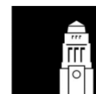


UNIVERSITY OF LEEDS

This is a repository copy of *Faster and Scalable MPI Applications Launching*.

White Rose Research Online URL for this paper:
https://eprints.whiterose.ac.uk/197061/

Version: Accepted Version

Article:

Dai, Y, Dong, Y, Xie, M et al. (5 more authors) (2022) Faster and Scalable MPI Applications Launching. IEEE Transactions on Parallel and Distributed Systems. ISSN 1045-9219

https://doi.org/10.1109/tpds.2022.3218077

© 2022, IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

Reuse
Items deposited in White Rose Research Online are protected by copyright, with all rights reserved unless indicated otherwise. They may be downloaded and/or printed for private study, or other acts as permitted by national copyright laws. The publisher or other rights holders may allow further reproduction and re-use of the full text version. This is indicated by the licence information on the White Rose Research Online record for the item.

Takedown
If you consider content in White Rose Research Online to be in breach of UK law, please notify us by emailing eprints@whiterose.ac.uk including the URL of the record and the reason for the withdrawal request.

eprints@whiterose.ac.uk
https://eprints.whiterose.ac.uk/

Faster and Scalable MPI Applications Launching

Yiqin Dai, Yong Dong, Min Xie, Kai Lu, Ruibo Wang, Juan Chen, Mingtian Shao, and Zheng Wang

Abstract—Distributed parallel MPI applications are the dominant workload in many high-performance computing systems. While optimizing MPI application execution is a well-studied field, little work has considered optimizing the initial MPI application launching phase, which incurs extensive cross-machine communications and synchronization. The overhead of MPI application launching can be expensive, accounting for over 200 million processor core hours and 15% of the user core time annually on the production Tianhe-2A supercomputer, which will increase as the number of parallel machines used grows. Therefore, it is critical to optimize the MPI application launching process. This paper presents a novel approach to optimizing the MPI application launch. Our approach adopts a location-aware address generation rule to eliminate the need for address exchange and a topology-aware global communication scheme to optimize cross-machine synchronization. We then design a new application launch procedure to support the proposed optimizations to further reduce the pressure of the shared I/O system. Our techniques have been deployed to production in the Tianhe-2A supercomputer and the Next Generation Tianhe Supercomputer. Experimental results show that our approach scales well and outperforms alternative schemes, reducing the MPI application launching time by over 29% with 320K MPI processes.

Index Terms—Message Passing Interface (MPI), High Performance Computing (HPC), MPI Application Optimizaiton.

1 Introduction

Parallel applications are the dominant workload in high-performance computing (HPC) systems. Many of these parallel programs run across multiple compute nodes and processors and use the Message Passing Interface (MPI) for distributed communications and work distribution [1]–[3]. Effective management of MPI applications is thus vital for improving system utilization and application performance for HPC systems.

There is an extensive body of work in optimizing MPI applications [4]–[7]. The vast majority of the prior work focuses on the program execution stage, where MPI workers have been distributed and initialized to run on assigned computing machines and the required libraries, data, and hardware resources had been prepared. However, the overhead of the MPI launching process, which involves a range of sub-tasks like allocating the required computing resources, distributing the MPI jobs or workers to the assigned compute nodes, instantiating the process on each node, preparing the input files, and the initialization required the MPI runtime, has been largely ignored in prior work. While such overhead may seem to be trivial for a single-user program, the aggregated time spent on this process can quickly grow to many core hours on a large-scale HPC cluster with many MPI jobs running. For example, our study of the workload traces on the highly ranked Tianhe-2A supercomputer [8] suggests that MPI application launching accounts for over 200 million core hours annually, or over 15% of the total core-hours spent by users, and we expect this number to grow as the scale and size of MPI applications continue to increase on future generation HPC systems.

- *Y. Dai, Y. Dong, M. Xie, K. Lu, R. Wang, J. Chen, and M. Shao are with the College of Computer, National University of Defense Technology, China.*
  E-mail: {daiyq, yongdong, xiemin, kailu, ruibo, juanchen, shaomt}@nudt.edu.cn
- *Z. Wang is with the School of Computing, University of Leeds.*
  E-mail: z.wang5@leeds.ac.uk

Therefore, there is a critical need to optimize MPI launch to reduce the application delay and improve system utilization. The benefit of accelerating the launching of the MPI application is more pronounced during the application development stage and the debugging phase when deployed a new hardware system configuration. Both scenarios require frequently running MPI applications to test the design choice. Furthermore, speeding up the launching process is also useful for fault tolerance by reducing the delay in restarting a failed job.

The MPI launch process consisting a sequence of sub-tasks (termed as *the application launch sequence* in this paper), including loading the executable binary and its supporting libraries, setting up the internal library state, discovering local network interfaces to configure the communication channel, and preparing resources for communicating with other processes. As part of the process, distributed MPI workers need to agree on the network addresses used for communications and make sure all workers have initialized their state. Such synchronization steps are implemented by first performing a global exchange operation, which is then followed by a global barrier to synchronize the worker states for moving to the computation stage. The overhead of the synchronization steps depends on the participating number of compute nodes and MPI workers and can account for 4.2% to 42.3% of the launching time.

This paper presents a new approach for optimizing MPI application launch. Our approach is designed to reduce the overhead of coordinating the communication addresses and the global barrier operation during the MPI application launch phase. To determine the endpoint addresses of remote peers, we utilize the global scheduling information provided by the resource manager (RM), which provides the compute node an MPI work is placed, and the network hardware information. With our new address generation strategy, the application launch sequence is fully decoupled from the global address exchange, which, in turn, improves the efficiency and scalability of the application

launch. Our barrier optimization scheme uses the topology information of the interconnection network to accelerate the barrier among all application processes. Building upon our optimizations, we then propose a new launch sequence. Our approach permits each MPI process to initialize its own communication channel and assemble the communication addresses of remote peers locally. By doing so, we can greatly reduce the overhead of a global barrier operation.

We implemented our approach by modifying the system software stack and deploying it to the Tianhe-2A supercomputer system [8] and the Next Generation Tianhe Supercomputer. We compare our approach against alternative methods [9]–[11]. Our extensive evaluation shows that our technique gives the fastest application launch time, reducing the MPI launching overhead by 30% for an MPI program with 320K processes running on 20K nodes when compared to prior launch techniques.

This paper makes the following contributions:

- It presents a new optimization technique to reduce the MPI application launching overhead (Sections 3.1 and 3.2);
- It develops a new launch sequence to accelerate the MPI launch process (Section 3.3);
- Extensive experiments conducted on production supercomputer systems show that the proposed technique can greatly speed up the launch phase of large-scale MPI applications (Section 5).

A preliminary version of this article entitled “The Fast and Scalable MPI Application Launch of the Tianhe HPC system” by Yiqin Dai, Yong Dong, Min Xie, Kai Lu, Ruibo Wang, Mingtian Shao, Juan Chen [12] appeared in the 36th IEEE International Parallel & Distributed Processing Symposium (IPDPS), 2022. This extended version makes several additional contributions to the conference paper: (1) we port our approach to a new HPC system with 320K processes to show the portability and scalability of the techniques (Section 5.3); (2) we greatly extend our evaluation by showing the impact of file caching (Section 5.1), the experimental results of Open MPI (Section 5.3), and the extra time and space overhead of the proposed method (Section 5.4). We demonstrate the optimization results of the proposed method on actual workloads in a production environment (Section 5.5). In addition, we study the total runtime and results of the benchmark programs to evaluate the potential impact of the proposed optimized launch technique on MPI applications in terms of correctness and performance (Section 5.6); (3) we present a new, more compact launch sequence design (Section 3.3), leading to better performance than the approach proposed in [12]; (4) we provide a set of new generation rule for the virtual port (VP_NUM) (Section 3.1.1) to accommodate different node usage patterns (exclusive or shared); (5) and we extend the background and motivation sections (Section 2) to describe our system software stack.

2 Background and Motivation

2.1 TH-Express interconnection

Our techniques have been deployed to production in the Tianhe-2A supercomputer. This system uses TH-Express, an in-house interconnection network for cross-node communication. However, our techniques can be easily ported to other high-performance interconnection networks like InfiniBand [14] and Tofu interconnect [15], [16] too.

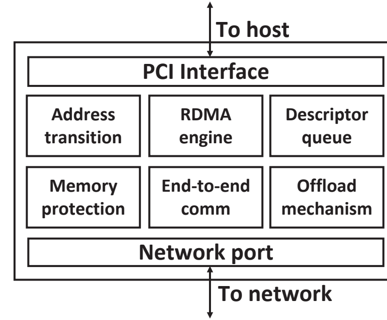


Fig. 1. Overview of the TH-Express network interface chip (NIC) (reproduced from from [13]).

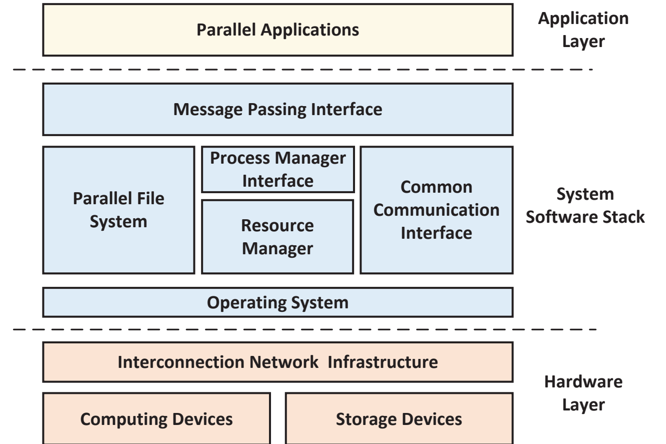


Fig. 2. A typical HPC infrastructure for supporting MPI applications.

The TH-Express network builds upon two specialized chips, a Network Interface Chip (NIC) and a Network Router Chip (NRC). The latter adopts a high-level routing structure, allowing the construction of a variety of interconnection topologies with mixed optoelectronics. The NIC provides interconnection communication services for various systems and applications within a compute node and uses the NRC to realize data transmission with each compute node of the whole HPC system.

As shown in Fig. 1, TH-Express NIC connects a compute node through a PCIe interface and communicates with the interconnect fabric through the network port. The TH-Express NIC provides several state-of-art mechanisms to implement protected user-level communications - the most important of which is the virtual port (VP) mechanism. With this mechanism, when a process is initialized, it must exclusively bind a VP on a NIC to perform inter-process communication. Therefore, a user process can be uniquely identified by the combination of NIC and VP. VP is essentially a combination of a small set of memory-mapped registers and a bunch of in-memory data structures organized as several queues. All data structures can be mapped to userspace to be accessed concurrently with protection.

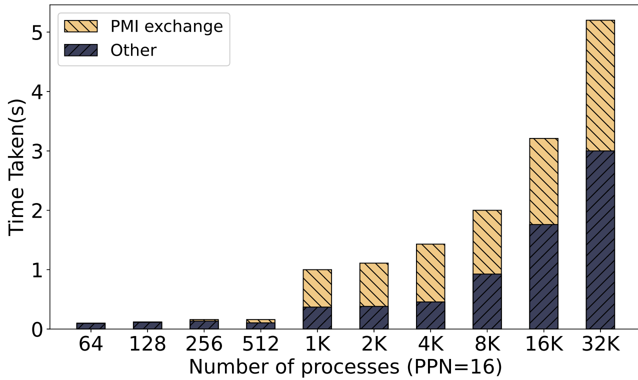


Fig. 3. Breakdown of time spent in MPI_Init.

2.2 System software stack

Fig. 2 depicts the typical HPC hardware and software stack for MPI. Our work lies in the software stack by utilizing the underlying hardware interconnection networks. It is a standard practice to use a communication library that implements a generic interface to hide the hardware complexity. Examples of such libraries include Unified Communication X (UCX) [17], OpenFabrics Interfaces (OFI) [18] and Common Communication Interface (CCI) [19]. Our implementation builds upon the UCX interface that was supported by TH-Express [20] which establishes the mapping from the UCX data structure to the resource of the TH-Express interconnection and implements a high-speed data transfer protocol. Furthermore, our evaluation system uses the Slurm [21] resource manager (RM) for job distributions and hardware resources allocation. The RM uses the process management interface (PMI) that is supported by mainstream MPI frameworks like MPICH [22] and Open MPI [23] to exchange information required by distributed MPI workers.

2.3 Problem Scope

The classical MPI application launch process (or sequence) consists of several steps. The user first submits a job, and then the RM schedules the job and allocates resources for the job. Next, the RM sends the job information to an RM daemon running on each allocated compute node, where the RM daemon then retrieves the file system and instantiates the processes on each node. Finally, each MPI process invokes the standard MPI_Init method to complete its initialization before performing computation. Our work solely focuses on the last three steps, i.e., after the job is scheduled to run on the allocated resources until the completion of MPI_Init.

As part of the MPI_Init process, each MPI worker must discover the local interconnection resources and generate its endpoint address for interactions with remote peers. Then, an MPI library uses the standard interface provided by PMI to exchange the network endpoint addresses for inter-process communication - a process known as PMI exchange. Fig. 3 shows the breakdown of the time taken during MPI_Init when launching an MPI microbenchmark using a different number of compute nodes in Tianhe-2A. The number of processes running on each compute node (PPN) is 16, one for each processor core. With the increase of node scale, the time consumed by barrier-based PMI exchange gradually grows and is expected to dominate the total MPI_Init time in larger applications. As a result, the barrier-based PMI exchange can become a bottleneck in the MPI launch process for MPI applications with hundreds or thousands of nodes. Our work aims to reduce the overhead of PMI exchanges during the MPI launch process.

3 Our Approach

This section presents the details of our MPI application launch approach. We first develop a location-aware address generation strategy (Section 3.1) to eliminate the need for end-point exchanges. Our technique employs a topology-aware global barrier optimization to reduce the overhead of global barriers (Section 3.2). With these two optimizations in place, we then introduce a new MPI launch sequence that supports the above optimizations for PMI exchange (Section 3.3) and relieves the library retrieval storm problem. Our prototype implementation has been deployed to the Tianhe-2A supercomputing system by utilizing the TH-Express interconnection network.

3.1 Location-Aware Address Generation Rule

The underlying reason for the MPI process to exchange network endpoint addresses during application launch is that the address generated by each process does not follow a consistent rule and is somewhat random. To create the endpoint address, each MPI process must locally find and select the available resources to establish a communication channel based on the current resource utilization. As a result, the remote peers cannot acquire the communication address of an MPI process until it has been generated.

Our approach avoids this problem by introducing a location-aware address generation rule to eliminate the randomness of address generation. In this way, each process can obtain the address of its remote peers according to the pre-defined endpoint address protocol without incurring cross-node communications. We describe our approach based on UCX (Section 2.2), but our methodology can be applied to other network communication interfaces too.

Fig. 4 shows the basic communication entity of the TH-Express-based UCX. Here, a communication entity contains the endpoint address of a particular MPI process, and the communication entities within the same MPI application constitute a complete communication domain of the application. We note that the total length of the communication entity and the length of each field are fixed.

Community entity. Each communication entity is 38-byte long. The first 30 bytes of a communication entity contain various information required for inter-process communication except for the endpoint address. The last 8 bytes of a communication entity are the network endpoint address over the high-performance interconnection network. Specifically, the Header is the flag of the UCX version. The Model-Info (Model Information) and Dev-Info (Device Information) indicate the information of communication modules and devices, respectively. The Length is the length

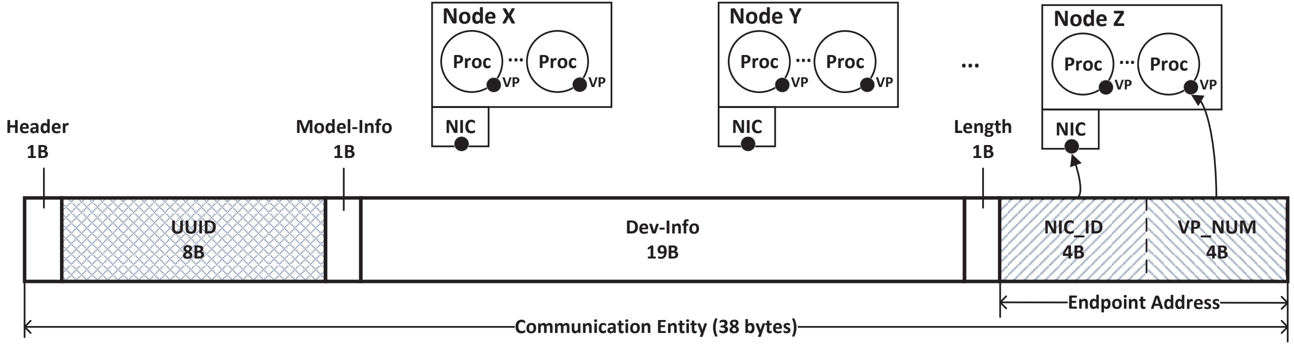


Fig. 4. The communication entity based on TH-Express interconnection.

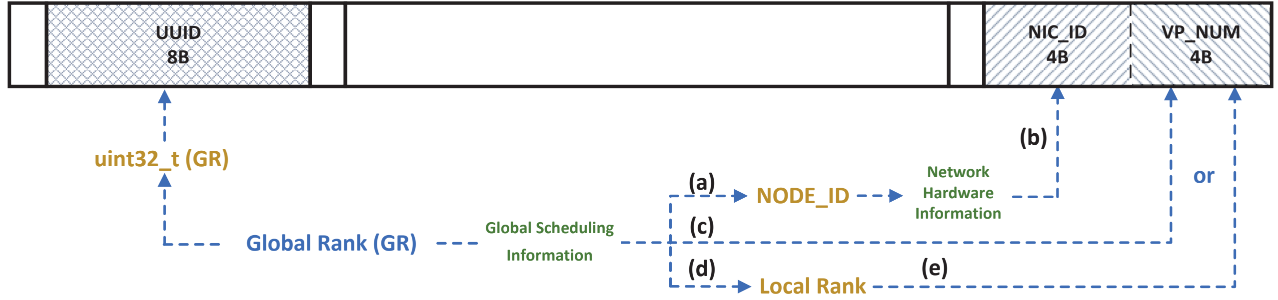


Fig. 5. The location-aware address generation rule.

of the endpoint address. These fields are the same for all communication entities within a communication domain of the same Tianhe HPC system, so they do not require new generation rules. However, the 8-bytes UUID, the identifier of the communicating entity, and the 8-bytes endpoint address must be unique within the communication domain, which requires a new address generation rule.

Endpoint address. We use an 8-byte endpoint address consisting of a 4-bytes NIC identifier (NIC_ID) and a 4-bytes virtual port number (VP_NUM). The NIC_ID can be used to locate a remote NIC, while the VP_NUM can be used to find the virtual memory space of the remote process on the specified NIC (see also Section 2.1). Fig. 5 presents the proposed generation rule of the key fields, including UUID, NIC_ID, and VP_NUM in the communication entity. We note that each process has a global unique identifier named Global Rank (GR) within an application. The GR is assigned to each process by the RM when it is placed on a determined node and is passed to the MPI library via environment variables, which is a common implementation of current HPC systems. Therefore, GR is available for both the RM and the MPI library. As we will see later, GR plays a crucial role in our proposed address generation rule because it is unique within the communication domain and can be linked to the process location information in the RM.

3.1.1 Endpoint address generation

Fig. 5 presents our proposed generation rule of the key fields of the endpoint address in the communication entity. These include UUID, NIC_ID, and VP_NUM. We describe these key fields as follows.

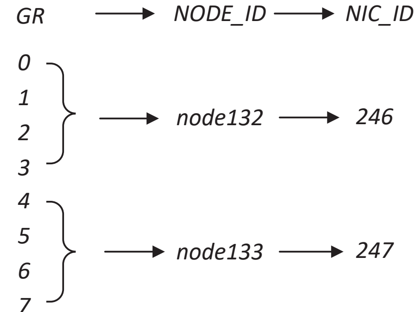


Fig. 6. The generation rule of NIC_ID which is used to locate a remote network interface chip (NIC).

UUID. We convert the GR of each process into a 32-bit integer and directly use it as a UUID, that is $UUID = uint32_t(GR)$.

NIC_ID. We map the global rank of a process to the NIC_ID of the node where the process resides. We do so by first using the global scheduling information to find the location information of each process, i.e., on which compute node each process is located. This step completes the mapping from the GR to the NODE_ID (see Fig. 5 (a)). We then use the network hardware information to get the NIC_ID corresponding to each compute node. This step completes the mapping from NODE_ID to NIC_ID (see Fig. 5 (b)). Fig. 6 illustrates the NIC_ID mapping process for a job with a total process count of 8 and a PPN of 4. The global scheduling information is stored in the RM, and the network hardware information can be read into the RM from a pre-prepared file when the RM is started. Specifically, on

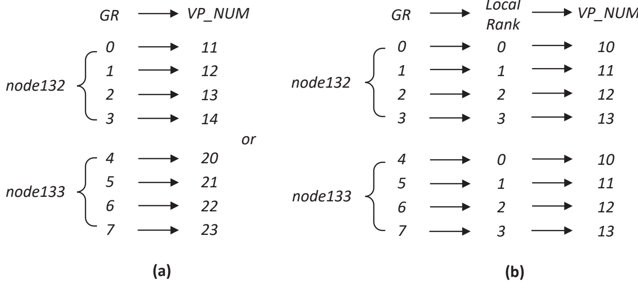


Fig. 7. The generation rule of VP_NUM.

the Tianhe HPC system, we store the network hardware information on each compute node as a file that contains mappings from the NODE_ID to the NIC_ID. RM reads all the network hardware information from this file to build a hash table to store it at startup. When launching the MPI application, RM can use the NODE_ID as an index to look up its corresponding NIC_ID in the hash table. Since global scheduling information and network hardware information are available to RM but not to the MPI library, it is logical for RM to take on the task of mapping from GR to NIC_ID before instantiating processes and passing the mapping results to the MPI library through PMI wireup. This process is detailed in Section 3.3. At this time, the RM stores a key-value pair in the PMI key-value store for each process, where the key is the GR of the process and the value is the NIC_ID of the process. The PMI wireup used to store key-value pairs is specifically the *PMI2_Put* function or the *PMIx_Put* function. When the mapping results are available, each process can locally obtain the NIC_ID corresponding to any remote processes from the PMI key-value store through PMI wireup. At this time, the local process uses the GR of the remote process as a key to look up the corresponding key-value pair in the PMI key-value store and obtains the NIC_ID of the remote process through the key-value pair. The PMI wireup used at this time is the *PMI2_Get* function or the *PMIx_Get* function.

VP_NUM. We design two types of generation rules for VP_NUM, described as follows.

When multiple user applications share a compute node, processes from different user programs can use the computational resources of the node at the same time. In this case, we consider VP as an exclusive resource to be scheduled by the RM (see Fig. 5 (c)). Since the number of VP on a single node is usually larger than the number of cores in an HPC environment, we can use a simple strategy to schedule an available virtual port for each MPI process. For example, the virtual ports can be scheduled in a round-robin manner by excluding the reserved ports. In this way, the RM can organize the scheduling results into several mappings from GR to VP_NUM and passes them to each process. This process is illustrated in Fig. 7 (a), which visualizes the VP scheduling of an MPI application with a total process count of 8 and a PPN of 4 on a shared mode node. In this example, the same MPI application will most likely occupy a different port range on each compute node.

When a compute node is used exclusive by a single user program, we can adopt a more aggressive mapping scheme by first converting GR to Local Rank (LR), where the identification of processes belonging to the same application inside the compute node (see Fig. 5 (d)). We then use LR plus the number of reserved virtual ports to get VP_NUM (see Fig. 5 (e)). This mapping is computed by the RM before the process is instantiated and passed to the process via PMI wireup. As shown in Fig. 7 (b), this mapping scheme allows processes from the same application to use the same virtual port range on each node, which facilitates debugging for large-scale applications.

In summary, the location-aware address generation rule makes use of process location information (specifically, global scheduling information) that is reachable by the RM. It requires the RM to prepare the mapping information needed for address generation and store it in an agreed-upon way before the processes are instantiated. Then after instantiation, each process first generates its communication entity according to the address generation rule and then uses the mapping results to assemble the communication entities of each remote process locally. The above technique eliminates the need for inter-process address exchange, thus improving the efficiency and scalability of the launch process. Implementing the address generation method requires the support of the interconnection network and the system software stack. Although the hardware information, topological structure, and address format differ across different high-performance interconnection networks, the idea of using the global scheduling information to generate the endpoint addresses of the remote processes locally is general on various platforms. For example, for Infiniband or RoCE networks with UCX [24], the fields of the communication entity have the same meaning, and only the final endpoint address is network specific. Therefore, for HPC systems using the same system software stack (including UCX, RM, and MPI library), the need for global address exchange can be eliminated by designing location-aware address generation rules based on specific network characteristics. Therefore, it is worth looking forward to implementing a non-exchange application launch sequence based on different high-performance interconnection networks.

3.2 Topology-Aware Global Barrier Operation

In the traditional MPI application launch sequence, the PMI exchange is followed by a global barrier operation to confirm that all processes are ready for communication. Regardless of the address generation rules, the global barrier operation still makes sense.

Recently, the PMIx community proposed that increasing the processing of unexpected requests (i.e., communication requests received before the complete initialization of the communication channel) is a feasible means to help the launch sequence get free from the global barrier operation [25]. However, the commonly used approaches to handling unexpected requests are based on timed reconnection, which leads to performance degradation in the early stages of the application's operation. In addition, adding handling of unexpected requests requires modifications to the high-performance interconnect network, which deviates from our original intent of achieving optimization of MPI launch by modifying the system software stack only. Based on these

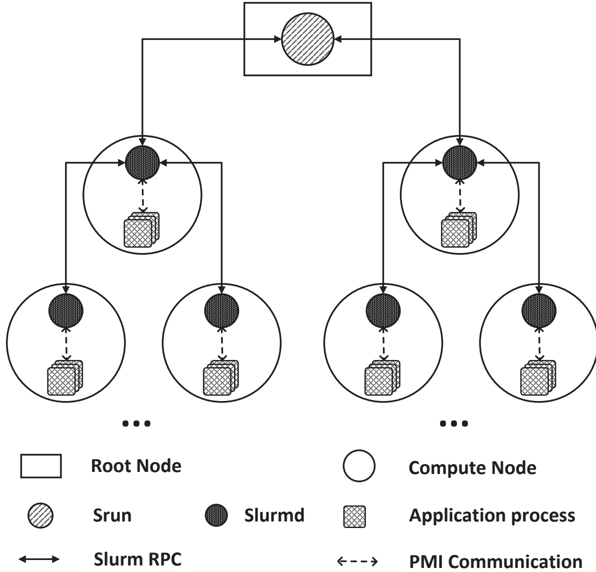


Fig. 8. The schematic diagram of a k-nomial tree.

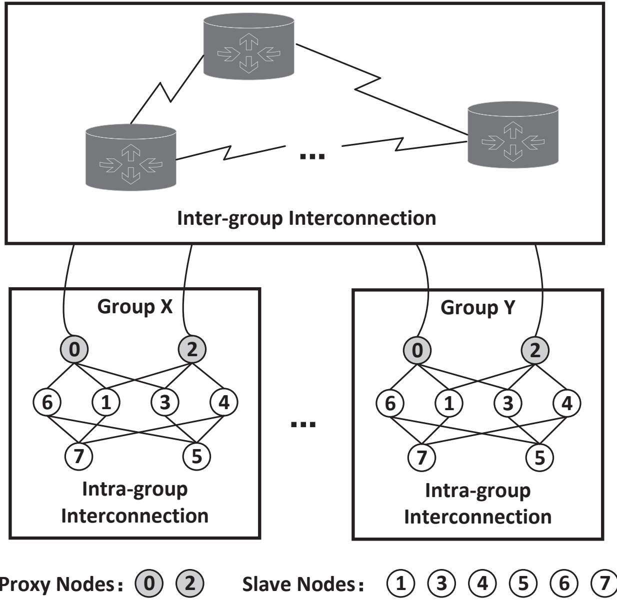


Fig. 9. The TH-Express interconnection topology.

considerations, we choose to keep the barrier operation and strive to improve its efficiency.

3.2.1 Traditional out-of-band global barrier

The global barrier operation is still a preamble of communication over the interconnection network, so it is usually executed across the out-of-band management Ethernet. In the Tianhe HPC system, Slurm (Simple Linux Utility for Resource Management) [26], an open-source, fault-tolerant, and highly scalable RM for large and small Linux clusters, is the actual bearer of the barrier operation. Slurm uses a slurmctld daemon running on the control node to manage, schedule, and allocate resources, and a slurmd daemon running on each compute node to start and clear processes, redirect I/O, and so on. When a job is scheduled, slurmctld

TABLE 1
Time consumed by intra-group communication

Initiator	Recipient	#Forwarding	Time
Group X-0	Group X-3	1	0.032ms
Group X-0	Group X-5	2	0.036ms

TABLE 2
Time consumed by inter-group communication

Initiator	Recipient	#Forwarding	Time
Group X-0	Group Y-0	0	0.047ms
Group X-0	Group Y-3	1	0.049ms
Group X-0	Group Y-5	2	0.053ms
Group X-3	Group Y-5	3	0.057ms
Group X-5	Group Y-5	4	0.061ms

builds a k-nomial tree between slurmds to support the launch of the parallel application. Fig. 8 shows a k-nomial tree based on Slurm RPC (Remote Process Call) and PMI wireup.

As noted in Section 2.3, in the traditional application launch sequence with the PMI library, the barrier operation is integrated into the barrier-based Fence operation. In the Tianhe HPC systems and other supercomputers using Slurm, the barrier-based Fence operation is done based on Slurm's k-nomial tree.

Specifically, the barrier-based Fence operation consists of two steps, including allgather and broadcast. First, the processes of each compute node send local data to the local slurmd daemon through PMI wireup, and the information collected by each slurmd is sent to the root node from the bottom to the top layer by layer to complete the allgather operation. The root node then broadcasts the aggregated data to all slurmds, which finally reaches each process. The allgather and broadcast operations are performed based on Slurm's k-nomial tree. As the scale of the MPI application increases, the inter-node communication contained in the k-nomial tree also increases substantially. In addition, this hierarchical communication scheme in Slurm is used for other steps of application launch, such as file caching (see Section 3.3).

Based on the general consideration of the RM, the out-of-band global barrier operation done by the uncustomized RM cannot take the topology of the interconnected network into account. As a result, the structure of the traditional k-nomial tree used in Slurm is completely separated from the topology of the interconnected network, leading to inefficient communication. Herein, we are committed to proposing a topology-aware tree that can rationalize the communication between nodes based on the network topology information and improve global communication efficiency.

3.2.2 The topology of interconnection networks

TH-Express uses the network topology based on a hybrid optoelectronic interconnection. Fig. 9 illustrates a classical TH-Express topology. Each node group consists of eight compute nodes. Nodes in the same node group perform intra-group communication. Only two of the eight nodes in each node group are used as proxy nodes to participate in inter-group communication. A proxy node is a node that is directly connected to a communication link between groups.

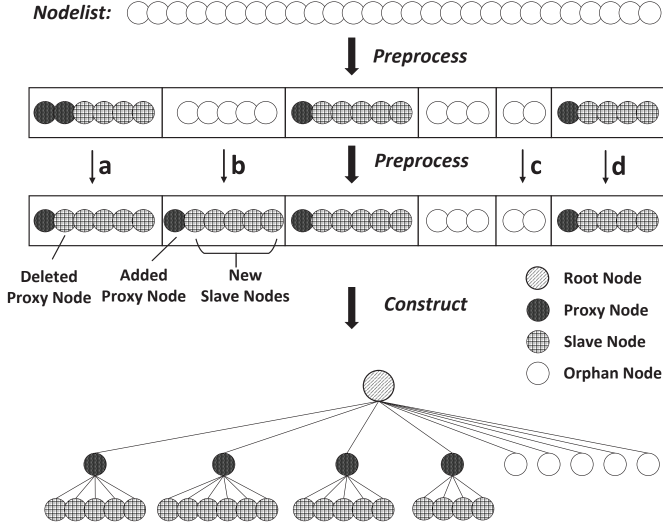


Fig. 10. The construction process of a TA-tree.

The non-proxy nodes in each group are called slave nodes. Data from another group to the slave nodes of this group must be forwarded through the NIC of one of the proxy nodes of this group. Nodes in a node group usually have a tighter physical structure with each other, e.g., located on the same physical board. Therefore, inter-group communication tends to be more time-consuming compared to intra-group communication. We evaluate the average time consumed by sending ICMP ping packets between different combinations of nodes on the Next Generation Tianhe Supercomputer. As shown in Table 1 and Table 2, the time consumption by inter-group communication is larger than that of intra-group communication. In addition, an increase in the number of intra-group forwards causes an increase in communication time.

We design a topology-aware tree structure (TA-tree) to minimize the number of inter-group communications and the total length of intra-group forwarding length. The priority of reducing inter-group communication is higher than reducing the number of intra-group forwarding. Therefore, we build a topology-aware tree that can first reduce the number of inter-group communications and then reduce the total length of intra-group forwarding paths.

Although the above analysis is based on the Tianhe HPC systems, the interconnection network topology with inter-group and intra-group communications co-exist is adopted by many HPC systems. In the TofuD interconnection network [16], the hypercube topological structure composed of 12 compute nodes is called a tofu unit. The nodes in the same unit perform intra-unit interconnection, while inter-unit interconnection is performed between tofu units. In Sunway Taihulight [27], every 256 compute nodes constitutes a supernode. The communication within the supernode and the communication between supernodes form a multi-level Sunway network. In the interconnect design of Summit [28], there are 18 compute nodes in a calculation frame. The nodes in the same frame are connected to a separate TOR switch. Then the TOR switch on each calculation frame is then connected to the core switch. Based on the above facts, we believe that the above idea of reducing the number of inter-group communications and intra-group forwarding has some generality. In this case, the TA-tree can be tuned to accommodate different interconnection network topologies, for example, by setting each local root node as some network facilities, such as switches.

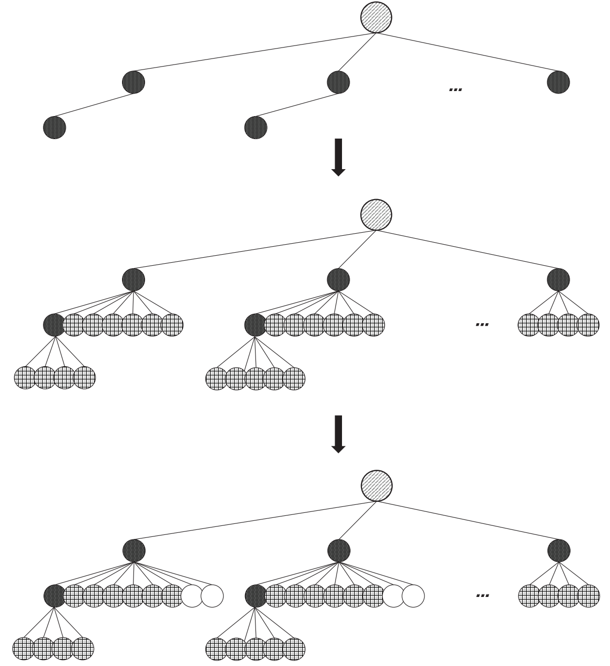


Fig. 11. The tree construction stage of a TA-tree with 4 layers and a tree width of 9.

3.2.3 Constructing topology-aware tree

Our TA-tree is constructed based on two principles. The first is to collect nodes in the same group as much as possible to form a local tree structure. The second is that the root node of each local tree structure is preferably the proxy node of that group, and the parent node of each local root node is preferably the proxy node of another group.

For each node, if its parent node belongs to another group, there must be an inter-group communication between the parent node and that node. In comparison, if the parent node is a node in the same group, only intra-group communication exists. Thus, our first principle reduces the number of inter-group communications. If we now consider the local tree structure as a whole. The shortest path between the two groups is the path between the proxy nodes of the two groups. At this time, the communication between the two groups only includes one inter-group communication without any intra-group forwarding, which reduces the number of intra-group forwarding to the greatest extent.

There are two stages in constructing a TA-tree, including the pre-processing stage and the tree construction stage. Fig. 10 visually shows the workflow of the two stages. To reduce the complexity, we do not blindly pursue the optimal solution when constructing a TA-tree, but find a better solution within the acceptable complexity range.

Pre-processing stage. After getting the original node list, we put the nodes that belong to the same node group in the network topology into the same node set. We classify nodes into three categories within each node set: proxy

nodes, slave nodes, and orphan nodes. Here, the orphan nodes refer to those slave nodes that do not have any proxy nodes of the same group appearing in the node list. Then, we analyze each obtained node set. For a set that already has two proxy nodes, we reserve a proxy node and modify the other proxy node as a slave node (see Fig. 10 (a)). For a set with more than four orphan nodes, we select a proxy node from the same node group and add it to the node list while modifying all orphan nodes in the same group to slave nodes (see Fig. 10 (b)). The newly added proxy node is not in the original node list. In other words, the newly added proxy node is not the node where the MPI application process is located. Therefore, the newly added proxy node can only perform the data forwarding function after adding to the TA-tree but cannot process the data locally. Specifically, we set the forwarding-only flag on the node newly added to the node list to indicate that the node only forwards the data and does not perform any processing on the data. For the set with less than four orphan nodes and the set with only one proxy node, we do not perform any operations (see Fig. 10 (c) and (d)).

Tree construction stage. We organize the nodes in the same node set into a local tree structure as much as possible. However, for node sets with only orphan nodes, we do not organize them into a local tree structure because a local tree structure formed by a small number of nodes can significantly reduce the concurrency of message transmission, resulting in a decrease in communication efficiency. According to the two principles described in the second paragraph of Section 3.2.3, we use the following construction rules. **R1:** Firstly, each proxy node must be the root node of the local tree structure. The parent node of each proxy node can only be another proxy node or the global root node, and choosing the global root node as the parent node has a higher priority. **R2:** Secondly, any slave node must be a leaf node, and its parent node must be the specified proxy node within the group. **R3:** Thirdly, the parent node of an orphan node can be the global root node, a proxy node, or another orphan node, in descending order of priority. Among these, the first and the second rule meets the requirements of our second and first principles, respectively, while the third rule is created to deal with some exceptional cases.

When constructing the TA-tree, we first determine the location of each local root node, i.e., the specified proxy node within each node set, according to **R1**. Then the slave nodes of each proxy node are added to the tree as the child nodes according to **R2**. Finally, orphan nodes fill the tree structure according to **R3**. Fig. 11 shows the tree construction stage of a TA-tree with more layers and a treewidth of 9. In the above construction process, the number of child nodes of each node cannot exceed the treewidth. Therefore, the treewidth needs to be set larger than the number of nodes in each node group which ensures that a proxy node has enough tree width to connect another proxy or orphan node after connecting all the slave nodes in the group.

3.3 Optimized MPI Application Launch sequence

Fig. 12 compares the traditional launch sequence with our proposed optimized launch sequence. There are four steps in total in the launch sequence, and we first briefly describe the traditional launch sequence.

Stage 1. A user submits a job script describing the required resources to the workload manager (WLM). The WLM allocates resources to the job according to the resource availability, relative priorities, and scheduling algorithm.

Stage 2. WLM broadcasts the job-related information (such as the job script and global scheduling information) to the RM daemons of allocated nodes.

Stage 3. The RM daemon on each node retrieves the executable binary and dependencies in the file system (FS) and instantiates it on the compute node. When multiple local RM daemons simultaneously retrieve the executable binary and the libraries from a shared file system (typically used in an HPC environment), these simultaneous I/O accesses can put enormous pressure on the file and the I/O forwarding layer, leading to a problem known as the library retrieval storm [25].

Stage 4. Each process discovers the local resources and generates an endpoint address for inter-process communication. Then, all processes in the application implement a global information exchange followed by a global barrier operation. The exchange body includes at least the global rank and endpoint address. The global barrier confirms that all processes have completed the above work and are ready to communicate. Finally, each process creates an address table to store the address of remote peers obtained by global exchange. This stage is completed by the MPI_Init function in MPI.

Our current implementation of the optimized launch technique on the Tianhe HPC system[1] supports two mainstream parallel programming models, including MPICH, Open MPI. It does not require PMI exchange and uses the TA-tree to optimize global communications. Compared with the optimized launch sequence proposed in [12], our new optimized launch sequence directly uses the initialized PMI key-value store instead of shared memory. This choice eliminates the additional operations of creating, setting, and reclaiming shared memory, thus reducing the launch time. In the launch sequence, PMI is a generic term for both PMI-2 and PMIx. We have now implemented the optimized launch sequence on PMI-2 and PMIx.

Stage 1. Same as the traditional launch sequence.

Stage 2. WLM retrieves executables and dependencies from the FS and caches the required files locally. At the same time, WLM constructs a TA-tree containing all the compute nodes assigned to the job. Finally, WLM broadcasts the required files and job-related information to the RM daemons of allocated nodes through the TA-tree. The Tianhe HPC system takes a more direct approach by caching the required files directly to the local memory of each compute node. Within this stage, the added file caching process occurs at the same time as the WLM broadcasts job-related information to the compute nodes, the latter being a mandatory step in any launch sequence. Thus, the additional file caching process can be seen as enabling the WLM to broadcast more

1. Specifically, in the Tianhe HPC system, WLM and the RM daemon are implemented as the slurmctld and the slurmd daemon using the Slurm cluster management system.

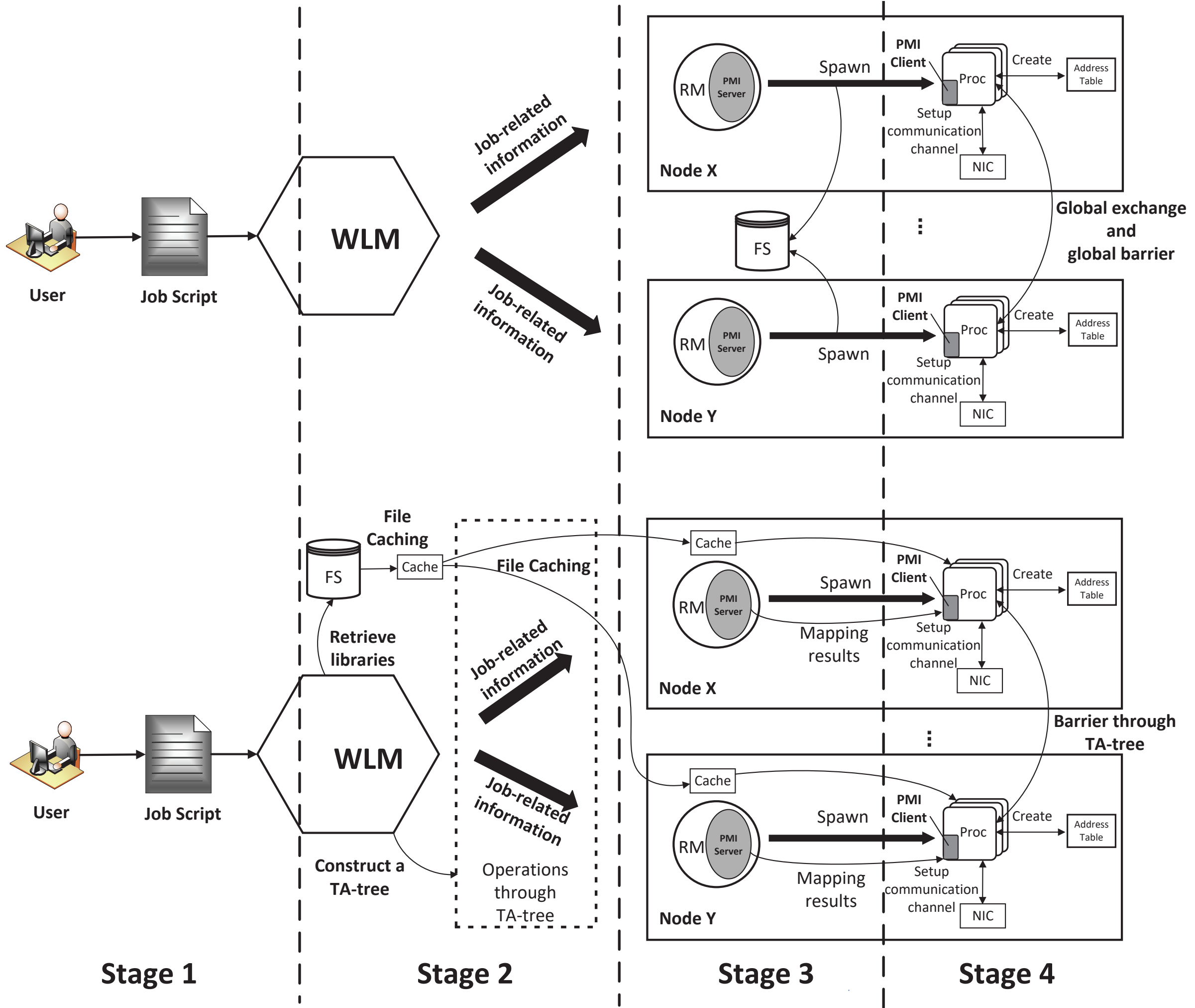


Fig. 12. The MPI application launch sequence. The above figure shows the traditional launch sequence, while the below figure shows the optimized launch sequence.

relevant information to the compute nodes and replacing the traditional tree structure used with the TA-tree compared to the traditional launch sequence. Due to the rational arrangement of the process and the superior performance of the TA-tree, the file caching time can be significantly hidden.

Stage 3. The RM daemon on each compute node uses the global scheduling information and network hardware information to map the global rank to NIC_ID and VP_NUM. Next, each RM daemon stores the mapping result in the PMI key-value store with the global rank as the key and the NIC_ID or VP_NUM as the value through the PMI wireup. With this accomplished, each RM daemon instantiates the application processes.

Stage 4. At this stage, each MPI process establishes a connection to the PMI server and gets mapping results prepared by RM. Then the process creates its own communication channel on the interconnection network. Then, an out-of-band barrier is executed through TA-tree to ensure that all processes are ready to communicate. Finally, each process uses the mapping results to assemble the addresses of the remote peers locally according to the proposed address generation rule. The addresses of all processes are stored in the process address table. This stage is completed by the MPI_Init function in MPI.

TABLE 3
System software stack used in evaluation

	System Software
File System	Lustre v2.14.0
Resource Manager	Slurm v20.11.7
Process Management Interface	PMI-2 / PMIx v3.2.3
Parallel Programming Model	MPICH v3.4.2 / Open MPI v4.1.1
Common Communication Interface	UCX v1.11.0

Furthermore, compared with the traditional launch sequence, our optimized launch sequence is designed to speed up the **Stage 4**. It introduces additional time and space overhead in **Stages 2** and **Stage 3**, which we will evaluate in the next section. Thereafter, we refer the launch time to the time between **Stages 2** and **Stage 4**. That is, from the time when the job is being allocated computing resources to the end of MPI_init function.

4 Evalutaion Setup

4.1 Hardware Platforms

We evaluate our approach on two HPC systems. The first is the Tianhe-2A supercomputer, ranked in the seventh

TABLE 4
PMI and MPI library settings used in each figure.

Figure	PMI	MPI Library
Fig. 13	PMIx v3.2.3	Open MPI v4.1.1
Fig. 15	PMIx v3.2.3	Open MPI v4.1.1
Fig. 16	PMIx v3.2.3	MPICH v3.4.2
Fig. 17	PMI-2	MPICH v3.4.2
Fig. 18	PMI-2	MPICH v3.4.2
Fig. 19	PMI-2	MPICH v3.4.2
Table 5	PMI-2	MPICH v3.4.2
Table 7	PMI-2	MPICH v3.4.2

place on the TOP500 list as of November 2021. The second platform is The Next Generation Tianhe Supercomputer. Tianhe-2A has 16K compute nodes, with a total of 4,981,760 cores and 2,277 TB of memory. Each computing node on Tianhe-2A has 64GB of RAM with a 12-core 2.2 GHz Intel Xeon processor and a Matrix-2000 accelerator. The Next Generation Tianhe Supercomputer has more than 20K computing nodes. Each computing node on the Next Generation Tianhe Supercomputer has a heterogeneous many-core MT processor. Both HPC systems are equipped with the in-house designed interconnection network TH-Express. This proprietary interconnection network interface chip is designed to provide high-speed network interconnection. A single network port uses a four-lane high-speed serial transmission link, providing a link rate up to 25 Gbps. It provides a one-port one-way bandwidth of 100 Gbps, and the external one-way bandwidth of a single compute node is 400 Gbps in total. The CPU and the network interface chip are connected through the PCIe Gen3 16x interface.

4.2 System Software Stack

Table 4 illustrates the system software stack used in the evaluation, where the resource manager Slurm, the parallel programming model MPICH and Open MPI, and the common communication interface UCX are revised to support our optimized launch sequence. Our changes are described as follows.

Slurm. We update Slurm to read the network hardware information from a configuration file at startup. It uses TA-tree instead of the traditional tree structure in the MPI application launch sequence. It retrieves files from the file system in advance and caches them to compute nodes via TA-tree. It then generates mappings from Global_Rank to NIC_ID and VP_NUM at each compute node and store the mapping results in the shared memory and store the pointer of shared memory in PMI key-value store.

MPICH and Open MPI. We update the MPI runtime to obtain the pointer of shared memory through PMI wireup and b) uses the mapping results in the shared memory to assemble the communication address of the remote peers instead of using the Put-Fence-Get sequence to exchange the communication address of the remote peers.

UCX. We revised UCX to generate the communication address of the local process according to the new address generation rule.

4.3 Experimental Roadmap

Our evaluation is divided into six parts. First, we evaluate the effect of file caching on mitigating library retrieval storms (Section 5.1). Next, we evaluate the data transfer performance of the TA-tree (Section 5.2). Then, we evaluate the acceleration effect of the optimized launch sequence on MPI_Init (Section 5.3). We then show the additional cost of the optimized launch sequence compared to the traditional launch sequence (Section 5.4). Again, we compare the total time consumed by launching the benchmark program and the real HPC workload using the traditional and optimized launch sequences (Section 5.5). Finally, we study the total runtime and results of the benchmark programs to evaluate the potential impact of the proposed optimized launch technique on MPI applications in terms of correctness and performance (Section 5.6).

It is worth mentioning that the file caching process is integrated into the system software stack in the optimized launch sequence shown in Fig. 12. Still, it is also possible to manually copy the required files to compute nodes before the application starts. Since there are various implementations of file caching and it is closely related to the underlying network implementation of the HPC system, we do not consider the acceleration effect from file caching as the main optimization in this paper. Therefore, in the following Section 5.3 and Section 5.5, the baseline approach refers to the launch sequence with the file caching technique added to the traditional launch sequence. Therefore, the optimization effects in Section 5.3 and Section 5.5 are brought about by the location-aware address generation rule and topology-aware global barrier operation, independent of the file caching.

To evaluate the optimization effect of the optimized launch sequence, we compare the following five launch techniques on the Tianhe-2A in the evaluation.

Baseline. The baseline method represents the traditional application launch method using the PMI library.

Shm. Shm method represents the method of using shared memory to reduce the data retrieved from PMI during PMI exchange, which is proposed in [9].

Ring. Ring method refers to the method of using the ring exchange operation proposed in [10] to replace the Fence operation.

Allgather. Allgather method is a new API that combines three separate operations of Put, Fence, and Get as a single collective call, which is proposed in [11].

LATA. LATA method refers to the use of the optimized launch techniques and its corresponding optimized launch sequence proposed in this paper. LATA is taken from the initials of the words Location-Aware and Topology-Aware.

5 Experimental Results

5.1 Optimization Effect of File Caching

Caching files needed by the MPI application to the compute nodes can avoid large retrievals from the file system in a short time. Fig. 13 compares the launch time of MPI Hello_World program of different sizes with and without file caching on the Tianhe-2A. Note that in this figure, the use and non-use of file caching are based on the traditional launch sequence, which means that the figure only compares the optimization effect brought by file caching. The launch time is defined as the time from when the resource manager starts scheduling this MPI application until all

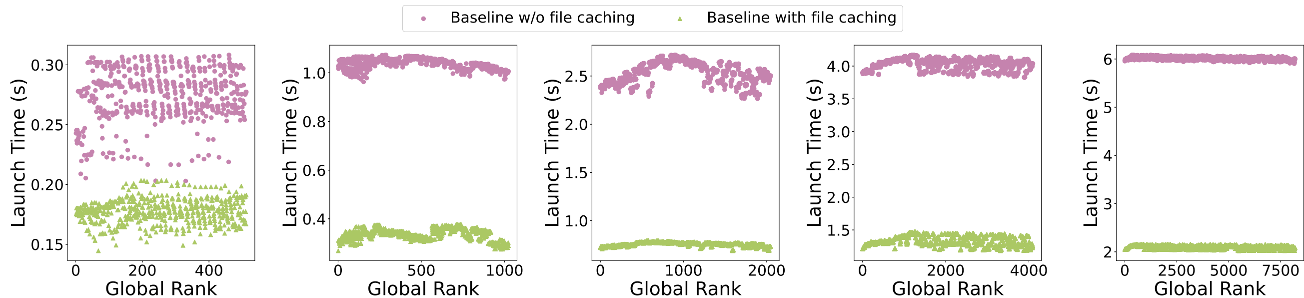


Fig. 13. MPI application launch time for different sizes with and without file caching on the Tianhe-2A. From left to right the total number of processes for MPI applications is 512, 1024, 2048, 4096, 8192, where PPN=16.

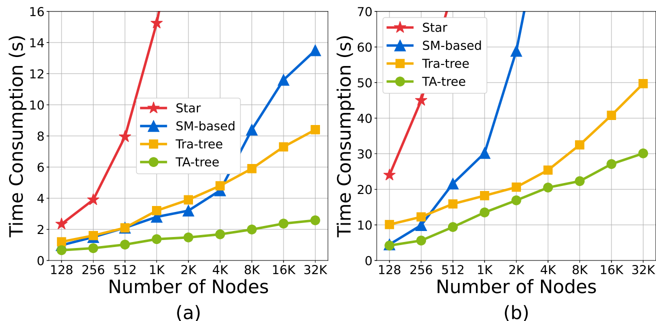


Fig. 14. Time consumed to transfer 16MB (figure a) and 400MB (figure b) of data on the Next Generation Tianhe Supercomputer using different data transfer methods.

processes exit the MPI_Init function. Referring to Fig. 12, the launch time is the time from the start of **Stage 2** to the end of **Stage 4**. In Fig. 13, as the application size increases, the launch time without file caching increases significantly, while using file caching can significantly reduce the increase. This result fully illustrates library retrieval storms' impact on launch time.

5.2 TA-tree Performance

We evaluate the performance of the TA-tree in **Stage 2** (see Section 3.3) on the Next Generation Tianhe Supercomputer. We compare four data transfer methods:

Star. This method refers to the use of a star structure to transfer data directly from one location to multiple nodes. Some tools based on ssh connections can be equated to the Star method, such as the clustershell tool [29].

SM-based. This method caches the first request to shared memory or a burst buffer and serves all subsequent requests from the cache (e.g., SPINDLE [30]).

Tra-tree. This strategy transfers data using the traditional tree structure. A good case in point is the sbcast command of Slurm.

TA-tree. This scheme transfers data using our proposed topology-aware tree structure.

Fig. 14 shows the time consumption of transferring files with a total size of 16MB and 400MB using different transfer methods. As shown in Fig. 14, using the TA-tree consumes the least time for different node sizes, which fully demonstrates that the TA-tree has a good optimization effect with better scalability. These features of the TA-tree are significant for speeding up the launch of parallel applications.

5.3 Time Consumed by MPI_Init Function

The optimized launch sequence mainly speeds up **Stage 4** (see Section 3.3), the MPI_Init function. Fig. 15 shows the time consumed by Open MPI's MPI_Init when launching the MPI Hello_World program of different sizes on the Tianhe-2A. As the total number of processes in the program increases, **LATA** accelerates MPI_Init more significantly. When starting a program with a total number of 8K processes, **LATA** can reduce the Open MPI's MPI_Init time by 54.6%. A similar trend can be seen in Fig. 16, which shows the time consumed by MPICH's MPI_Init when launching the MPI Hello_World program of different sizes on the Tianhe-2A. When starting a program with a total number of 8K processes, **LATA** can reduce the MPICH's MPI_Init time by 28.8%.

Fig. 17 shows the time consumed by MPI_Init when launching the MPI Hello_World program with five different launch techniques. The **Shm**, **Ring**, and **Allgather** reduce the time consumed by MPI_Init to some extent compared to the **Baseline** method with file caching, but the most significant improvement comes from our **LATA** method. When launching an application with 32K processes, our **LATA** method can reduce the time consumption by 22.9% compared to the **Baseline** method with file caching.

To evaluate the scalability of our **LATA** method, we conduct experiments on larger clusters and collect experimental data that is generally lacking in previous related work. We use 10K and 20K nodes in the Next Generation Tianhe supercomputer to evaluate the time consumed by the MPI_Init function of the MPI Hello_World program while the PPN grows from 1 to 16. As the scale of the parallel application increases, the optimization effect of our **LATA** method becomes more pronounced. When launching a parallel application with 160K processes on 10K nodes, the **LATA** method reduces the MPI_Init time by 26.6% compared to the **Baseline** method with file caching. When launching an application with 320K processes on 20K nodes, the reduction is 29.7%. Another interesting fact is that launching the same application on more nodes (i.e., reducing PPN) increases the MPI_Init time, regardless of the launch technique used. This is because the global address exchange and the global barrier operation contain a number of communications that are positively correlated with the number of nodes.

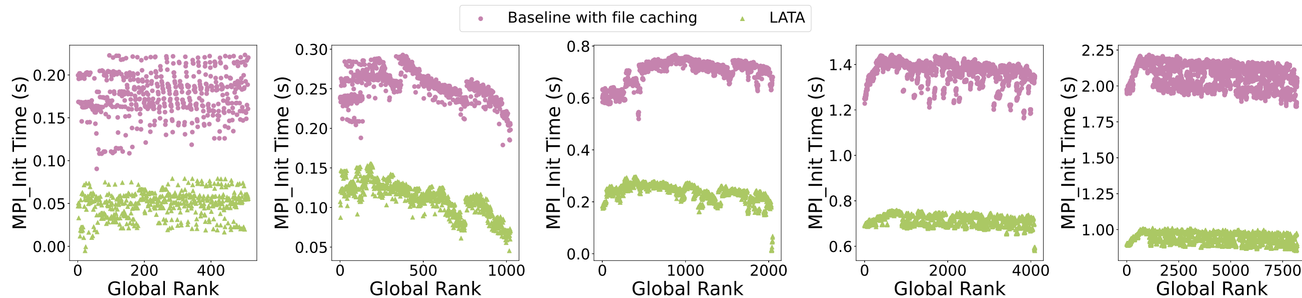


Fig. 15. Time Consumed by Open MPI's MPI_Init when launching Hello_World program of different sizes on the Tianhe-2A. From left to right the total number of processes for MPI applications is 512, 1024, 2048, 4096, 8192, where PPN=16.

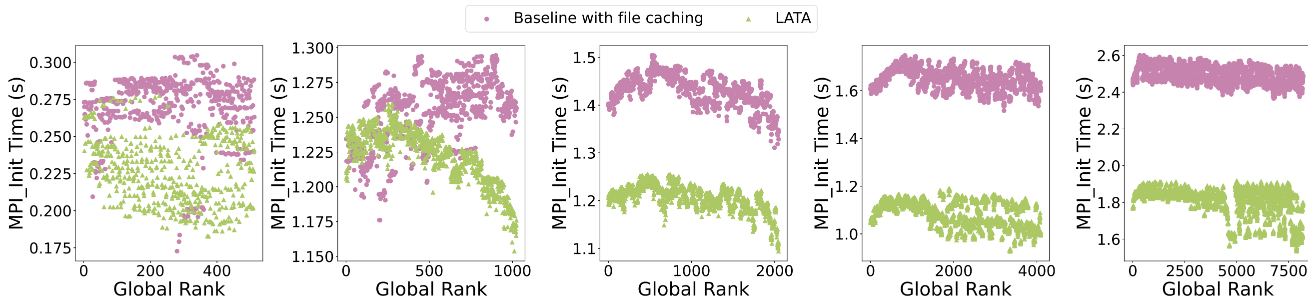


Fig. 16. Time Consumed by MPICH's MPI_Init when launching Hello_World program of different sizes on the Tianhe-2A. From left to right the total number of processes for MPI applications is 512, 1024, 2048, 4096, 8192, where PPN=16.

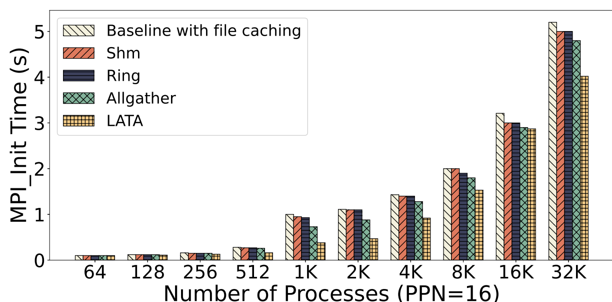


Fig. 17. Time consumed by MPI_Init with different launch techniques on the Tianhe-2A.

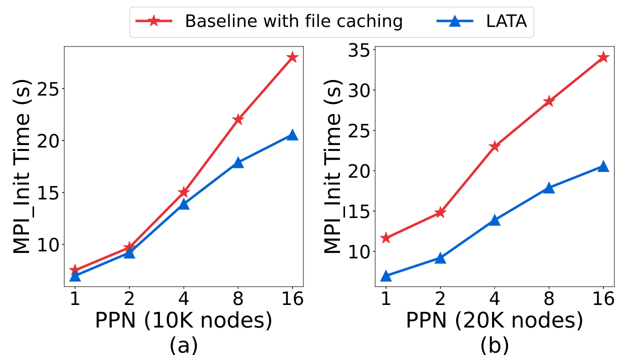


Fig. 18. Time consumed by MPI_Init on the Next Generation Tianhe Supercomputer.

TABLE 5
Additional overhead

# Proc	Additional time overhead	Additional PMI key-value store
64	0.165s	1KB
256	0.169s	4KB
1,024	0.175s	16KB
4,096	0.182s	64KB
16,384	0.194s	256KB
65,536	0.205s	1MB
262,144	0.224s	4MB
163,840	0.218s	2.56MB
327,680	0.230s	5.12MB

TABLE 6
Workload used in the evaluation

Name	Category	# Proc	PPN
CG	NPB benchmark	8,192	16
EP	NPB benchmark	8,192	16
FT	NPB benchmark	4,096	16
MG	NPB benchmark	4,096	16
GROMACS [31]	GROningen MAchine for Chemical Simulation	2048	16
QUEST [32]	QUantum mechanics Enabled Simulation Toolset	2,048	4
Kmeans [33]	Clustering Algorithm	1,024	4
CSEM [34]	Community Earth System Model	1,024	16
WRF [35]	The Weather Research and Forecasting model	1,024	16

5.4 Additional Overhead

To implement the optimized launch sequence, the RM in the **Stage2** and **Stage 3** (see Fig. 12) adds additional workflows and put more key-value pairs into the PMI key-value store on each node. Table 5 shows this additional overhead for launching MPI applications of different sizes. The increase in additional time overhead with application size is small. When launching an MPI application with the optimized

TABLE 7
Total runtime of NPB benchmark programs on the Next Generation Tianhe Supercomputer

Program	Scale	#Proc	PPN	Launch Technique	Total Runtime (s)	Verification	Program	Scale	#Proc	PPN	Launch Technique	Total Runtime (s)	Verification
CG	B	4096	8	Baseline	7.13	Successful	IS	B	1024	8	Baseline	1.87	Successful
				LATA	6.03	Successful					LATA	1.86	Successful
	D	4096	8	Baseline	64.23	Successful		D	1024	8	Baseline	8.45	Successful
				LATA	62.83	Successful					LATA	8.76	Successful
EP	B	4096	8	Baseline	3.26	Successful	BT	B	1024	8	Baseline	4.22	Successful
				LATA	3.15	Successful					LATA	4.05	Successful
	D	4096	8	Baseline	4.06	Successful		D	1024	8	Baseline	103.16	Successful
				LATA	3.96	Successful					LATA	103.88	Successful
FT	B	4096	8	Baseline	3.66	Successful	SP	B	1024	8	Baseline	6.62	Successful
				LATA	3.49	Successful					LATA	6.39	Successful
	D	4096	8	Baseline	23.49	Successful		D	1024	8	Baseline	186.48	Successful
				LATA	22.36	Successful					LATA	187.27	Successful
MG	B	4096	8	Baseline	3.25	Successful	LU	B	1024	8	Baseline	3.87	Successful
				LATA	3.16	Successful					LATA	3.91	Successful
	D	4096	8	Baseline	7.46	Successful		D	1024	8	Baseline	76.26	Successful
				LATA	7.14	Successful					LATA	76.60	Successful

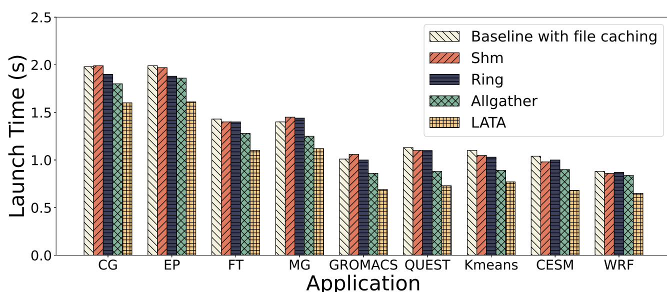


Fig. 19. Launch time with different launch techniques on the Tianhe-2A.

launch sequence, the additional time overhead of the RM is only 0.16-0.23s. Compared to the reduction of MPI_Init time by the optimized launch sequence (see Fig. 17), the additional time overhead is perfectly acceptable at larger application sizes. Therefore, we recommend using the optimized launch sequence for large-scale applications with more than 1K processes while still using the traditional launch sequence for smaller applications. The last two rows in Table 5 correspond to the largest-scale experimental scenarios in Fig. 18 (a) and (b), respectively. Specifically, when an application with 160K processes is started on 10K nodes, the additional time overhead brought by using the optimized launch sequence is 0.78% of the traditional launch sequence's launch time. When launching an application with 320K processes on 20K nodes, this ratio is only 0.48%. On the other hand, the size of shared memory on each node appears to increase linearly with the size of the system. When launching an application with 256K processes, only 4MB of additional PMI key-value store is required on each node.

5.5 Application Launch Time

To evaluate the advantages of the optimized launch sequence over traditional launch sequences in real production environments, we run the workloads in Table 6 to evaluate the launch time. At this time, the launch time includes the time from **Stage 2** to **Stage 4**, so the additional time overhead of the RM in **Stage 2** and **Stage 3** is also taken into account. We make sure that the same application runs on the same set of nodes during the experiment to avoid the performance gap of compute nodes from affecting the experimental result. Fig. 19 shows the experimental results on the Tianhe-2A. Compared with the other four launch techniques, our **LATA** method can minimize the launch time of each application.

5.6 Total Runtime

To evaluate the potential impact of **LATA** on MPI applications in terms of correctness and performance, we launched NPB benchmark programs of different sizes using both **LATA** and **Baseline** methods. The programs' total runtime and running results are shown in Table 7. The total runtime refers to the time from the program submission to RM to the completion of the program running when the resources are sufficient. The running result is the verification result (SUCCESSFUL or FAILED) output by the NPB benchmark programs.

As can be seen from Table 7, all NPB benchmark programs run correctly, suggesting that launching the MPI application with **LATA** does not affect the correctness of the programs. When the total number of processes of the MPI application is 4K, using **LATA** to start the program reduces the total runtime of the program compared to the **Baseline** method. However, when the total number of processes in the MPI application is only 1K, this advantage does not exist. Moreover, looking at all the experimental results, we find that the advantage of **LATA** is more pronounced when the size of the MPI application is larger. This is because the objects of our optimization, i.e., global address exchange and global barrier operation, only become bottlenecks when the number of processes is high. Therefore, our optimization is mainly intended for launching large-scale applications (at least more than 4K processes). In practice, we also recommend using **LATA** launching technology only when launching large-scale applications.

6 Related Work

In 2018, the PMIx community proposed an application launch sequence without collective address exchange and

the global barrier operation [25]. However, the PMIx community is still working with network vendors to field the support of the new launch sequence, which requires modifications to the fabric manager and NIC libraries, as well as potential hardware changes for optimized performance. Since the above work is incomplete, the latest MPI versions, such as MPICH v4.0.1, still retain the barrier-based PMI exchange. In 2020, the PMIx community implemented an alternative wireup method known as Direct Modex [36], through which data can be retrieved from a remote host without the need for barrier-based information exchange. However, the Direct Modex reduces the efficiency of each retrieve, so it remains best suited for sparsely connected applications. Holmes et al. [37] proposed the MPI Session in 2016, and the MPI Forum added the MPI Session extensions to MPI-4.0 [38] in 2021. The session model describes an alternative approach to MPI initialization, which can instantiate MPI resources for specific communication needs. The MPI Session can address limitations, including MPI cannot be initialized from different application components without prior knowledge or coordination, MPI cannot be initialized more than once, and MPI cannot be reinitialized after the MPI_Finalize function has been called. However, the MPI Sessions approach has some overhead (20%) compared to that used for MPI initialization with Open MPI release [39].

There have been significant efforts to improve the performance and scalability of launching parallel applications. Yu et al. [40] explored a method of using InfiniBand to reduce the startup costs of MPI jobs. Chakraborty et al. [10] proposed three extensions to the PMI specification: a ring exchange collective, a broadcast hint to Put operation, and an enhanced Get operation to reduce communication over the out-of-band PMI channel. [11] designed and implemented an Allgather API to reduce the data processing overheads by cutting down the amount of data being transferred in existing PMI designs. Polyakov et al. [41] made specific improvements in the UCX endpoint address format, the layout of PMIx metadata, and the use of Little-Endian Base 128 encoding, which decreased the volume of inter-node data exchanged by up to 8.6x. The main idea of these approaches is to reduce the launch costs by reducing the amount of data globally exchanged out-of-band. In contrast, our method eliminates the need for global address exchange, which minimizes the information that needs to be exchanged. Turilli et al. [42] describe the performance of running multitasking applications on the Summit [28]. Although this research evaluates the overhead of resource management and task scheduling in the application launch, it lacks the application launch overhead (i.e., the overhead of the MPI_Init function) embedded in the MPI library.

Related research also focuses on using shared memory to improve the efficiency of application launches. [43] used a shared memory-based channel to reduce the memory consumption when launching applications. [9] presented the use of shared memory to reduce the total amount of information retrieved from PMI and reduce the dependence on PMI by employing the HPC fabric to transfer the bulk of address data. These researches were devoted to solving the memory bottleneck in PMI communication. In fact, PMI communication, including the Put, Get, and Fence operations, mainly occurs during global information exchange. Our launch technique significantly reduces PMI communication by avoiding global exchanges and fundamentally alleviates the memory bottleneck caused by a large amount of PMI communication.

Much work focuses on how to optimize MPI applications using topology-aware techniques. Vardas et al. [44] use topology-aware techniques to allocate optimal resources to MPI applications to reduce communication costs. Tsujita et al. [45] exploit MPI's rank reordering mechanism to achieve runtime topology awareness for collective communication, especially MPI_Allgather to reduce communication latency. Ma et al. [46] proposed the kernel-assisted topology-aware collective framework HierKNEM to coordinate the collaboration among multilayer collective algorithms. This scheme maximizes the overlap of intra- and inter-node communication in collective operations. Rashti et al. [47] used graph embedding and node/network architecture discovery modules to match the communication topology of an application to the physical topology of a multicore multilayer network cluster to improve communication performance. These works focused on using topology-aware techniques to optimize MPI applications' communication performance, particularly to reduce MPI collective operations' communication latency. However, communication at MPI launch is typically borne by resource management systems and has not received sufficient attention.

Some efforts to improve the performance of global communications during application launch deserve attention. Claudel et al. [48] proposed a pipelining mechanism to overlap communication. Gupta et al. [49] proposed an smp-aware multi-level startup scheme with batching of remote shells. Goehner et al. [50] proposed a framework called LIBI, which supports different tree configuration. [41] presented a modification of the Bruck concatenation algorithm that optimizes the tree-based implementations currently used in RMs for PMIx exchange. The above studies effectively improved the performance of out-of-band communications. However, none of them made use of the topological information of the HPC system. Furthermore, these related works lack experimental evaluation on large-scale clusters. Specifically, these works are evaluated on hundreds of nodes, and the optimization effect is only milliseconds.

7 Conclusion

We have presented an approach to accelerate MPI application launch in a distributed HPC environment. Our approach adopts a new communication address generation strategy and a topology-aware global barrier operation. By doing so, we can reduce the frequency of cross-machine communications and the associated overhead during the MPI application launch. We evaluate our approach by applying it to two HPC systems. Experimental results show that our techniques significantly reduce the launch time of large-scale MPI applications, giving up to 29% boost in the MPI launch time.

References

[1] K. Raffenetti, A. Amer, L. Oden, C. Archer, W. Bland, H. Fujita, Y. Guo, T. Janjusic, D. Durnov, M. Blocksome, M. Si, S. Seo, A. Langer, G. Zheng, M. Takagi, P. K. Coffman, J. Jose,

S. Sur, A. Sannikov, S. Oblomov, M. Chuvelev, M. Hatanaka, X. Zhao, P. F. Fischer, T. Rathnayake, M. Otten, M. Min, and P. Balaji, “Why is MPI so slow?: analyzing the fundamental limits in implementing MPI-3.1,” in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2017, Denver, CO, USA, November 12 - 17, 2017*, B. Mohr and P. Raghavan, Eds. ACM, 2017, pp. 62:1–62:12. [Online]. Available: https://doi.org/10.1145/3126908.3126963

[2] C. Wang, P. Balaji, and M. Snir, “Pilgrim: scalable and (near) lossless MPI tracing,” in *SC '21: The International Conference for High Performance Computing, Networking, Storage and Analysis, St. Louis, Missouri, USA, November 14 - 19, 2021*, B. R. de Supinski, M. W. Hall, and T. Gamblin, Eds. ACM, 2021, pp. 52:1–52:14. [Online]. Available: https://doi.org/10.1145/3458817.3476151

[3] I. Laguna, R. J. Marshall, K. Mohror, M. Ruefenacht, A. Skjellum, and N. Sultana, “A large-scale study of MPI usage in open-source HPC applications,” in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2019, Denver, Colorado, USA, November 17-19, 2019*, M. Taufer, P. Balaji, and A. J. Peña, Eds. ACM, 2019, pp. 31:1–31:14. [Online]. Available: https://doi.org/10.1145/3295500.3356176

[4] Q. Zhou, C. Chu, N. S. Kumar, P. Kousha, S. M. Ghazimirsaeed, H. Subramoni, and D. K. Panda, “Designing high-performance MPI libraries with on-the-fly compression for modern GPU clusters*,” in *35th IEEE International Parallel and Distributed Processing Symposium, IPDPS 2021, Portland, OR, USA, May 17-21, 2021*. IEEE, 2021, pp. 444–453. [Online]. Available: https://doi.org/10.1109/IPDPS49936.2021.00053

[5] S. Hunold, A. Bhatele, G. Bosilca, and P. Knees, “Predicting MPI collective communication performance using machine learning,” in *IEEE International Conference on Cluster Computing, CLUSTER 2020, Kobe, Japan, September 14-17, 2020*. IEEE, 2020, pp. 259–269. [Online]. Available: https://doi.org/10.1109/CLUSTER49012.2020.00036

[6] K. Ouyang, M. Si, A. Hori, Z. Chen, and P. Balaji, “CAB-MPI: exploring interprocess work-stealing towards balanced MPI communication,” in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2020, Virtual Event / Atlanta, Georgia, USA, November 9-19, 2020*, C. Cuicchi, I. Qualters, and W. T. Kramer, Eds. IEEE/ACM, 2020, p. 36. [Online]. Available: https://doi.org/10.1109/SC41405.2020.00040

[7] F. Zahn and H. Fröning, “On network locality in mpi-based HPC applications,” in *ICPP 2020: 49th International Conference on Parallel Processing, Edmonton, AB, Canada, August 17-20, 2020*, J. N. Amaral, L. K. John, and X. Shen, Eds. ACM, 2020, pp. 57:1–57:10. [Online]. Available: https://doi.org/10.1145/3404397.3404436

[8] Tianhe-2A, 2021. [Online]. Available: https://www.top500.org/system/177999/

[9] K. Raffenetti, N. Bayyapu, D. Durnov, M. Takagi, and P. Balaji, “Locality-aware pmi usage for efficient mpi startup,” in *2018 IEEE 4th International Conference on Computer and Communications (ICCC)*. IEEE, 2018, pp. 624–628.

[10] S. Chakraborty, H. Subramoni, J. L. Perkins, A. Moody, M. D. Arnold, and D. K. Panda, “PMI extensions for scalable MPI startup,” in *21st European MPI Users' Group Meeting, EuroMPI/ASIA '14, Kyoto, Japan - September 09 - 12, 2014*, J. J. Dongarra, Y. Ishikawa, and A. Hori, Eds. ACM, 2014, p. 21. [Online]. Available: https://doi.org/10.1145/2642769.2642780

[11] S. Chakraborty, H. Subramoni, A. Moody, A. Venkatesh, J. L. Perkins, and D. K. Panda, “Non-blocking PMI extensions for fast MPI startup,” in *15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGrid 2015, Shenzhen, China, May 4-7, 2015*. IEEE Computer Society, 2015, pp. 131–140. [Online]. Available: https://doi.org/10.1109/CCGrid.2015.151

[12] Y. Dai, Y. Dong, M. Xie, K. Lu, R. Wang, M. Shao, and J. Chen, “The fast and scalable mpi application launch of the tianhe hpc system,” in *36th IEEE International Parallel & Distributed Processing Symposium (IPDPS 2022) 2022*. IEEE Computer Society, 2022.

[13] Z. Pang, M. Xie, J. Zhang, Y. Zheng, G. Wang, D. Dong, and G. Suo, “The TH express high performance interconnect networks,” *Frontiers Comput. Sci.*, vol. 8, no. 3, pp. 357–366, 2014. [Online]. Available: https://doi.org/10.1007/s11704-014-3500-9

[14] Infiniband, 2021. [Online]. Available: https://www.infinibandta.org/

[15] Y. Ajima, T. Inoue, S. Hiramoto, S. Ando, M. Maeda, T. Yoshikawa, K. Hosoe, and T. Shimizu, “The tofu interconnect 2,” in *22nd IEEE Annual Symposium on High-Performance Interconnects, HOTI 2014, Mountain View, CA, USA, August 26-28, 2014*. IEEE Computer Society, 2014, pp. 57–62. [Online]. Available: https://doi.org/10.1109/HOTI.2014.21

[16] Y. Ajima, T. Kawashima, T. Okamoto, N. Shida, K. Hirai, T. Shimizu, S. Hiramoto, Y. Ikeda, T. Yoshikawa, K. Uchida, and T. Inoue, “The tofu interconnect D,” in *IEEE International Conference on Cluster Computing, CLUSTER 2018, Belfast, UK, September 10-13, 2018*. IEEE Computer Society, 2018, pp. 646–654. [Online]. Available: https://doi.org/10.1109/CLUSTER.2018.00090

[17] P. Shamis, M. G. Venkata, M. G. Lopez, M. B. Baker, O. R. Hernandez, Y. Itigin, M. Dubman, G. Shainer, R. L. Graham, L. Liss, Y. Shahar, S. Potluri, D. Rossetti, D. Becker, D. Poole, C. Lamb, S. Kumar, C. B. Stunkel, G. Bosilca, and A. Bouteiller, “UCX: an open source framework for HPC network apis and beyond,” in *23rd IEEE Annual Symposium on High-Performance Interconnects, HOTI 2015, Santa Clara, CA, USA, August 26-28, 2015*. IEEE Computer Society, 2015, pp. 40–43. [Online]. Available: https://doi.org/10.1109/HOTI.2015.13

[18] P. Grun, S. Hefty, S. Sur, D. Goodell, R. D. Russell, H. Pritchard, and J. M. Squyres, “A brief introduction to the openfabrics interfaces - A new network API for maximizing high performance application efficiency,” in *23rd IEEE Annual Symposium on High-Performance Interconnects, HOTI 2015, Santa Clara, CA, USA, August 26-28, 2015*. IEEE Computer Society, 2015, pp. 34–39. [Online]. Available: https://doi.org/10.1109/HOTI.2015.19

[19] S. Atchley, D. Dillow, G. M. Shipman, P. Geoffray, J. M. Squyres, G. Bosilca, and R. G. Minnich, “The common communication interface (CCI),” in *IEEE 19th Annual Symposium on High Performance Interconnects, HOTI 2011, Santa Clara, CA, USA, August 24-26, 2011*. IEEE Computer Society, 2011, pp. 51–60. [Online]. Available: https://doi.org/10.1109/HOTI.2011.17

[20] M. Xie, E. Zhou, Y. Dong, and W. Zhang, “Implementation and evaluation of ucx communication interface on th-express interconnection,” in *Journal of Computer Applications*, 2019, pp. 113–118.

[21] A. Yoo, M. A. Jette, and M. Grondona, “Slurm: Simple linux utility for resource management,” in *JSSPP*, 2003.

[22] MPICH, 2022. [Online]. Available: https://www.mpich.org/

[23] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall, “Open MPI: Goals, concept, and design of a next generation MPI implementation,” in *Proceedings, 11th European PVM/MPI Users' Group Meeting*, Budapest, Hungary, September 2004, pp. 97–104.

[24] N. Papadopoulou, L. Oden, and P. Balaji, “A performance study of ucx over infiniband,” in *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, 2017, pp. 345–354.

[25] R. H. Castain, J. Hursey, A. Bouteiller, and D. G. Solt, “Pmix: Process management for exascale environments,” *Parallel Comput.*, vol. 79, pp. 9–29, 2018. [Online]. Available: https://doi.org/10.1016/j.parco.2018.08.002

[26] schedmd, 2021. [Online]. Available: https://slurm.schedmd.com/

[27] “The sunway taihu light supercomputer:system and applications,” *Science China(Information Sciences)*, vol. 07, no. v.59, pp. 113–128, 2016.

[28] OLCF, “Summit,” 2022. [Online]. Available: https://www.olcf.ornl.gov/summit/

[29] S. Thiell, A. Degrémont, H. Doreau, and A. Cedeyn, “Clustershell, a scalable execution framework for parallel tasks.”

[30] W. Frings, D. H. Ahn, M. P. LeGendre, T. Gamblin, B. R. de Supinski, and F. Wolf, “Massively parallel loading,” in *International Conference on Supercomputing, ICS'13, Eugene, OR, USA - June 10 - 14, 2013*, A. D. Malony, M. Nemirovsky, and S. P. Midkiff, Eds. ACM, 2013, pp. 389–398. [Online]. Available: https://doi.org/10.1145/2464996.2465020

[31] S. Pronk, S. Páll, R. Schulz, P. Larsson, P. Bjelkmar, R. Apostolov, M. R. Shirts, J. C. Smith, P. M. Kasson, D. van der Spoel, B. Hess, and E. Lindahl, “GROMACS 4.5: a high-throughput and highly parallel open source molecular simulation toolkit,” *Bioinformatics*, vol. 29, no. 7, pp. 845–854, 02 2013. [Online]. Available: https://doi.org/10.1093/bioinformatics/btt055

[32] T. Jones, A. Brown, I. Bush, and S. C. Benjamin, “QuEST and high performance simulation of quantum computers,” *Sci. Rep.*, vol. 9, no. 1, p. 10736, Jul. 2019.

[33] S. P. Lloyd, "Least squares quantization in PCM," *IEEE Trans. Inf. Theory*, vol. 28, no. 2, pp. 129–136, 1982. [Online]. Available: https://doi.org/10.1109/TIT.1982.1056489

[34] G. Danabasoglu, J.-F. Lamarque, J. Bacmeister, D. A. Bailey, A. K. DuVivier, J. Edwards, L. K. Emmons, J. Fasullo, R. Garcia, A. Gettelman, C. Hannay, M. M. Holland, W. G. Large, P. H. Lauritzen, D. M. Lawrence, J. T. M. Lenaerts, K. Lindsay, W. H. Lipscomb, M. J. Mills, R. Neale, K. W. Oleson, B. Otto-Bliesner, A. S. Phillips, W. Sacks, S. Tilmes, L. van Kampenhout, M. Vertenstein, A. Bertini, J. Dennis, C. Deser, C. Fischer, B. Fox-Kemper, J. E. Kay, D. Kinnison, P. J. Kushner, V. E. Larson, M. C. Long, S. Mickelson, J. K. Moore, E. Nienhouse, L. Polvani, P. J. Rasch, and W. G. Strand, "The community earth system model version 2 (cesm2)," *Journal of Advances in Modeling Earth Systems*, vol. 12, no. 2, p. e2019MS001916, 2020, e2019MS001916 2019MS001916. [Online]. Available: https://agupubs.onlinelibrary.wiley.com/doi/abs/10.1029/2019MS001916

[35] NCAR, "Wrf," 2022. [Online]. Available: https://www.mmm.ucar.edu/weather-research-and-forecasting-model

[36] pmix-standard v4.0, 2020. [Online]. Available: https://pmix.github.io/uploads/2020/12/pmix-standard-v4.0.pdf

[37] D. Holmes, K. Mohror, R. Grant, A. Skjellum, M. Schulz, W. Bland, and J. Squyres, "Mpi sessions: Leveraging runtime infrastructure to increase scalability of applications at exascale," 09 2016, pp. 121–129.

[38] "Mpi-4," 2021. [Online]. Available: https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf

[39] N. Hjelm, H. Pritchard, S. K. Gutiérrez, D. J. Holmes, R. Castain, and A. Skjellum, "Mpi sessions: Evaluation of an implementation in open mpi," in *2019 IEEE International Conference on Cluster Computing (CLUSTER)*, 2019, pp. 1–11.

[40] W. Yu, J. Wu, and D. K. Panda, "Fast and scalable startup of MPI programs in infiniband clusters," in *High Performance Computing - HiPC 2004, 11th International Conference, Bangalore, India, December 19-22, 2004, Proceedings*, ser. Lecture Notes in Computer Science, L. Bougé and V. K. Prasanna, Eds., vol. 3296. Springer, 2004, pp. 440–449. [Online]. Available: https://doi.org/10.1007/978-3-540-30474-6_47

[41] A. Y. Polyakov, B. I. Karasev, J. Hursey, J. Ladd, M. Brinskii, and E. Shipunova, "A performance analysis and optimization of pmix-based HPC software stacks," in *Proceedings of the 26th European MPI Users' Group Meeting, EuroMPI 2019, Zürich, Switzerland, September 11-13, 2019*, T. Hoefler and J. L. Träff, Eds. ACM, 2019, pp. 9:1–9:10. [Online]. Available: https://doi.org/10.1145/3343211.3343220

[42] M. Turilli, A. Merzky, T. J. Naughton, W. R. Elwasif, and S. Jha, "Characterizing the performance of executing many-tasks on summit," in *IEEE/ACM Third Annual Workshop on Emerging Parallel and Distributed Runtime Systems and Middleware, IPDRM@SC 2019, Denver, CO, USA, November 22, 2019*. IEEE, 2019, pp. 18–25. [Online]. Available: https://doi.org/10.1109/IPDRM49579.2019.00007

[43] S. Chakraborty, H. Subramoni, J. L. Perkins, and D. K. Panda, "SHMEMPMI - shared memory based PMI for improved performance and scalability," in *IEEE/ACM 16th International Symposium on Cluster, Cloud and Grid Computing, CCGrid 2016, Cartagena, Colombia, May 16-19, 2016*. IEEE Computer Society, 2016, pp. 60–69. [Online]. Available: https://doi.org/10.1109/CCGrid.2016.99

[44] I. Vardas, M. Ploumidis, and M. Marazakis, "Improving the performance and resilience of MPI parallel jobs with topology and fault-aware process placement," *CoRR*, vol. abs/2012.14757, 2020. [Online]. Available: https://arxiv.org/abs/2012.14757

[45] Y. Tsujita, A. Hori, T. Kameyama, and Y. Ishikawa, "Topology-aware data aggregation for high performance collective MPI-IO on a multi-core cluster system," in *Fourth International Symposium on Computing and Networking, CANDAR 2016, Hiroshima, Japan, November 22-25, 2016*. IEEE Computer Society, 2016, pp. 37–46. [Online]. Available: https://doi.org/10.1109/CANDAR.2016.0022

[46] T. Ma, G. Bosilca, A. Bouteiller, and J. J. Dongarra, "Kernel-assisted and topology-aware MPI collective communications on multicore/many-core platforms," *J. Parallel Distributed Comput.*, vol. 73, no. 7, pp. 1000–1010, 2013. [Online]. Available: https://doi.org/10.1016/j.jpdc.2013.01.015

[47] M. J. Rashti, J. Green, P. Balaji, A. Afsahi, and W. Gropp, "Multi-core and network aware MPI topology functions," in *Recent Advances in the Message Passing Interface - 18th European MPI Users' Group Meeting, EuroMPI 2011, Santorini, Greece, September 18-21, 2011. Proceedings*, ser. Lecture Notes in Computer Science, Y. Cotronis, A. Danalis, D. S. Nikolopoulos, and J. J. Dongarra, Eds., vol. 6960. Springer, 2011, pp. 50–60. [Online]. Available: https://doi.org/10.1007/978-3-642-24449-0_8

[48] B. Claudel, G. Huard, and O. Richard, "Taktuk, adaptive deployment of remote executions," 01 2009, pp. 91–100.

[49] A. Gupta, G. Zheng, and L. V. Kalé, "A multi-level scalable startup for parallel applications," in *Proceedings of the 1st International Workshop on Runtime and Operating Systems for Supercomputers*, 2011, pp. 41–48.

[50] J. Goehner, D. Arnold, D. Ahn, G. Lee, B. Supinski, M. LeGendre, B. Miller, and M. Schulz, "Libi: A framework for bootstrapping extreme scale software systems," *Parallel Computing*, vol. 39, p. 167–176, 03 2013.

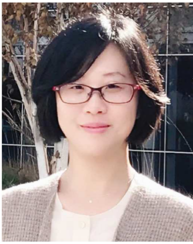
Yong Dong received the PhD degrees from National University of Defense Technology in 2012. He is now a professor in the College of Computer, National University of Defense Technology. His research interests include high performance computing, parallel storage, MPI, and resource management.

Yiqin Dai received the BS degree in National University of Defense Technology in 2019. He is currently working toward the PhD degree in the College of Computer, National University of Defense Technology. His research interests include high performance computing and resource management.

Min Xie received the PhD degrees from National University of Defense Technology. He is now a professor in the College of Computer, National University of Defense Technology. His research interests include parallel and distributed computing, high performance communications.

Kai Lu received the BS and PhD degrees from National University of Defense Technology in 1995 and 1999, respectively. He is now a professor in the College of Computer, National University of Defense Technology. His research interests include parallel programming and operating system and security.

Ruibo Wang received the BS and PhD degrees in computer science from National University of Defense Technology in 2003 and 2011, respectively. He is now a professor at National University of Defense Technology. His research interests include high performance computing and operating system.



Juan Chen received the PhD degree from National University of Defense Technology in 2007. She is now a professor at National University of Defense Technology. Her research interests include supercomputer systems and energy-efficient software optimization method.

Mingtian Shao received the BS and MS degree in National University of Defense Technology in 2019 and 2022, respectively. He is currently working toward the PhD degree in the College of Computer, National University of Defense Technology. His research interests include high performance computing and operating system.

Zheng Wang is Professor of Intelligent Software Technology at School of Computing at the University of Leeds. His research interests include compiler optimization, parallel programming and systems security.