Offloaded MPI message matching:
an optimistic approach

Jerónimo S. García[1,2], Salvatore Di Girolamo[2], Sokol Kosta[1],
J.J. Vegas Olmos[2], Rami Nudelman[2], Torsten Hoefler[3], and Gil Bloch[2]

[1]Aalborg Universitet
[2]NVIDIA
[3]ETH Zürich

Abstract—Message matching is a critical process ensuring the correct delivery of messages in distributed and HPC environments. The advent of SmartNICs presents an opportunity to develop offloaded message-matching approaches that leverage this on-NIC programmable accelerator, retaining the flexibility of software-based solutions (e.g., tailoring to application matching behaviors or specialized for non-MPI matching semantics) while freeing up CPU resources. This can be especially beneficial for intensive I/O systems, such as those protected with Post Quantum Cryptography. In this work, we propose a bin-based MPI message approach, *Optimistic Tag Matching*, explicitly designed for the lightweight, highly parallel architectures typical of on-path SmartNICs. We analyze several MPI applications, showing how most of them present a matching behavior suitable for offloading with the proposed strategy (i.e., low queue depths). Additionally, we show how, in those scenarios, offloaded optimistic matching maintains message rates comparable to traditional on-CPU MPI message matching while freeing up CPU resources.

I. INTRODUCTION

In distributed computing, effective communication among the processes of an application is critical for its performance and scalability. Message interfaces and frameworks, such as the Message Passing Interface (MPI) [1] and NCCL [2], provide application programmers with easy and powerful methods to manage the exchange of information among peers. A fundamental challenge in these interfaces is ensuring that messages sent by one process are correctly received by the intended recipient. This is accomplished through the message matching process.

Message matching involves pairing incoming messages with the appropriate receive requests and mapping the received data to user-defined buffers, while also ensuring that messages must comply with some constraints/semantics in their delivery (they depend on the interface used). This task becomes increasingly complex as the number of messages and processes grows. This is typically achieved using two queues: the posted receives queue (PRQ) and the unexpected messages queue (UMQ). The PRQ stores unresolved receive requests — requests to receive messages that have not yet arrived — while the UMQ stores messages that have been received but not yet requested. Then, these queues are searched for a match when a new message arrives or a new receive request is posted, respectively.

Efficient message matching is crucial for minimizing communication overhead, preventing bottlenecks and scalability issues in distributed applications [3], [4], especially important in intensive I/O (e.g., Post Quantum Cryptography-enabled clusters [5]) and exascale systems.

Typically, the message matching process is performed by the host CPU. This easily becomes a bottleneck, especially as the volume of messages increases while, at the same time, the CPU manages other tasks and compute. One solution to alleviate this issue is to offload the message matching process to accelerators, such as FPGAs [6], or to implement dedicated hardware circuitry in commercial Network Interface Cards (NICs) to perform message matching, also known as hardware tag matching [7].

However, with the emergence of in-network computing (INC) [8]–[10] in the form of novel hardware solutions such as SmartNICs (sNICs) and Data Processing Units (DPUs) [2], [11], there is a great opportunity to design in-network message matching techniques, which can leverage the benefits of INC to alleviate CPU load by offloading this task while keeping performance comparable to the CPU-based solution.

Additionally, offloading message matching to programmable sNICs offers several other benefits. First, since the matching process is implemented in software, it allows for custom optimizations and adjustments tailored to specific application characteristics or invariants - an approach that is not possible for hardware tag matching. Secondly, hardware tag matching does not handle unexpected messages, leaving to the application the search for a matching receive. Furthermore, it also brings improvements after the message matching process is completed, as further computations involving those messages can be performed directly on the network edge, eliminating the need to transfer the message back and forth between the sNIC and the host CPU. This is especially advantageous in GPU-centric communication, the one that dominates current AI and DL systems, where the matching can be performed on the sNIC, then the message is directly transferred to GPU memory, bypassing the CPU entirely [12].

Among the various message-based interfaces, MPI is arguably the most widely used in distributed computing, particularly in HPC environments. MPI's point-to-point commu-

nication semantics allow for both fully specified messages and the use of wildcards, such as `MPI_ANY_SOURCE` and `MPI_ANY_TAG`. These wildcards provide greater flexibility by enabling a receiving process to accept messages from any source or with any tag, although at the cost of extra complexity in the message matching logic. For this reason, typically, MPI implementations employ linked lists to maintain the ordering of posted receives and unexpected messages. For these cases, the search for a match becomes linear and dependent on the number of messages (i.e., the search time complexity is $\mathcal{O}(n)$) which can hinder the performance of some applications, specifically for the ones that use global communication patterns - where most processes send messages to most of their peers - or many-to-one communications, where a process receives messages from most of its peers simultaneously (e.g., `MPI_Gatherv`) [13]. The issue is also present in multi-threaded MPI (`MPI_THREAD_MULTIPLE`) where the need to lock the lists to ensure thread safety further exacerbates the problem [14].

To address the challenge of offloading the MPI message matching semantics, we propose "Optimistic Tag Matching", a novel bin-based message matching algorithm optimized for highly-parallel architectures, such as the ones commonly found in current SmartNICs [8], [15]. Our optimistic message matching approach offloads the entire message matching process of the host CPU, resulting in significant savings in CPU cycles, as the host CPU no longer performs the message matching. Furthermore, the implementation of our matching approach is highly generic, and designed to be used with any type of MPI application. While this generality is advantageous (i.e., our approach is not tied to specific invariants), a custom matching implementation tailored to a particular application or communication pattern would likely achieve better performance.

To validate the assumptions made during the design of our approach and to demonstrate the practical benefits of our algorithm, we have developed an MPI trace analyzer. This tool allows us to analyze the impact that different point-to-point communication patterns have on the matching data structures of our optimistic approach.

With this work, we aim to make a few notable contributions to the MPI message matching literature and problems:

i) **Innovative message matching approach**: We introduce and implement a cutting-edge message matching strategy specifically designed to enhance performance in highly parallel architectures.

ii) **Open-source MPI trace analyzer**: We provide an open-source MPI trace analyzer that enables detailed insights into message matching behavior.

iii) **Performance evaluation across applications**: We evaluate our proposed message matching approach by applying it to a variety of MPI applications using the trace analyzer we developed. This evaluation not only demonstrates the effectiveness of our method but also provides valuable data that can inform future developments and optimizations in the matter.

Through these contributions, we seek to improve performance, and understanding and offer practical tools for further improving MPI message matching, as well as some theoretical insights.

The rest of the paper is structured as follows. Section II provides a detailed overview of the MPI message matching process, emphasizing its performance challenges. It includes a brief literature review of alternative approaches and discusses the in-network computing architecture relevant to our proposed solution. Section III introduces our novel "Optimistic Tag Matching" approach in detail. Following this, Section IV covers the implementation of our optimistic matching approach on an on-path NIC accelerator. Section V presents and analyzes results related to tag usage and queue depth across several MPI applications, preceded by an introduction to the MPI trace analyzer developed for collecting these results and validating our matching approach. Section VI benchmarks the message rate in a prototype implementation of "Optimistic Tag Matching". Finally, Section VII offers concluding thoughts.

II. BACKGROUND AND RELATED WORK

A. MPI message matching

Message matching, also known as tag matching, is the crucial mechanism in MPI that ensures the correct delivery and order of messages in point-to-point (p2p) communications, as well as the process that maps these messages to user-registered buffers. It involves identifying which incoming/received message corresponds to which request. From here thereafter, received messages are called incoming messages, while requests are termed as posted receives. By matching specific information within the message, such as the process identifier ("rank"), a user-defined identifier ("tag"), and the communicator (or message channel), MPI implementations ensure the correct delivery of messages to the specified process in the correct order.

A more detailed explanation of the message matching process is provided in Figure 1. Starting as a process requesting messages (posting receives), Figure 1a delineates the logical flow for posting receives. First, in step 1, when a process posts a receive using functions like `MPI_Recv` or `MPI_Irecv`, it first checks the unexpected messages queue (UMQ). The UMQ holds messages that have arrived before the corresponding receive was posted. If a matching unexpected message is found, step 2a occurs, where the message is retrieved from the UMQ. Following this, in step 3a, the protocol to retrieve the message data (i.e., eager or rendezvous) is executed, and the message is successfully received. Conversely, if no matching message is found, step 2b is triggered, where a new entry is appended to the posted receives queue (PRQ). The PRQ holds information about pending receives, waiting for a corresponding message to arrive. The process then finishes with step 3b, where the posted receive is successfully recorded and awaits future matching.

Subsequently, when new messages arrive, a different process takes place, as shown in Figure 1b. In step 1, the PRQ is compared against the incoming messages. If a match is found,

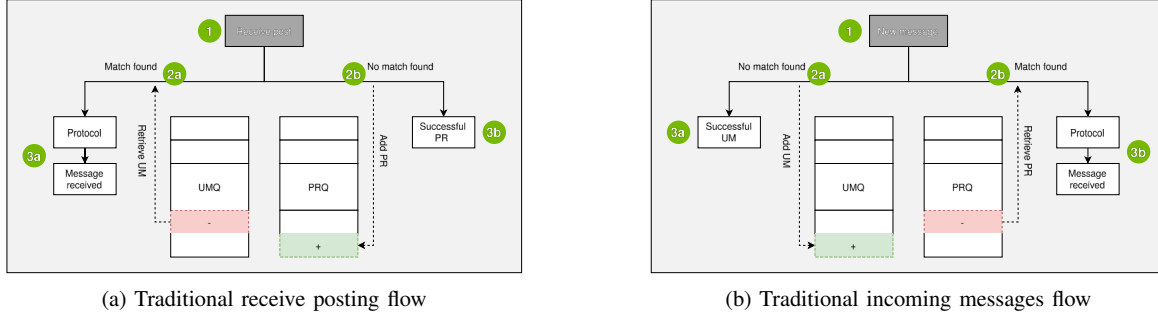(a) Traditional receive posting flow (b) Traditional incoming messages flow

Fig. 1: Traditional message matching scheme. Fig. 1a shows how a receive post is handled. Fig. 1b shows how incoming messages are handled. UMQ and PRQ denote the traditional matching queues: unexpected messages queue and posted receive queue, respectively. The symbols $+$ and $-$ denote the addition or removal of an entry.

Authors	Nature	Hardware accelerated	Matching strategy
Underwood et al. [16]	Static	✓	Associative lists
Dózsa et al. [17]	Static	✗	Rank-based
Flajslik et al. [13]	Static	✗	Bin-based
Bayatpour et al. [18]	Dynamic	✗	Traditional, bin- or rank-based
Xiong et al. [6]	Static	FPGA	Traditional
Denis Alexandre [4]	Static	✗	Bin-based

TABLE I: Overview of different tag matching approaches in the literature, sorted by publication date. 'Nature" indicates whether the matching algorithm is static (fixed for the application runtime) or dynamic (changes during its runtime).

step 2b involves removing the corresponding posted receive entry from the PRQ. The appropriate protocol is then used to gather the message data, and the message is successfully received. If no match is found, step 2a involves storing the incoming message in the UMQ. The process concludes with step 3a, where the message remains in the UMQ, awaiting a future match.

This process is necessary to satisfy the "message ordering" semantic imposed by MPI for p2p messages [19]. Furthermore, MPI offers the possibility of using wildcards (`MPI_ANY_SRC` and/or `MPI_ANY_TAG`) instead of the source and/or tag when posting a receive, allowing the application to match incoming messages in a more flexible way. Nevertheless, by using wildcards, the MPI tag matching process becomes more serialized, making it harder to optimize the matching structures of MPI implementations. For this reason, the MPI Forum updated the MPI standard to allow applications the possibility to hint implementations about the no-usage of wildcards [20].

This is why one way to reduce unexpected messages in MPI applications with extensive point-to-point communications is to post all immediate receives before transmitting any messages. Unexpected messages require temporary memory allocation while being received, increasing latency and reducing the throughput of applications.

B. Current MPI tag matching approaches

Improving the performance of MPI tag matching has been a longstanding research goal for enhancing HPC systems. Table I highlights several approaches discussed in the literature. Two main strategies have been explored: a) redesigning the message matching process by using optimized data structures and algorithms and b) offloading this process to hardware.

For the first strategy, there are proposals that use either bin-based approaches or rank-based approaches. More in-depth, Flajslik et al. [13] propose using two hash tables instead of the traditional two-queued implementation, where the hash function is based on the source rank, tag, and communicator. Then, to preserve message order, they also introduce timestamps to posted receives and unexpected messages, while, at the same time, they maintain a list of the order information for unexpected messages with wildcards. In their work, for an implementation with b bins, the average time spent in the search would be $\mathcal{O}(n/b)$, although in the case that posted receives or unexpected messages are contained within the same bin, the search time remains $\mathcal{O}(n)$.

Regarding hardware offloading strategies in the literature, there are proposals that suggest using FPGAs or NICs to offload MPI tag matching.

Building on the strategies previously discussed, our "Optimistic Tag Matching" approach integrates concepts from both hardware offloading and algorithmic optimization. This bin-based method employs three binned hash tables to categorize point-to-point messages according to their wildcards. A detailed explanation of our tag matching approach is provided in Section III.

C. Data Path Accelerator

Modern SmartNICs are equipped with a variety of processors and accelerators to provide additional computing power and offload network tasks. An example of this is the Data

Path Accelerator (DPA) found on the NVIDIA BlueField 3 DPUs (BF3) [15].

This power-efficient embedded processor accelerates networking tasks requiring access to NIC engines, such as packet processing and I/O workloads. Consequently, its architecture is optimized for these workloads. The DPA features numerous execution units, enabling the simultaneous processing of multiple tasks and the parallelization of others. Specifically, the DPA on the BF3 is equipped with 16 cores supporting 256 threads, with tasks executed in a run-to-completion fashion.

This architecture allows the DPA to excel in offloading simple, latency-sensitive network tasks and handling parallel workloads with small working sets [21] like our "Optimistic Tag Matching" algorithm.

III. OPTIMISTIC TAG MATCHING

With optimistic tag matching, we enable the offloading of MPI tag matching to programmable, power-efficient, on-NIC accelerators, such as the Data Path Accelerator (DPA). These accelerators typically include multiple lightweight compute cores (e.g., RISC-V [22]) and provide fast access to NIC resources. Due to their architecture, offloading complex and serial tasks to these accelerators is not ideal from a performance perspective. Instead, highly parallel and relatively light tasks are a good fit for offloading.

For tag matching, the MPI specification defines constraints on how incoming messages must be matched to posted receives. These constraints semantically serialize the matching, providing applications with a deterministic behavior. Our approach extracts parallelism from tag matching while remaining compliant with the MPI standard.

a) MPI matching constraints: The tag-matching constraints defined by MPI can be summarized as follows:

- **C1: Order of posted receives.** If a message matches two or more receives, then the receive that has been posted first must be matched first. This allows an application to know which receive will complete first if there are multiple receives that can match the same message.
- **C2: Non-overtaking messages.** If two messages from the same sender match the same receive, they should be matched in the same order they are sent. This allows the application to rely on message ordering.

A solution based on a linked list, where each element of the list is a posted receive, satisfies both constraints: each new receive is appended at the end of the list, making sure that receives are checked in the same order they are posted, and new messages are matched by scanning the list from the head, avoiding overtaking. However, this approach would be hard to parallelize without incurring high synchronization costs; hence, it is not a good candidate for offloading [17].

A. Overview

Optimistic matching is designed for lightweight parallel architectures, such as on-NIC programmable accelerators. In the following, we use the term *thread* to indicate a generic parallel task, which needs to be mapped on the specific
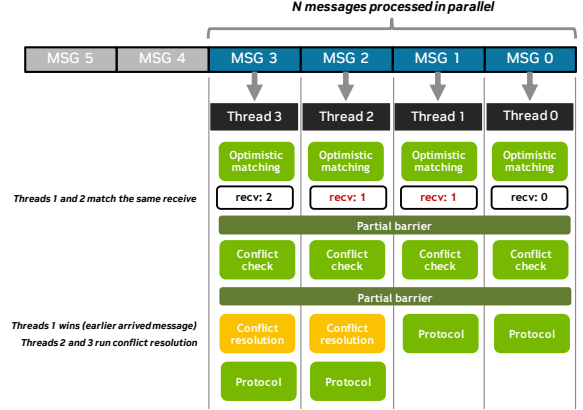

Fig. 2: Optimistic tag matching example on a stream of 6 messages. Four messages are processed in parallel. After the optimistic tag matching phase, threads 1 and 2 try to match the same receive but thread 1 wins because it is processing an earlier arrived message.

architecture where this solution is implemented (e.g., even handlers for DPA).

This strategy works on a stream of incoming messages, where blocks of N consecutive messages are processed simultaneously by different threads. Figure 2 shows an example with $N = 4$. The matching is performed optimistically, i.e., as if no other messages are matched concurrently, which could lead to a violation of the MPI matching constraints. After the optimistic matching phase, the threads check for conflicts (e.g., if two messages have been optimistically matched to the same receive) and resolve them if needed. After the conflict resolution phase, a thread can either have a receive matching the message or not. In the first case, the specific protocol, i.e., eager or rendezvous, is executed. Otherwise, the message is handled as unexpected. The idea behind this approach is that if there are few conflicts, then the optimistic matching phase will succeed without conflicts often, avoiding more expensive synchronization needed by the conflict resolution phase.

B. Indexing receives

To make this approach suitable to offload-enabled architectures, we need to minimize the computational cost of the matching, that is the number of matching attempts done to find a receive. For this, we organize the posted receives in different indexes, as depicted in Figure 3. The receives are split according to which wildcards they use:

- **No wildcards.** These are receives that have no wildcards as source rank or tag. They are indexed using a hash table with both the source and the tag as a key.
- **Source wildcards.** All receives that use the wildcard for the source rank but not for the tag. These receives are also indexed using a hash table. However, the key is only the tag, as the source is a wildcard.
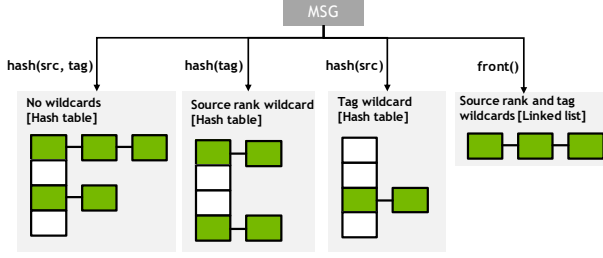
Fig. 3: A receive is indexed in one of the four data structures. For each incoming message, the four indexes are searched using the appropriate keys.

- **Tag wildcards.** All receives that use the wildcard for the tag but not for the source rank. Similarly to the previous class, these receives are also indexed in a hash table, but in this case, the key is only the source rank.
- **Source and tag wildcards.** Receives that use wildcards for both source rank and tag. As there are no fields to be used as keys, these receives are kept in a linked list: a message can be matched by the first receive in this list.

These indexes contain information needed for the matching and a pointer to the full receive descriptor. Receive descriptors are stored in a fixed-size table, where the size of the table determines the maximum number of receives that can be posted at the same time. If the number of posted receives exceeds this capacity, the application must fall back to software tag matching.

C. Optimistic matching

During the optimistic matching phase, each thread attempts to match its own message independently, searching for receives in each of the above-described indexes. A thread must ensure to select the oldest posted receive matching its message; otherwise, it will violate constraint **C1**. Inside an index, this constraint is already satisfied: if two receives have the same key, they will hash to the same hash table bucket. Hence, they will belong to the same linked list following the order in which they have been posted. The thread will stop once it finds the first matching receive in the linked list.

On the other side, the thread must enforce constraint **C1** between different indexes. This is done by labeling each receive with a monotonically increasing counter that reflects the posting order. After all indices have been checked, a thread might have up to four candidates receive (i.e., one per index) for a message and have to select the one with the minimum label, which is the one that has been posted first.

At this point, as we have multiple threads optimistically matching messages in parallel, we might end up with multiple messages matching the same receive, potentially violating constraint **C2**. As there is no synchronization between threads up to this point, two messages sent from the same sender and matching the same receive might be matched in the wrong order. It is worth noting that conflicts are time-dependent: two

threads attempt to book the same receive only if they process messages matching that same receive at the same time. To not violate **C2**, once a matching receive has been found, the thread tentatively books it. Each receive descriptor includes a booking bitmap of N bits that is used by threads to book the receive by setting their respective bit. The bitmap is then used to detect conflicts as described in the next section.

D. Conflict detection and resolution

A thread can determine whether there is a conflict on the selected receive by looking at the receive booking bitmap. However, before doing that, it must make sure that all threads processing messages that arrived before its own one have completed the optimistic matching phase and booked their candidate receive.

1) Synchronization: This synchronization is achieved using a partial barrier between threads. The barrier is partial because a thread must wait only on threads processing earlier messages. It is not necessary to wait for threads to process later messages because either they will match a different receive or, if they match the same receive, the current thread will have precedence according to **C2**. Additionally, since we operate on a stream of messages, waiting on future messages might stall the application as we do not know whether there will be new messages coming. As threads move over blocks of the incoming message stream, this barrier can be implemented by letting a thread i wait on all threads j with $j < i$. We implement the partial barrier with a bitmap, where each thread sets its own bit whenever it enters the barrier.

2) Conflict detection: Once threads are (partially) synchronized, they can check for conflicts. The check is done by looking at the booking bitmap of the selected receive: if a thread with a lower thread ID booked the receive, then there is a conflict, and the thread with the lowest ID wins the receive and can consume it. If a thread i detects a conflict, then all other threads $j > i$ need to enter the conflict resolution phase. In fact, even if a thread $j > i$ selects a candidate receive that is not conflicted, it might still happen that thread i selects the same candidate receive of j during the conflict resolution. In that case, i will have precedence because of constraint **C2**, and j must give up its receive.

3) Conflict resolution: To resolve a conflict, we identify two possible strategies that target different conflict scenarios.

a) Fast path: The fast conflict resolution is designed to quickly select the next receive candidate without additional synchronization costs. This strategy is useful for scenarios where the application posts long sequences of receives with the same source rank and tag, and incoming messages all match these receives. In other words, this is the case where all threads try to match the same receive at the same time. We define such a sequence of receive as *sequence of compatible receives*. This can be checked by looking at the booking bitmap of the candidate receive: if all threads selected it, then conflicted threads can try to use this strategy. In particular, the next candidate receive for thread i is selected as the receive with index $k + i$, where k is the index of the current
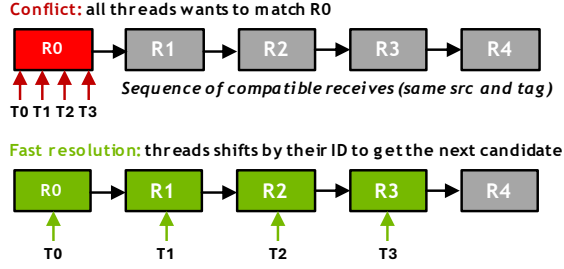
461

Fig. 4: Fast path for conflict resolution: if all threads want to match the first receive of a list of compatible receives (i.e., same source rank and tag, posted consecutively), then they can select the next candidate by shifting the current receive ID by their thread ID. Each thread must then check if the new candidate still belongs to the sequence of compatible receives.

candidate receive. Figure 4 shows an example of a sequence of compatible receives. Such a list can be contained in one of the indexes described in Section III-B. In this case, all threads want to match receive R0, but only thread T0 can consume it safely, as it has the lowest ID. Thread T1, can directly check receive R1 as candidate. Thread T2 should not check receive R1 as it already knows that it might be taken by thread T1, but will instead check receive R2.

However, this strategy assumes that the next candidate receive belongs to the same sequence of compatible receives of the current candidate one. If this is not true, then there could be a receive posted between k and $k + i$ that matches the current message and belongs to a different index data structure. In this case, the fast path should not be taken as it might break the MPI matching constraints. To prevent this issue, we assign to each receive a sequence ID at the time of posting. The MPI implementation on the host can increment the sequence number whenever it sees that the new receive is not compatible with the previous one (i.e., different source rank or tag). During the fast path execution, a thread will check if the new candidate receives belongs to the same sequence, and if not, it will switch to the slow path.

b) Slow path: A thread i hits the slow path when it cannot make assumptions on what will be the next candidate receive that will be selected by conflicted threads $j < i$. In this case, the thread must synchronize with the previous one, waiting for it to match and book a new receive. Once this happens, the thread is safe to match and book a new receive without violating the MPI matching constraints.

IV. OFFLOADING OPTIMISTIC TAG MATCHING

We now describe how optimistic tag matching can be offloaded to on-NIC programmable accelerators, defining how incoming messages are handled and how, after the optimistic matching phase, either the protocol or the unexpected message handling phases are executed. While this work focuses on the BlueField-3 DPA [15], this approach can be also mapped
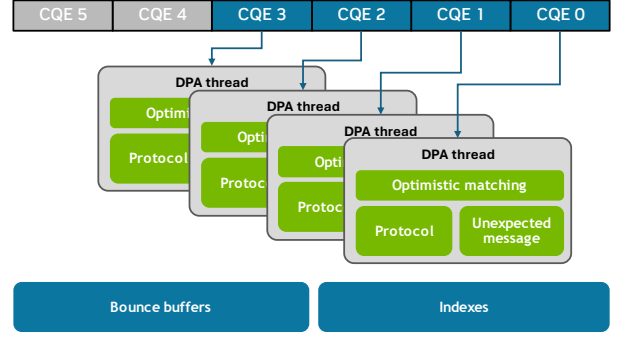

Fig. 5: Overview of the DPA-offloaded software architecture for optimistic tag matching.

onto other programmable on-NIC accelerators, like sPIN [8]. Figure 5 provides an overview of the offloaded optimistic tag-matching architecture.

A. Handling incoming messages

Messages are exchanged via RDMA send/receive operations. When an RDMA receive completes at the receiver, a completion notification is generated and stored in a RDMA completion queue. Incoming messages are staged into bounce buffers in NIC memory, which are pointed by the RDMA receive operations posted by the receiver. Bounce buffers are necessary because we only know the address of the user-provided receive buffer once the matching is performed. Additionally, staging data into bounce buffers has two more benefits: 1) it allow us to avoid registering user-provided buffers, and 2) as these buffers reside on the NIC, we eliminate the need to cross the PCIe bus twice for matching and data transfer, which would otherwise occur if the bounce buffers were on host memory.

In order to have multiple threads working on the same completion queue, we let each thread poll on the next expected completion queue entry for that thread: e.g., thread i will first wait for the completion notification i to be generated. Then, once message i is processed, it will wait on the completion notification $i + N$ for the next message (the completion queue needs to have a depth greater or equal to N). After a message is received and a DPA thread is triggered, the optimistic matching phase starts, as described in Section III.

B. Protocol handling

Once a receive is selected, a thread processing a message can move to the protocol handling stage. Typically, different protocols can be employed according to the message size. The protocol handling stage is not tightly coupled with the optimistic matching strategy described before. As the optimistic matching completes, the protocol can be handled either on the SmartNIC directly or on the host.

The eager protocol is used for small messages: the full message is sent by the sender and staged in a bounce buffer at

the receiver. After the matching occurs, the data is copied to the user-provided buffer if a receive is found. Otherwise, the message is handled as unexpected. If the optimistic matching phase is successful (i.e., a receive was found), the thread finds all the needed information to copy the message data from the staging buffer to the user-provided buffer in the descriptor of the matched receive.

For larger messages, the rendezvous protocol is used. The sender sends a Ready-To-Send (RTS) message, which might include some message data. The receiver matches the message and, if a receive is found, issues an RDMA read to move data from the sender-side buffer to the user-provided buffer on the receiver side. The message header brings information like the memory key to access the send buffer, while the receive descriptor contains information to access the receive buffer that is used for issuing the RDMA read.

C. Handling unexpected messages

A message is unexpected when there is no receive that can be matched to it. In this case, the message is stored for later match into an unexpected message buffer. When a new receive is posted, unexpected messages should be checked before indexing the receive and making it available for matching. If a matching unexpected message is found, the protocol handling stage is started: in the case of eager, the message is copied from the buffer where the unexpected message was stored to the user-provided one. For rendezvous, the stored data contains the information needed by the RDMA read.

We keep a set of indexes similar to the ones described in Section III-B to store information related to unexpected messages and that can be accessed at receive posting time. The main difference is that an unexpected message is indexed in each of these data structures, while a posted receive is indexed in only one of them (depending on the combination of wildcards it uses). It is worth noting that the MPI specification does not allow messages with wildcards, hence a message has source rank and tag always defined.

When a new receive is posted, only the index it belongs to is searched. If a matching receive is found, the receive is removed from all indexes before the protocol handling stage can be triggered.

D. Optimizations

We now discuss a set of possible optimizations that can be implemented to improve optimistic tag matching performance.
Inline hash values. To save compute resources on the SmartNIC, we can let the sender-side compute the possible hash values of a given message (i.e., `hash(src, tag)`, `hash(src)`, and `hash(tag)`), as they do not depend on the receiver-side state. These values can be included in the MPI message header and used on the SmartNIC to index the data structures described in Section III-B.
Early booking check. During the optimistic matching phase, we can check the booking bitmap of the receives we are considering for match. If the booking bitmap already signals that one or more threads with lower ID are trying to match

this receive, then we can skip it as we are guaranteed that this thread cannot consume it.
Lazy removal. If multiple threads consume a receive from the same list, then they might serialize on the removal of the receive from that list. To avoid this overhead, we implement a lazy removal scheme, where receives are marked as consumed (and considered in future matching attempts). Threads that successfully acquire a lock during the removal will proceed to clean up the list, removing also the marked receives.

E. Discussion

Posting receives. The cost of posting a receive to the DPA is comparable to that of hardware tag matching - the application must send a command to the DPA via a Queue Pair (QP).
Memory footprint. Each entry consists of a remove lock (4 bytes) and two pointers (8 bytes each) to the head and tail of the chained queue within the bin, totaling 20 bytes per bin. With the three index tables of our approach, this results in a total cost of 7.5 KiB for 128 bins. Additionally, each receive descriptor consumes 64 bytes. For example, to support 8 K receives (posted at the same time), we need to allocate about 520 KiB of DPA memory. For reference, DPA L2 and L3 caches in BlueField-3 are 1.5 MiB and 3 MiB, respectively. The implementation is expected to fall back to software tag matching if the DPA runs out of resources.
Multiple MPI communicators. Each MPI communicator is linked to its own set of index tables and data structures. If it is no possible to allocate DPA resources at communicator creation time, the MPI implementation is expected to fall back to software tag matching. Applications can provide MPI communicator info objects [23] to influence the offloading of tag matching for a given communicator.

V. MPI APPLICATION ANALYSIS

The optimistic tag matching approach is optimized for two scenarios: either there are no or only a few conflicts in the hash tables, or the application posts long lists of compatible receives. In this section, we analyze different MPI applications to extract information about their matching behavior, showing that the majority of analyzed applications fall into one of the two categories described above. To analyze applications, we developed an MPI trace analyzer, that runs existing MPI traces (of applications run at different scales), emulating the optimistic tag matching strategy and gathering statistics. This section is structured in two parts: first, we provide a general overview of the trace analyzer, and second, we discuss the results obtained from several MPI applications.

A. Trace analyzer overview

The MPI trace analyzer consists of two steps. a) a parsing stage to that converts MPI traces into a common and in-memory representation and, b) a trace processing stage in which the in-memory representation is processed using the intricacies of our tag matching approach.

a) Parsing stage: The initial phase consists of reading the trace files. These traces contain the complete execution of an MPI program. Currently, only a DUMPI text-traces reader is implemented, although the design of the application allows to easily add other formats.

Initially, the parser verifies the existence of a binary cache for the given input trace, as parsing the traces of an application is the most time-consuming step for the analyzer. If one is found, the remaining of the parsing can be skipped. On the other hand, the trace undergoes parsing, resulting in the generation of an in-memory representation of point-to-point (e.g., `MPI_Isend`), collective (e.g., `MPI_Alltoall`), one-sided (e.g., `MPI_Get`) and progress (e.g., `MPI_Wait`) MPI operations. We use a custom in-memory representation because it is easier to integrate and tailor to our specific needs.

The parsing is done in parallel in a per-rank fashion. This is done because, despite application traces not having a large number of ranks, they are usually quite long in the number of operations recorded on the trace. Finally, the in-memory trace representation is committed to storage for future re-runs of the application.

b) Processing stage: Following the parsing process, the trace processing stage commences. Initially, the key data structures for the processing of collisions - the three hash tables and the single linked list for messages based on the wildcards they use - and structures for the recording of statistics are created.

Each MPI operation within the in-memory representation of the trace gets sequentially processed until none remain. Only p2p and progress operations are processed, ignoring collectives and one-sided.

In the case of a p2p operation, the presence of wildcards is evaluated, as posted receives with different wildcards have different data structures, following our tag matching approach (see Section III). Concurrently, for each one of these p2p operations, various statistics, such as the current number of collisions, the percentage of empty bins per hash table, the percentage of p2p operations of each kind, and the usage of tags are updated.

In the event that the currently processed operation is of the progress type, all pertinent statistics and additional information regarding the current state of the hash tables and the linked list, such as the depth of said data structures are gathered. This compilation of information forms a data-point entry, encapsulating all progress achieved since the last recorded entry.

Finally, when all the operations have been processed, the data is formatted and recorded for later analysis.

B. Application analysis

a) Methodology: To analyze and investigate how different MPI applications perform regarding message matching, we (i) gathered MPI traces of different applications, (ii) designed a trace analyzer based on how our matching algorithm works (described in Section V-A) and (iii) analyzed the data generated from it. The trace analyzer and the script source code used to generate the results figures in this paper are
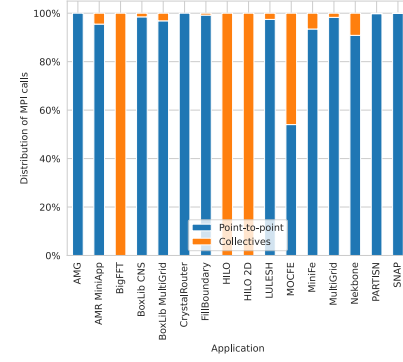

Fig. 6: Distribution of MPI calls for the application set

available under the project repository[1]. The traces used are part of the project 'Characterization of DOE mini-apps[2]' of the US National Energy Research Scientific Computing Center (NERSC). Regarding the set of applications analyzed, Table II provides a brief description of each application along with the number of processes recorded in the trace. The number of processes used for the analysis of each application was determined by the NERSC characterization project traces.

The experiments are machine independent. Only the MPI application traces are important.

b) Results: Our analysis begins with examining the distribution of the different types of MPI calls (point-to-point, collectives, and one-sided) used by the applications in our dataset. Figure 6 shows the proportion of each type of operation in the analyzed applications. It is evident that the majority of applications rely primarily on point-to-point MPI for communication, with minimal use of collectives, typically to perform synchronization among all processes. Only 3 applications in our dataset exclusively utilize p2p. Additionally, another 2 applications are entirely reliant on collectives (HILO has 2 different versions). Notably, none of the applications in the dataset use one-sided MPI operations for their communications.

The reduction in collisions is illustrated in Figure 7, which shows the queue depth of the applications using 1, 32, and 128 bins. The 1-bin configuration corresponds to the traditional tag matching approach. As seen, the average queue depth decreases from 8.21 to 0.8 with 32 bins and further to 0.33 with 128 bins, representing reductions of 90% and 95%, respectively. For example, in the BoxLib CNS application, the maximum queue depth decreases from 25 elements to 3 with 32 bins and to 1 with 128 bins, reflecting improvements of 88% and 96%, respectively.

VI. MESSAGE RATE BENCHMARK

In this section, we benchmark a prototype of the optimistic tag matching implemented on the NVIDIA DPA pro-

---

[1]https://github.com/ecn-aau/MPI-tgmtch-analyzer

[2]https://portal.nersc.gov/project/CAL/doe-miniapps.htm

Application	Description	Number of processes
AMG	Algebraic MultiGrid. Linear equation solver	8
AMR MiniApp	Single step AMR for hydrodynamics	64
BigFFT	Distributed Fast Fourier Transform	1024
BoxLib CNS	Compressible Navier Stokes equations integrator	64
BoxLib MultiGrid	Single step BoxLib linear solver	64
CrystalRouter	Proxy application for the Nek5000 scalable communication pattern	100
FillBoundary	Proxy application for ghost cell exchange using MultiFabs	1000
HILO	Modeling of Neutron Transport Evaluation and Test Suite	256
HILO 2D	Modeling of Neutron Transport Evaluation and Test Suite in 2D multinode	256
LULESH	Proxy application for hydrodynamic codes	64
MiniFe	Proxy application for finite elements codes	1152
MOCFE	Proxy application for Method of Characteristics (MOC) reactor simulator	64
MultiGrid	MultiGrid solver based on BoxLib	1000
Nekbone	Proxy application for the Nek5000 poison equation solver	64
PARTISN	Discrete-ordinates neutral-particle transport equation solver	168
SNAP	Proxy application for the PARTISN communication pattern	168

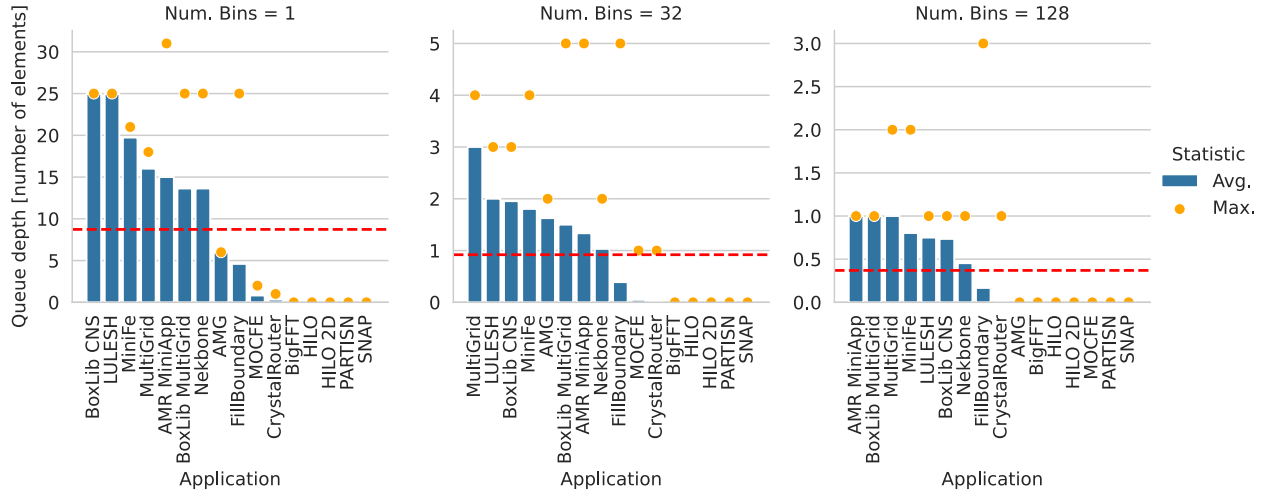TABLE II: Application traces analyzed, sorted by name alphabetically

Fig. 7: Queue depth for the different applications. The red line indicates the average queue depth across all applications for the given number of bins. Note that the plots are arranged in descending order of queue depth, not by application name.
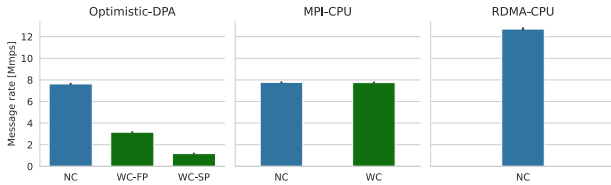

Fig. 8: Single-process message rate for the different configurations: optimistic tag matching, MPI tag matching on the CPU, and message exchange using RDMA on the CPU.

grammable accelerator (see Section IV). We focus on message rate, i.e., the number of messages matched per second, using small messages. This is the most critical metric for tag matching, as for larger messages the required message rate decreases. Experiments are run on two servers equipped with Intel(R) Xeon(R) Platinum 8480+ CPUs, 2 TB DRAM, and BlueField-3 network interfaces. The DPA-based prototype is compiled with the DPA compiler (dpacc) provided in DOCA 2.2 [24]. The DPA prototype is configured to use hash tables that are twice the maximum number of in-flight receives (which we set to 1024 for these experiments) and uses 32 DPA threads (limited by the bookkeeping bitmap size). The MPI benchmarks are run with OpenMPI 4.1.5.

We run a ping-pong benchmark, where a node sends a sequence of $k = 100$ messages to its peer. Once the peer receives (and matches) all messages in a sequence, it replies with an acknowledgment. We measure the message rate as k divided by the time from when the first message is sent to when the acknowledgment is received. For each run, we repeat the sequence 500 times. We test two main scenarios: all

465

posted receives have different source rank and tag combination (referenced as no-conflict case, **NC**), or all receives have the same source rank and tag (referenced as with-conflict case, **WC**). This allows us to get insights on the best and worst case for optimistic tag matching.

Figure 8 shows the benchmark results. The leftmost plot shows the performance of our prototype (Optimistic-DPA) for the no-conflict case, the with-conflict case using the fast path (**WC-FP**), and the with-conflict case using the slow path (**WC-SP**). We compare against two baselines running on the host CPU: MPI-CPU and RDMA-CPU. The RDMA-CPU is a reference baseline where no matching is performed and messages are exchanged via RDMA (i.e., no match conflicts are possible). We observe how optimistic tag matching has performance comparable with MPI-CPU for the non-conflict case. When there are conflicts, either the fast or the slow path is taken, causing a lower message rate due to the additional conflict resolution overheads. In all cases, the offloading fully frees the host CPU from tag-matching overheads.

VII. Discussion

We propose an optimistic approach to message matching that enables multiple threads to perform concurrent matching attempts and later check and resolve potential conflicts. In particular, we focus on satisfying MPI tag matching constraints, which are the most general and complex ones, especially because of wildcards. It is worth noting that MPI already allows applications to relax these constraints by specifying communicator hints. In principle, these hints can be propagated to the offloaded matching solution, reducing matching costs. For example, `mpi_assert_no_any_tag` and `mpi_assert_no_any_source` indicate that no receive with tag and source wildcards will be posted, respectively. These hints can be combined together to signal that no wildcards will be used at all. Another example is `mpi_assert_allow_overtaking` that relaxes matching order. Additionally, by having a software solution to offloaded message matching, we retain the flexibility of specializing the matching according to the specific communication library being used, which could adopt weaker matching constraints than MPI (e.g., NCCL [2]).

Lastly, running message matching on the SmartNIC not only saves CPU cycles but also enables more high-level tasks to be offloaded to the SmartNIC, such as tasks depending on incoming data. In order to be executed, the incoming message needs to be matched, so the receive upon which the task depends can be completed. Offloading tag matching is a necessary step to be able to offload the full chain of actions (e.g., match; complete receive; run task). Examples of such tasks are collective operations, which are normally built on top of point-to-point operations, and hence need matching to be performed in order to be offloaded.

VIII. Conclusion

In this work, we introduce a new message matching strategy optimized for highly parallel architectures. This strategy adopts an optimistic approach that allows for the extraction and exploitation of parallelism during the matching phase, which is normally meant to be serial due to the matching constraints. We specialize our solution for two common cases: either receives are well spread over the index data structures (no conflicts) or there are long sequence of subquent receives with the same source rank and tag. To confirm that this is indeed the common case, we developed an MPI trace analyzer, that allows us to emulate optimistic matching performance over DUMPI traces. Our analysis shows that the number of unique source/tag posted receives is low, indicating that the receives are well spread in the hash tables, keeping collisions low.

Moreover, we demonstrated that *Optimistic Tag Matching* performs comparably to traditional MPI tag matching when collisions are absent. This is a significant advantage, as our approach offloads the tag matching process from the host CPU, removing matching overheads from the host CPU while maintaining comparable performance.

References

[1] L. Clarke, I. Glendinning, and R. Hempel, "The MPI message passing interface standard," in *Programming Environments for Massively Parallel Distributed Systems* (K. M. Decker and R. M. Rehmann, eds.), pp. 213–218, Birkhäuser.

[2] NVIDIA, "NVIDIA collective communications library (NCCL)."

[3] S. Kumar, A. R. Mamidala, D. A. Faraj, B. Smith, M. Blocksome, B. Cernohous, D. Miller, J. Parker, J. Ratterman, P. Heidelberger, D. Chen, and B. Steinmacher-Burrow, "PAMI: A parallel active message interface for the blue gene/q supercomputer," in *2012 IEEE 26th International Parallel and Distributed Processing Symposium*, pp. 763–773. ISSN: 1530-2075.

[4] A. Denis, "Scalability of the NewMadeleine communication library for large numbers of MPI point-to-point requests," in *2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CC-GRID)*, pp. 371–380.

[5] "Preparing for quantum-safe cryptography."

[6] Q. Xiong, A. Skjellum, and M. C. Herbordt, "Accelerating MPI message matching through FPGA offload," in *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 191–1914, IEEE.

[7] 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 *Proceedings of the 26th European MPI Users' Group Meeting*, EuroMPI '19, pp. 1–10, Association for Computing Machinery.

[8] T. Hoefler, S. Di Girolamo, K. Taranov, R. E. Grant, and R. Brightwell, "sPIN: High-performance streaming processing in the network," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–16, ACM.

[9] W. Schonbein, R. E. Grant, M. G. F. Dosanjh, and D. Arnold, "INCA: in-network compute assistance," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–13, ACM.

[10] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz, "Forwarding metamorphosis,"

[11] Asterfusion, "Helium DPU."

[12] G. Shainer, A. Ayoub, P. Lui, T. Liu, M. Kagan, C. R. Trott, G. Scantlen, and P. S. Crozier, "The development of mellanox/NVIDIA GPUDirect over InfiniBand—a new model for GPU to GPU communications," vol. 26, no. 3, pp. 267–273.

[13] M. Flajslik, J. Dinan, and K. D. Underwood, "Mitigating MPI message matching misery," in *High Performance Computing* (J. M. Kunkel, P. Balaji, and J. Dongarra, eds.), Lecture Notes in Computer Science, pp. 281–299, Springer International Publishing.

[14] W. W. Schonbein, M. Dosanjh, R. Grant, and P. Bridges, "Measuring multithreaded message matching misery.." ISSN: 0302–9743 Volume: 11014.

[15] NVIDIA, "BlueField DPA subsystem."

[16] K. Underwood, K. Hemmert, A. Rodrigues, R. Murphy, and R. Brightwell, "A hardware acceleration unit for MPI queue processing," in *19th IEEE International Parallel and Distributed Processing Symposium*, pp. 96b–96b, IEEE.

[17] G. Dózsa, S. Kumar, P. Balaji, D. Buntinas, D. Goodell, W. Gropp, J. Ratterman, and R. Thakur, "Enabling concurrent multithreaded MPI communication on multicore petascale systems," in *Recent Advances in the Message Passing Interface* (R. Keller, E. Gabriel, M. Resch, and J. Dongarra, eds.), Lecture Notes in Computer Science, pp. 11–20, Springer.

[18] M. Bayatpour, H. Subramoni, S. Chakraborty, and D. K. Panda, "Adaptive and dynamic design for MPI tag matching," in *2016 IEEE International Conference on Cluster Computing (CLUSTER)*, pp. 1–10. ISSN: 2168-9253.

[19] "Semantics of point-to-point communication."

[20] Message Passing Interface Forum, "MPI: a message-passing interface standard version 4.0."

[21] X. Chen, J. Zhang, T. Fu, Y. Shen, S. Ma, K. Qian, L. Zhu, C. Shi, Y. Zhang, M. Liu, and Z. Wang, "Demystifying datapath accelerator enhanced off-path SmartNIC."

[22] "Specifications – RISC-v international."

[23] "Communicator info: Hints."

[24] NVIDIA, "DOCA documentation v2.2.1."

Appendix: Artifact Description

I. OVERVIEW OF CONTRIBUTIONS AND ARTIFACTS

A. Paper's main contributions

- C_1 A novel MPI tag matching approach (Optimistic Tag Matching) prototype for a highly-parallel SmartNIC architecture.
- C_2 MPI trace analyzer for tag matching behavior.

Due to confidentiality, the source code of the tag matching prototype (C_1) and its benchmark cannot be disclosed - only the analysis script for the benchmark results and the required data to reproduce the corresponding paper elements is provided.

B. Computational artifacts

- A_1 https://github.com/ecn-aau/MPI-tgmtch-analyzer/exampl
- A_2 https://github.com/ecn-aau/MPI-tgmtch-analyzer

Artifact ID	Contributions Supported	Related Paper Elements
A_1	C_1	Figure 8
A_2	C_2	Table 2, Figures 6-7

II. ARTIFACT IDENTIFICATION

A. Computational artifact A_1

1) Relation to contributions: The artifact A_1 features the script to generate the message rate plot for the prototype implementation of "Optimistic Tag Matching" (C_1) results gathered from its benchmark.

2) Expected results: A plot showing the different message rates for the prototype implementation (fast and slow path), for MPI tag matching on the CPU and a baseline of the peak message rate for the connection in the test.

3) Expected reproduction time: The expected reproduction time is around 1 minute.

4) Artifact setup:

- Hardware: Any general-purpose computer that can run a Python 3 interpreter.
- Software: Python 3 interpreter and an IDE that supports Jupyter notebooks (e.g., VSCode).
- Datasets/Input: Provided with the source code.
- Installation and deployment: Any Python 3 interpreter should suffice. The Python environment must contain the **pandas**, **matplotlib**, **seaborn** and **numpy** libraries.

5) Artifact evaluation: The artifact consist on a single task $T1$, that using the data provided with the artifact (benchmark results) produces the Figure 9 from the paper. The script is self-contained.

6) Artifact analysis: The plot script calculates some statistics based on the data from the tag matching message rate benchmark in order to generate the corresponding plot presented in the text.

B. Computational artifact A_2

1) Relation to contributions: The artifact A_2 features of the analyzer (C_2) used to generate information about the matching behavior of the different MPI applications and the post execution analysis script that generates the plots and figures displayed in the paper.

2) Expected results: After executing the analysis for all applications, the artifact generates a folder for each application in the analysis, and, for each application, it generates 6 folders representing the number of bins used (from 1 to 256, in powers of 2).

Then, this data is fed into the analysis script to generate the plots in the text.

3) Expected reproduction time: It wildly depends on the specifications of the running machine, but in our case it is around 45 minutes to 1 hour.

4) Artifact setup:

- Hardware: Any general-purpose computer that can run the Rust compiler in Tier 1 with host tools (x64, aarch64) and a Python 3 interpreter.
- Software: Rust compiler (1.78+), Python 3 interpreter and an IDE that supports Jupyter notebooks (e.g., VSCode).
- Datasets/Input: The inputs for the analyzer are the application traces. They are provided with the source code.
- Installation and deployment: Compiling the analyzer only requires to execute `cargo b --release` on the parent folder of the project. It automatically downloads all required dependencies. For the plots script, the Python environment must contain the **pandas**, **matplotlib**, **seaborn** and **numpy** libraries. In order to able to execute the analyzer application, the Structural Simulation Toolkit (SST) DUMPI Trace Library (https://github.com/sstsimulator/sst-dumpi) must be installed and the run script (`run.sh` second line) must be pointed to the SST installation.

5) Artifact evaluation: This artifact consists of 2 tasks, $T1$ and $T2$. Task $T1$ generates information about the tag matching behavior of a set of applications, generating as output a set of data. Then, this output is fed to task $T2$ for the generation of plots.

Task $T1$ is self-contained, meaning that all the parameters used in generating the paper elements are already provided.

468

6) Artifact analysis: The raw data generated by analyzer for each application is join together in the plot script and then, depending on the plot, some statistics are calculated based on them, generating the plots presented in the text.