks," *Opt. Switching Netw.*, vol. 41, Sep. 2021, Art. no. 100621.
[47] X. Xue and N. Calabretta, "Nanosecond optical switching and control system for data center networks," *Nature Commun.*, vol. 13, no. 1, p. 2257, Apr. 2022.
[48] X. Xue, S. Zhang, B. Guo, W. Ji, R. Yin, B. Chen, and S. Huang, "Optical switching data center networks: Understanding techniques and challenges," 2023, *arXiv:2302.05298*.
[49] W. M. Mellette, G. M. Schuster, G. Porter, G. Papen, and J. E. Ford, "A scalable, partially configurable optical switch for data center networks," *J. Lightw. Technol.*, vol. 35, no. 2, pp. 136–144, Jan. 15, 2017.
[50] W. M. Mellette, R. McGuinness, A. Roy, A. Forencich, G. Papen, A. C. Snoeren, and G. Porter, "RotorNet: A scalable, low-complexity, optical datacenter network," in *Proc. Conf. ACM Special Interest Group Data Commun.*, Aug. 2017, pp. 267–280.
[51] G. Michelogiannakis, Y. Shen, M. Y. Teh, X. Meng, B. Aivazi, T. Groves, J. Shalf, M. Glick, M. Ghobadi, L. Dennison, and K. Bergman, "Bandwidth steering in HPC using silicon nanophotonics," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, Nov. 2019, pp. 1–25.
[52] J. L. Benjamin, T. Gerard, D. Lavery, P. Bayvel, and G. Zervas, "PULSE: Optical circuit switched data center architecture operating at nanosecond timescales," *J. Lightw. Technol.*, vol. 38, no. 18, pp. 4906–4921, Sep. 15, 2020.
[53] (2023). *MPICH*. [Online]. Available: https://www.mpich.org/
[54] OpenMPI. (2023). *Open MPI: Open Source High Performance Computing*. [Online]. Available: https://www.open-mpi.org/
[55] MPICH Repository. (2023). *MPICH Source Code*. [Online]. Available: https://github.com/pmodels/mpich/blob/main/doc/wiki/notes/pmi.md
[56] W. P. Marts, M. G. F. Dosanjh, W. Schonbein, R. E. Grant, and P. G. Bridges, "MPI tag matching performance on ConnectX and ARM," in *Proc. 26th Eur. MPI Users' Group Meeting*, Sep. 2019, pp. 1–10.
[57] M. Bayatpour, S. M. Ghazimirsaeed, S. Xu, H. Subramoni, and D. K. Panda, "Design and characterization of InfiniBand hardware tag matching in MPI," in *Proc. 20th IEEE/ACM Int. Symp. Cluster, Cloud Internet Comput. (CCGRID)*, May 2020, pp. 101–110.
[58] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: Portable Parallel Programming With the Message-Passing Interface*. Cambridge, MA, USA: MIT Press, 2014.
[59] H. Casanova, A. Giersch, A. Legrand, M. Quinson, and F. Suter, "Versatile, scalable, and accurate simulation of distributed applications and platforms," *J. Parallel Distrib. Comput.*, vol. 74, no. 10, pp. 2899–2917, Oct. 2014.
[60] *Ohio State University Benchmark Suite*. Accessed: Jul. 18, 2022. [Online]. Available: https://mvapich.cse.ohio-state.edu/benchmarks/
[61] E. Nuriyev and A. Lastovetsky, "Efficient and accurate selection of optimal collective communication algorithms using analytical performance modeling," *IEEE Access*, vol. 9, pp. 109355–109373, 2021.
[62] P. Patarasuk and X. Yuan, "Efficient MPI bcast across different process arrival patterns," in *Proc. IEEE Int. Symp. Parallel Distrib. Process.*, Apr. 2008, pp. 1–11.
[63] D. Bailey, T. Harris, W. Saphir, R. Van Der Wijngaart, A. Woo, and M. Yarrow, "The NAS parallel benchmarks 2.0," NASA, Washington, DC, USA, Ames Res. Center, Tech. Rep., NAS-95-020, 1995.
[64] M. Y. Teh, Z. Wu, and K. Bergman, "Flexspander: Augmenting expander networks in high-performance systems with optical bandwidth steering," *J. Opt. Commun. Netw.*, vol. 12, no. 4, pp. 44–54, Apr. 2020.
[65] A. Ottino, J. Benjamin, and G. Zervas, "RAMP: A flat nanosecond optical network and MPI operations for distributed deep learning systems," 2022, *arXiv:2211.15226*.

[66] Z. Wu, L. Y. Dai, A. Novick, M. Glick, Z. Zhu, S. Rumley, G. Michelogiannakis, J. Shalf, and K. Bergman, "Peta-scale embedded photonics architecture for distributed deep learning applications," *J. Lightw. Technol.*, vol. 41, no. 12, pp. 3737–3749, Jun. 15, 2023.

[67] X. Guo, X. Xue, F. Yan, B. Pan, G. Exarchakos, and N. Calabretta, "DACON: A reconfigurable application-centric optical network for disaggregated data center infrastructures [Invited]," *J. Opt. Commun. Netw.*, vol. 14, no. 1, pp. 69–80, Jan. 2022.

[68] K. Ishii, R. Matsumoto, T. Inoue, and S. Namiki, "Disaggregated optical-layer switching for optically composable disaggregated computing [invited]," *J. Opt. Commun. Netw.*, vol. 15, no. 1, pp. 11–25, Jan. 2023.



MAURICIO G. PALMA is currently pursuing the Ph.D. degree with the Computing Systems Laboratory, University of Campinas, São Paulo, Brazil. His current research interests include computer architecture, optical interconnects, memory systems, and parallel processing.

JORGE GONZALEZ received the Ph.D. degree from the University of Campinas, São Paulo, Brazil. He is currently an Assistant Professor with the Computer Science Department, University of Technology and Engineering, Peru. His current research interests include computer architecture, optical interconnects, memory systems, and intra-chip traffic.

MARTIN CARRASCO received the B.S. degree from the University of Engineering and Technology, Peru. He is currently pursuing the M.S. degree in artificial intelligence with Vrije Universiteit Amsterdam. His research interests include computer architecture, deep learning, and computational linguistics.

RUTH RUBIO-NORIEGA received the Ph.D. degree from the University of Campinas, São Paulo, Brazil. She is currently an Assistant Professor with the National University of Engineering, Peru, and a Researcher with the National Institute for Research and Training in Telecommunications, Lima, Peru. Her research interests include photonic devices, electrooptics, optical interconnects, and applied electromagnetics.

KEREN BERGMAN (Fellow, IEEE) received the B.S. degree in electrical engineering from Bucknell University, Lewisburg, PA, USA, in 1988, and the M.S. and Ph.D. degrees in electrical engineering from the Massachusetts Institute of Technology, Cambridge, MA, USA, in 1991 and 1994, respectively. She is currently a Charles Batchelor Professor with Columbia University, New York City, NY, USA, where she also directs the Lightwave Research Laboratory. She leads multiple research programs on optical interconnection networks for advanced computing systems, data centers, optical packet-switched routers, and chip multiprocessor nanophotonic networks-on-chip. She is a fellow of the Optica.

RODOLFO AZEVEDO (Member, IEEE) received the Ph.D. degree in computer science from the University of Campinas (UNICAMP) in 2002. He is currently a Full Professor with UNICAMP. He is a member of the Computer Science Graduate Program, where he advises master's and Ph.D. students. He was the Director of the Institute of Computing, from 2017 to 2019, and was the President of the São Paulo Virtual University (UNIVESP), from 2019 to 2022. He received four best papers in conferences (SBAC-PAD 2004, SBAC-PAD 2008, 2018, and WSCAD-SSC 2012). In 2012, he received the Zeferino Vaz Academic Award and the newly created UNICAMP Teaching Award. He has had a CNPq Research Fellowship since 2006. He has been honored eight times in the computer science and computer engineering graduations.