Journal of Intelligent Computing and Networking
https://www.ffspub.com/index.php/jicn/index
ISSN 3079-9228 (print)
ISSN 3107-2798 (online)
E-mail: jicn.office@ffspub.com

Article

Computing Resource-Aware Operation Optimization Strategy for MPI Jobs in Cloud-Native Environment

Wenxiao Wang[1], Zibo Gao[1,2], Guoding Ji[1], Chentian Yong[3], Bo Li[3,†]
[1]*State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing 210023, China*
[2]*SINOPEC Key Laboratory of Geophysics, Nanjing 211103, China*
[3]*SINOPEC Geophysical Research Institute, Nanjing 211103, China*
[†]*E-mail: libo.swty@sinopec.com*

Received: July 22, 2026/ Revised: July 27, 2026/ Accepted: August 01, 2026/ Published online: August 06, 2026

Abstract: Large-scale high-performance computing workloads in petroleum geophysical exploration commonly use the Message Passing Interface (MPI) as their parallel programming model. However, MPI does not provide native mechanisms for resource management, which complicates the efficient execution of multiple MPI jobs in shared-resource environments. As MPI workloads migrate to cloud-native platforms, their high degrees of parallelism and communication-intensive behavior may lead to scheduling delays and contention for shared resources, resulting in prolonged execution time and inefficient resource utilization. This paper identifies two major limitations of existing cloud-native MPI deployments: cluster-unaware process-count selection and insufficient exploitation of Non-Uniform Memory Access (NUMA) locality. To address these limitations, we propose a resource-aware optimization framework that dynamically selects the MPI process count and performs node- and NUMA-aware process placement. Experimental results show that the proposed parallelism-selection method reduces task-sequence execution time by at least 18% and improves the evaluated resource-utilization metrics by more than 30%. The topology-aware placement method further reduces execution time by at least 10% compared with the evaluated affinity baselines under the tested shared-resource cluster configurations.

Keywords: Cloud-Native; MPI; Resource-Aware Optimization; NUMA Affinity; Dynamic Parallelism
https://doi.org/10.64509/jicn.23.137

1 Introduction

Petroleum geophysical exploration relies on seismic imaging to reconstruct subsurface structures and support geological interpretation. Reverse-time migration (RTM) uses two-way wave-equation extrapolation and is well suited to complex structures, including steeply dipping reflectors and strong lateral velocity variations [1]. However, repeated wavefield propagation over large computational grids and the storage or reconstruction of source wavefields make production-scale RTM computationally and memory intensive [2, 3]. Such workloads are executed on high-performance computing systems using the Message Passing Interface (MPI) to coordinate processes across nodes [4].

Cloud-native technologies provide a practical foundation for deploying distributed applications. Docker packages applications and dependencies into portable images [5], while Kubernetes automates deployment, resource allocation, and lifecycle management [6]. Cloud-native data-processing systems, including Apache Spark [7], serverless MapReduce [8], and serverless stream processing [9], demonstrate the benefits of automated resource management and adaptive parallelism. Although MPI workloads are more tightly coupled and communication intensive, migrating them to cloud-native platforms can improve deployment portability, resource sharing, and cluster manageability.

The Kubeflow MPI-operator represents MPI jobs as Kubernetes Custom Resources and automates their lifecycle management [10]. However, it primarily addresses deployment and orchestration rather than workload-specific performance optimization. Kubernetes provides node affinity, pod affinity, anti-affinity [11], and optional Non-Uniform Memory Access (NUMA) -aware topology management [12], but these mechanisms do not directly select MPI process counts or place ranks according to application behavior and current resource availability. These mechanisms automate job

† Corresponding author: Bo Li
*Academic Editor: Mengwei Xu

management or enforce user-specified placement and topology constraints, but they do not estimate workload-specific NUMA-level process capacity or adapt the MPI process count to the dynamically available cluster resources in real time.

Existing studies have examined containerized High-Performance Computing (HPC) and MPI execution [13–15], MPI deployment and management [16–18], runtime and configuration tuning [19–21], and resource coordination in distributed environments [22]. Nevertheless, limited attention has been paid to considering cluster availability, job waiting time, process count, and NUMA locality for MPI jobs on shared Kubernetes clusters. In contrast to single-job runtime and configuration tuning, the proposed parallelism-selection algorithm evaluates candidate process counts using predicted execution time, current resource availability, and the estimated completion times of running jobs, and minimizes the sum of job waiting time and execution time. In contrast to generic affinity and topology-alignment mechanisms, the proposed placement algorithm estimates the runnable process capacity of each NUMA node from the predicted per-process memory, network-bandwidth, and memory-bandwidth requirements. It then applies a hierarchical placement policy that prioritizes one NUMA node, one physical node, and finally the minimum feasible number of physical nodes. These two algorithms therefore provide workload- and cluster-state-aware optimization rather than applying a static process count or generic affinity constraints.

1.1 Challenges

This paper focuses on two coupled challenges in optimizing MPI jobs on shared cloud-native clusters: selecting an appropriate process count under dynamically available resources and placing MPI processes according to node and NUMA topology. These challenges affect not only the execution time of individual jobs but also job waiting time, resource contention, and task-sequence completion time.

Resource sharing introduces an important difference between cloud-native clusters and traditional resource-exclusive MPI environments. Kubernetes can enforce CPU and memory limits through cgroups, but shared resources such as memory bandwidth, network bandwidth, and cache capacity are not isolated with the same granularity. Consequently, concurrently running jobs may interfere with one another, a problem also observed in cloud-native and edge computing systems [23]. A process count that minimizes the isolated execution time of a single MPI job may therefore be unsuitable for a shared cluster. Requesting excessive parallelism can increase resource contention or keep a job in the Pending state until sufficient resources become available, thereby increasing its total completion time and reducing task-sequence efficiency.

Process placement becomes another optimization dimension when MPI jobs are deployed on Kubernetes. Kubernetes provides affinity and anti-affinity mechanisms to influence the placement of Pods across nodes, but these generic policies do not directly determine an efficient placement for tightly coupled MPI processes. Placing communicating processes on fewer physical nodes can reduce inter-node communication, whereas excessive consolidation may place processes from multiple jobs on the same NUMA node and intensify contention for local memory bandwidth. Therefore, MPI process placement must jointly consider inter-node communication, node-level resource availability, and NUMA locality rather than relying only on coarse-grained affinity rules.

1.2 Contributions

To address these challenges, this paper presents a resource-aware optimization framework and a Kubernetes-based prototype system for cloud-native MPI jobs. The main contributions are summarized as follows:

(1) We propose two complementary resource-aware optimization mechanisms. The parallelism selection method combines performance and resource prediction with the current cluster state to select a process count for shared clusters that minimizes the estimated sum of job waiting time and execution time. The process placement method estimates the runnable process capacity of NUMA nodes and assigns MPI processes according to node and NUMA locality, thereby reducing memory-bandwidth contention and unnecessary inter-node communication. The algorithmic novelty lies in incorporating predicted job waiting time into process-count selection and using estimated NUMA-level runnable capacity to drive hierarchical process placement.

(2) We implement the proposed methods in a Kubernetes-based prototype system and evaluate them using task sequences constructed from the publicly available NAS Parallel Benchmarks (NPB) suite, supplemented by a production-related RTM case study. The experimental results show that resource-aware parallelism selection reduces task-sequence execution time by at least 18% and improves the evaluated resource-utilization metrics by more than 30% compared with the single-job performance-oriented baseline. The node- and NUMA-aware placement method further reduces execution time by at least 10% compared with the evaluated affinity baselines.

2 Background

This section introduces the technical background required by the proposed optimization framework, including containerized resource isolation, Kubernetes scheduling, cloud-native MPI deployment, MPI computation–communication trade-offs, and NUMA locality. These concepts explain why process-count selection and process placement must consider both dynamic cluster resources and hardware topology.

2.1 Containerization and Kubernetes Orchestration

Containerization is an operating-system-level virtualization technique that packages an application and its dependencies into a portable image. Compared with virtual machines, containers share the host operating-system kernel and introduce lower startup and deployment overhead in practical deployment scenarios. For MPI applications, container images improve software consistency across computing nodes and simplify environment configuration. However, containers running on the same host still share physical resources,

including processor caches, memory channels, network interfaces, and storage devices, which may cause performance interference among concurrently running workloads.

Docker provides image construction and container execution mechanisms for packaging and running MPI applications [5]. At runtime, Linux namespaces isolate process, network, and file-system views, while control groups constrain the CPU and memory resources available to containers. These mechanisms provide basic isolation and accounting but do not guarantee equivalent isolation for shared resources such as memory bandwidth, last-level cache capacity, and network bandwidth. Consequently, resource limits expressed only in terms of CPU cores and memory capacity may not accurately reflect the interference experienced by communication- and memory-intensive MPI workloads.

Kubernetes manages containerized workloads using Pods as its basic scheduling units [6]. The default scheduler assigns Pods to nodes according to declared resource requests, node availability, and placement constraints. CPU and memory requests are directly considered during scheduling, while node affinity, pod affinity, anti-affinity, and topology constraints can be used to influence placement [11]. However, the default scheduling model operates primarily at node granularity and does not directly capture application-specific communication patterns or fine-grained NUMA-level resources such as local memory bandwidth. Batch scheduling systems such as Volcano and MPI-specific controllers such as Kubeflow MPI-operator extend Kubernetes to support coordinated execution of multi-Pod MPI jobs [10].

2.2 MPI and Cloud-Native Deployment

The Message Passing Interface defines standardized communication operations for coordinating multiple processes, commonly referred to as MPI ranks, within a parallel application [4]. For example, RTM, a core algorithm in petroleum geophysical exploration, is a typical MPI-parallel application in which each MPI rank handles a subdomain of the computational grid and exchanges boundary data with neighboring ranks at each time step. An MPI job may distribute its ranks across multiple cores and physical nodes. Increasing the number of ranks can reduce the execution time of parallelizable computation, but it can also increase communication, synchronization, and resource demand. Therefore, the process count that minimizes isolated job runtime is not necessarily the process count that minimizes total completion time in a resource-sharing cluster.

MPI communication includes point-to-point operations between pairs of ranks and collective operations involving groups of ranks. Blocking and non-blocking communication differ in whether communication can overlap with computation, but the proposed method does not optimize MPI primitives separately. Instead, it models the execution time of an MPI job as the sum of computation time and communication time. The communication component is affected by the process count, data size, process placement, and network resources, while the computation component depends on the parallelizable fraction of the workload and the computing resources assigned to each process.

Collective communication coordinates data exchange and synchronization among groups of MPI ranks. Representative operations include broadcast, reduction, data redistribution, and all-to-all exchange. Their costs generally depend on the number and placement of participating ranks, the communicated data volume, and the underlying network topology. Although this paper does not distinguish individual collective algorithms, their aggregate cost contributes to the communication term in the performance model.

Cloud-native MPI deployments commonly rely on Kubernetes extensions such as MPI-operator. The MPI-operator represents an MPI job as a Kubernetes Custom Resource and manages its launcher and worker Pods through the Operator reconciliation mechanism [10]. Although it automates job creation, status tracking, and lifecycle management, users must still specify the desired process count and Pod resource requirements. Consequently, execution efficiency depends on whether the selected process count matches current cluster availability and whether worker processes are placed on suitable physical nodes. A process count optimized only for isolated execution may increase scheduling delay under resource scarcity, while topology-unaware placement may increase inter-node communication or memory-bandwidth contention. These limitations motivate resource-aware process-count selection and node- and NUMA-aware process placement.

2.3 NUMA Locality and Process Affinity

Modern multi-socket servers commonly employ a NUMA architecture, in which processor cores are organized into NUMA nodes with directly attached local memory. Memory-access performance is non-uniform: accessing memory attached to the local NUMA node generally incurs lower latency and provides higher effective bandwidth than accessing memory attached to a remote node [24]. Moreover, processes executing on the same NUMA node share local memory channels and may contend for memory bandwidth. CPU binding alone therefore does not necessarily guarantee memory locality; CPU affinity and memory placement should be coordinated so that MPI processes primarily access local memory.

Kubernetes provides topology-management mechanisms that can align requested CPU, device, and memory resources with NUMA nodes when the corresponding managers and policies are explicitly configured [12]. However, these mechanisms primarily perform node-local resource alignment at the Pod or container level and do not by themselves select the MPI process count or determine workload-specific, cluster-wide placement according to predicted communication and memory-bandwidth demands. In shared cloud-native clusters, resource-aware placement of processes across NUMA nodes can therefore further reduce remote-memory access and inter-job memory-bandwidth contention.

3 Computing Resource-Aware Optimization for Cloud-Native MPI Jobs

To address the dynamic resource contention and operational bottlenecks inherent to MPI workloads in cloud environments, this paper proposes a computing resource-aware optimization strategy tailored for cloud-native deployments. The proposed framework systematically mitigates performance degradation through two complementary mechanisms: a parallelism optimization strategy that dynamically aligns computational concurrency with available resource capacity, and a topology-aware affinity placement strategy that minimizes inter-process communication overhead. The efficacy and scalability of these approaches are rigorously validated through comprehensive experiments, demonstrating significant improvements in resource utilization and execution efficiency under varying cloud-native conditions.

3.1 Motivation and Problem Analysis

In traditional bare-metal or cloud host operating environments, tuning of MPI tasks has been extensively studied. However, as MPI tasks migrate to cloud-native platforms, the shared and orchestrated execution environment introduces new optimization requirements. On the one hand, although many studies [25–27] have shown that the containerized operating environment does not affect the performance of MPI jobs under the assumption of exclusive resources and no resource competition, in the actual environment provided by Kubernetes, resource competition such as memory bandwidth still exists. Unreasonable parallelism setting can easily lead to excessive resource competition, which in turn drags down performance. On the other hand, the affinity characteristics of the Kubernetes platform add a new dimension to MPI task tuning. Reasonably setting the affinity between multiple MPI processes or between nodes can significantly reduce competition in network bandwidth, memory bandwidth, etc., and effectively improve task operating efficiency.

3.1.1 Cluster-Unaware Parallelism Selection

In terms of parallelism setting of MPI jobs, traditional tuning strategies usually aim at pursuing the best performance of a single task, and assume that resources are exclusive and unlimited, without considering the efficiency of the overall operating platform. Specifically, this traditional approach tends to maximize the number of processes to achieve the highest possible speedup ratio, assuming that all allocated cores and memory bandwidth are dedicated solely to the target job. However, in the new environment of cloud-native Kubernetes, which emphasizes efficient resource sharing, these assumptions no longer hold. Cloud-native platforms can only allocate and share resources on demand, and only provide isolation at the CPU and memory levels. If the original performance-first parallelism setting strategy is continued, it is very likely that multiple tasks will be deployed on the same node, causing resource competition and scheduling delays. In addition, setting parallelism only based on the performance of a single task and ignoring the current cluster resource status may cause the tuned task to be unable to run for a long time due to insufficient resources, which not only prolongs the total running time but also reduces the resource utilization of the cluster.

As shown in Figure 1, adopting the parallelism setting strategy based on the optimal performance of a single task, although each task has the best theoretical performance, in the resource-sharing cloud-native platform, due to resource constraints, tasks often cannot run immediately and need to wait for the previous task to complete and release resources, which undoubtedly leads to longer waiting time, thus prolonging the total duration and reducing cluster resource utilization. In contrast, as shown in Figure 2, adding the perception function of the current cluster resource status on the basis of the task performance-based parallelism setting strategy can effectively reduce task waiting time, shorten the total running time, and improve resource utilization.

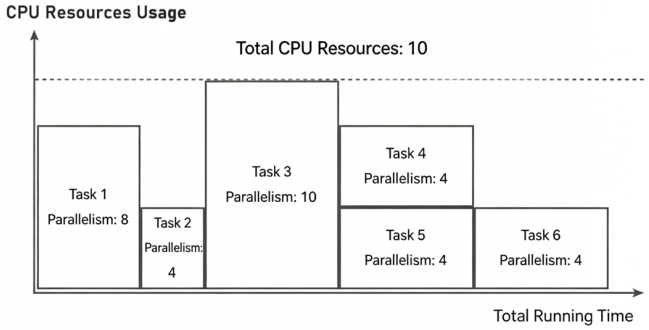

Figure 1: Task running sequence with parallelism setting based on optimal single-task performance.

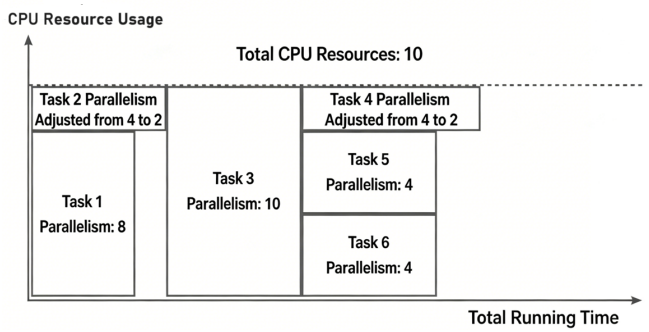

Figure 2: Task running sequence after adjusting parallelism based on cluster status.

Based on this, this paper conducted a motivation experiment on two nodes (each node is configured with 24-core CPU, 256GB memory, and inter-node bandwidth of 10Gb/s), simulating the task sequence running with parallelism based on the best performance of the task and the task sequence after adjusting parallelism based on the current resource status respectively. The task sequence consists of multiple Conjugate Gradient Solver (CG.D.X) and Gauss-Seidel Solver (LU.D.X) from NAS Parallel Benchmarks [28]. The experimental results are shown in Figure 3. Although parallelism adjustment based on resource status increases the running time of some tasks to a certain extent, it significantly shortens the waiting time, improves the overall resource utilization, and reduces the total running time by about 20%.

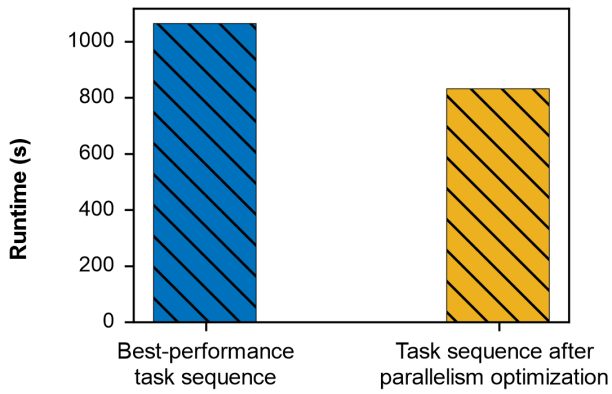

Figure 3: Execution time of task sequence before and after parallelism adjustment.

3.1.2 Affinity-Unaware Process Placement

When deploying tasks on the cloud-native platform Kubernetes, inter-task affinity is often required to place frequently interacting tasks on the same node for communication efficiency. Meanwhile, node affinity can deploy tasks to more suitable nodes, thereby improving operating efficiency. However, traditional deployment methods such as MPI do not have affinity characteristics themselves, and tasks are usually centrally deployed according to resource consumption. Moreover, due to resource exclusivity in traditional environments, NUMA binding is generally not involved. In the new cloud-native deployment environment, resource allocation and scheduling modes have changed significantly, and the original deployment method can no longer meet the demand. Therefore, a method is needed to flexibly set affinity according to the real-time cluster status, so as to adapt to the complexity and dynamics of cloud-native environment and ensure efficient and stable task execution.

In cloud-native platforms, unreasonable affinity settings may cause many problems, such as scheduling failures. For example, if hard affinity is forcibly set without considering the actual cluster state, tasks may fail to start and fall into a long wait because no node satisfies the conditions. When soft affinity is used, ignoring resource consumption on nodes may cause too many processes to be deployed on the same node, resulting in resource competition.

In addition, today's servers mostly adopt multi-NUMA structures. CPUs in the same NUMA node share memory bandwidth, while tasks on different NUMA nodes do not affect each other. In traditional resource-exclusive environments, since only a single task runs, there is no competition between tasks, and NUMA affinity binding is unnecessary. However, in cloud-native resource-sharing platforms, binding different tasks to different NUMA nodes can compensate for the lack of memory bandwidth isolation in Kubernetes and avoid inter-task resource competition. As shown in Figure 4, if two tasks run on the same NUMA node simultaneously, memory access competition will occur, reducing task execution efficiency.

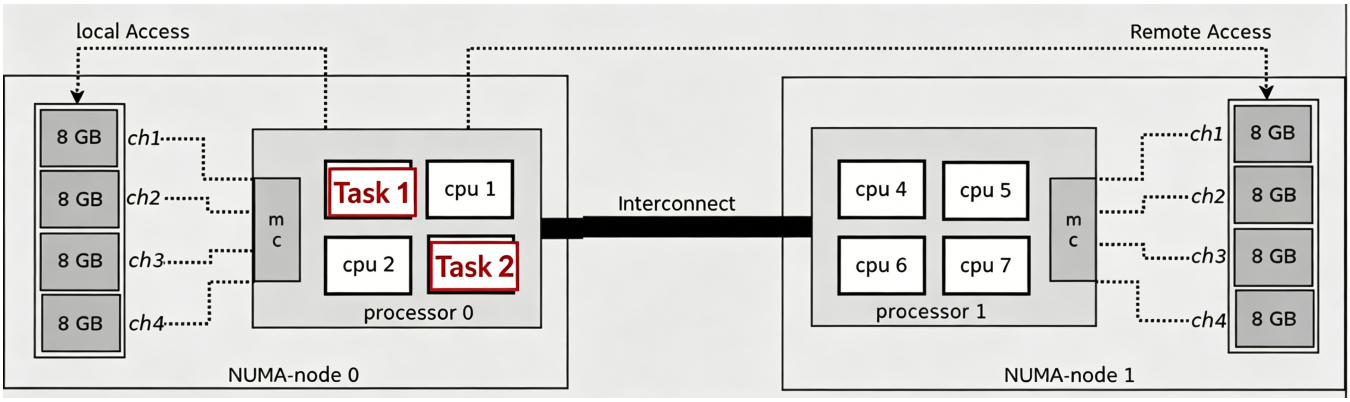

Figure 4: Memory bandwidth resource competition caused by multiple tasks running on the same NUMA node.

Based on the above background, this paper conducted a motivation experiment on two nodes (each node is configured with 48-core CPU, 256GB memory, and inter-node bandwidth of 10Gb/s). The experiment simulated task sequences with soft affinity, hard affinity, and NUMA-aware node affinity respectively. The task sequence consists of multiple Conjugate Gradient Solver (CG.D.X) and Discrete Three-Dimensional Fast Fourier Transform (FT.D.X) from NAS Parallel Benchmarks. The experimental results are shown in Figure 5. By reasonably binding tasks to different NUMA nodes according to the resource status on NUMA nodes, the running time of tasks can be effectively shortened, with a reduction of about 18%. The motivation experiments shown in Figures 3 and 5 were conducted under separate two-node configurations and are independent of the formal evaluation presented in Section 4.

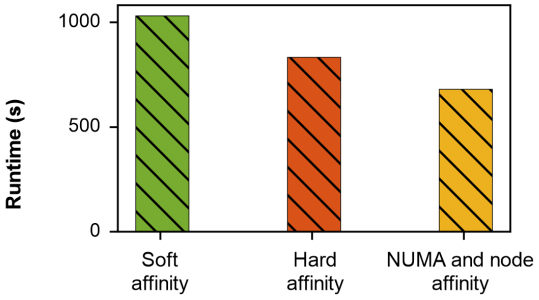

Figure 5: Execution time of task sequence under different affinity setting strategies.

3.2 Resource-Aware Parallelism Optimization

To effectively solve the parallelism setting problem of MPI tasks in cloud-native environment, this paper proposes a computing resource-aware parallelism optimization strategy for

cloud-native MPI jobs. The specific process of this strategy is shown in Figure 6, which mainly covers two key stages: performance and resource prediction, and parallelism selection under resource constraints.

In the first stage, the framework uses the performance and resource models to estimate the execution time, resource demand, and waiting time for each candidate process count. In the parallelism selection stage, the algorithm evaluates the candidate process counts and selects the one that minimizes the estimated sum of waiting time and execution time.

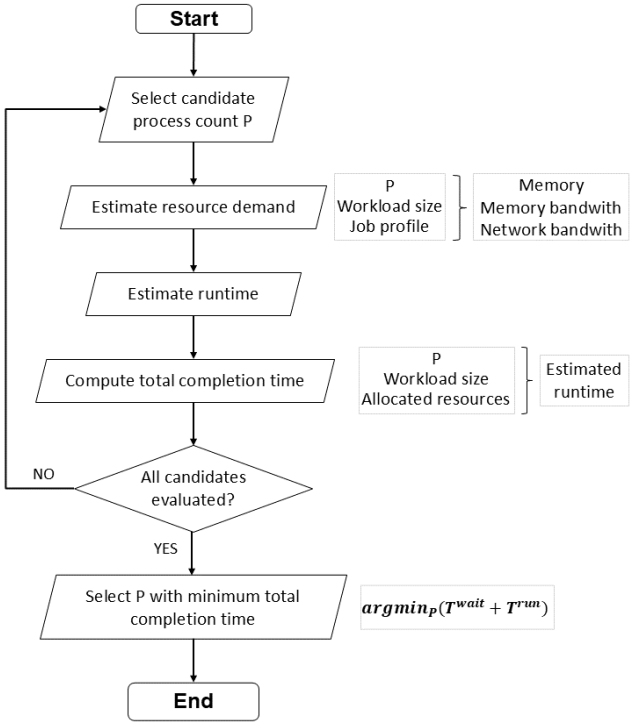

Figure 6: Workflow of computing resource-aware parallelism optimization strategy for cloud-native MPI jobs.

3.2.1 Performance and Resource Models

To predict the performance and resource requirements of MPI jobs, we model the computation and communication characteristics of parallel execution and construct regression models from historical resource measurements. For the performance model, to simplify the analysis process, this paper assumes that the task will not encounter resource competition from other tasks during operation (the guarantee measures will be detailed in subsequent sections). Accordingly, the execution time of an MPI task is divided into two components: communication and computation, as shown in Equation (1), where T_i^{run} represents the running time of task i, T_i^{comp} represents the time consumed by task i in the computation part, and T_i^{comm} represents the time consumed by task i in the communication part. For the communication process, the communication library provides two ways: point-to-point communication and collective communication. Among them, the overhead of one-to-many communication has a linear relationship with the number of nodes, while the overhead of many-to-many communication has a square relationship with the number of processes. At the same time, there is a specific relationship between the data volume of a single communication and the total data volume. Based on this, the communication process model we constructed is shown in Equation (2), where β and γ are fitting coefficients for collective communication and point-to-point communication respectively, D_i is the data size of task i, and P_i is the parallelism of task i, that is, the number of processes.

$$T_i^{run} = T_i^{comp} + T_i^{comm} \tag{1}$$

$$T_i^{comm} = \beta \cdot D_i \cdot P_i^2 + \gamma \cdot D_i \cdot P_i \tag{2}$$

For the computation time consumption during MPI operation, this paper further decomposes it into parallelizable part and non-parallelizable part. The parallelizable part refers to operations that can be processed in parallel by distributing data to each node, such as accumulation operations of each node. The non-parallelizable part refers to operations that can only run on a single node, such as the summary operation of accumulation results of each node. The running time of the parallelizable part will decrease with the increase of the number of processes, while the running time of the non-parallelizable part is not affected by the change of the number of processes. In addition, the entire computation process is proportional to the data size. Based on these characteristics, we model the computation part as shown in Equation (3), where α is the fitting coefficient of the computation part, D_i is the data size of task i, P_i is the parallelism of task i, that is, the number of processes, and $F_i^{parallel}$ is the parallelizable ratio of task i. The fitting coefficients in both computation and communication models can be fitted through the performance of subsequent actual tasks, and there are few fitting parameters, so the construction of the task running time model can be completed with only a small number of runs.

$$T_i^{comp} = \alpha \cdot D_i \cdot \left(\frac{F_i^{parallel}}{P_i} + 1 - F_i^{parallel} \right) \tag{3}$$

In terms of resource prediction of MPI tasks, through preliminary observation and analysis of relevant literature, it is found that constructing memory usage, memory bandwidth usage, and network bandwidth usage models requires in-depth analysis of the specific running code of the task, and it is difficult to achieve abstract generalization through simple modeling. In view of this, this paper establishes a regression model based on the historical data of task operation under different workload configurations to analyze the relationship between memory usage, network bandwidth usage, memory bandwidth usage and parallelism and data volume. The established regression model is shown in Equation (4), where R_i^{mem}, R_i^{net} and R_i^{mbw} are the predicted values of memory, network bandwidth and memory bandwidth consumed by each process of task i respectively, and f_i^{mem}, f_i^{net} and f_i^{mbw} are the regression functions of task i on memory, network bandwidth and memory bandwidth with respect to data size D_i and parallelism P_i respectively. In terms of the resource model of the task, the regression function will be constructed based on the historical resource consumption of the task.

$$\begin{cases} R_i^{mem} = f_i^{mem}(D_i, P_i) \\ R_i^{net} = f_i^{net}(D_i, P_i) \\ R_i^{mbw} = f_i^{mbw}(D_i, P_i) \end{cases} \tag{4}$$

Actual MPI performance can be affected by various software and hardware factors, including fine-grained communication patterns, cache and memory-hierarchy behavior, disk I/O, and time-varying resource contention. The current models do not parameterize all these factors separately. This simplification is intentional because incorporating detailed microarchitectural behavior would substantially increase the profiling cost, parameter-acquisition overhead, online scheduling latency, and deployment complexity of the scheduling system. Instead, the models focus on workload size, process count, aggregate computation and communication costs, and predicted per-process resource demands, which capture the dominant performance and resource trends required by the scheduler within the evaluated workloads. The aggregate effects of factors that are not explicitly modeled are partially reflected in the workload-specific fitting coefficients and historical measurements, although previously unseen workload behavior or strong transient interference may still increase the prediction error. Therefore, the models are designed to provide sufficiently accurate performance estimates for scheduling decisions rather than cycle-level performance prediction.

3.2.2 Parallelism Selection Algorithm

To select an appropriate process count for an MPI task in a cloud-native environment, the proposed algorithm combines the performance and resource models with the current cluster state to minimize task completion time and improve resource utilization. When an MPI task is deployed to a resource-sharing cloud-native platform, if cluster resources are sufficient, its Pods can be scheduled and executed. Conversely, if cluster resources are insufficient, the corresponding Pods will remain Pending until some jobs complete and release sufficient resources. Based on this resource-sharing environment, the total completion time of a task consists of waiting time and running time. Adjusting task parallelism will affect the total completion time. For example, increasing task parallelism can shorten running time, but may also prolong waiting time due to resource shortage. In view of this, this paper proposes the optimization objective shown in Equation (5), where T_i^{wait} is the waiting time of task i, T_i^{run} is the running time of task i, and P_i is the parallelism of task i, that is, the solution target.

$$argmin_{P_i} \left(T_i^{\text{wait}} + T_i^{\text{run}} \right) \tag{5}$$

To solve the optimal parallelism that minimizes the total completion time of the task, this paper proposes an MPI parallelism setting algorithm with cluster computing resource awareness. The algorithm first obtains the current resource usage status of the cluster, and combines the estimated completion time of running jobs to calculate the waiting time and estimated running time of the target task under a specific parallelism setting. Then, the algorithm continuously adjusts the parallelism and searches for the optimal solution under current cluster conditions until it finds the optimal parallelism that minimizes the total completion time of the task. The specific algorithm flow is shown in Algorithm 1.

In Algorithm 1, P_i^{opt} denotes the candidate parallelism explored in each iteration, and T records the minimum total completion time found so far. R_i^{mem}, R_i^{net}, and R_i^{mbw} denote the per-process requirements of memory, network, and memory bandwidth, while R_{node}^{mem}, R_{node}^{net}, and R_{node}^{mbw} denote the corresponding available resources on each node. P_{node} and P_{sum} represent the node-level and cluster-level runnable process capacities, respectively.

Algorithm 1 Computing Resource-Aware MPI Parallelism Setting Algorithm

Require: Initial parallelism P_i^{init}, task data size D_i
Ensure: Optimal parallelism P_i^{res}

```
1:  function GETBESTPROCESS(P_i^init, D_i)
2:      P_i^opt ← P_i^init
3:      T ← ∞
4:      while true do
5:          R_i^mem ← f_i^mem(D_i, P_i^opt)
6:          R_i^net ← f_i^net(D_i, P_i^opt)
7:          R_i^mbw ← f_i^mbw(D_i, P_i^opt)
8:          P_sum ← 0
9:          for each node in Nodes do
10:             P_node ← min( R_node^mem / R_i^mem , R_node^net / R_i^net , R_node^mbw / R_i^mbw )
11:             P_sum ← P_sum + P_node
12:         end for
13:         T_wait ← 0
14:         if P_sum < P_i^opt then
15:             Tasks ← sortTasksByFinishTime()
16:             for each task in Tasks do
17:                 P_sum ← P_sum + tryFreeResource(task)
18:                 if P_sum ≥ P_i^opt then
19:                     T_wait ← finishTime − now
20:                     break
21:                 end if
22:             end for
23:         end if
24:         if P_sum ≥ P_i^opt then
25:             T_total ← T_wait + getRuntime(P_i^opt, D_i)
26:             if T_total < T then
27:                 P_i^res ← P_i^opt
28:                 T ← T_total
29:             end if
30:             if T_wait = 0 then
31:                 break
32:             end if
33:         end if
34:         P_i^opt ← P_i^opt − 1
35:         if P_i^opt < 1 then
36:             break
37:         end if
38:     end while
39:     return P_i^res
40: end function
```

This algorithm takes the parallelism input by the user (usually the optimal parallelism under resource exclusive conditions) as the starting point of exploration, and decreases the parallelism value by one after each exploration. In each exploration (lines 3-29), the algorithm obtains the current resource status of the cluster in real time, and calculates the number of processes that the node can carry according to the available resources of NUMA nodes on each node and the usage of memory, network and memory bandwidth per process (lines

5-8). If the number of task processes that the cluster can run at this time is less than the currently set parallelism, the algorithm sorts the running tasks in ascending order of completion time. Then, it determines after which tasks are completed the current job can obtain sufficient resources, and takes this waiting duration as the task waiting time (lines 9-18). Finally, the algorithm adds the task running time and waiting time to get the total completion time (lines 19-25). In the entire exploration process, the parallelism setting corresponding to the shortest total completion time is selected as the final return value, which is the optimal parallelism setting.

The computational complexity of Algorithm 1 is analyzed as follows. Let K denote the number of candidate process counts evaluated, N the number of cluster nodes, and M the number of currently running jobs. For each candidate process count, the algorithm traverses the N nodes and, when the available resources are insufficient, sorts and scans the running jobs according to their estimated completion times. Assuming that each performance or resource model evaluation takes constant time, the worst-case time complexity is $O(K(N + M\log M))$. The additional space complexity is $O(M)$, mainly for storing the ordered list of running jobs.

3.3 Node- and NUMA-Aware Placement

3.3.1 NUMA-Level Process Capacity Estimation

With the continuous development of hardware systems, the traditional way of improving performance by increasing the number of cores for a single CPU has gradually reached a bottleneck. In multi-core architectures, the competition for shared memory access among multiple cores is becoming increasingly fierce under high concurrency workloads, leading to a significant increase in memory access latency, and bus bandwidth has also become a bottleneck for performance improvement, seriously affecting the overall system performance. In this context, the Non-Uniform Memory Access (NUMA) architecture emerged as the times require.

The NUMA architecture breaks through the mode of unified shared memory in traditional multi-processor architectures, dividing memory into multiple nodes, each equipped with local memory, and processors are closely connected to local memory. This makes processors access local memory with fast speed and low latency, while accessing remote memory has poor performance. In the NUMA architecture, there are multiple NUMA nodes, each with a multi-core processor and directly connected to multiple memory slots. When tasks run on different NUMA nodes, if the memory allocated to them is located in the local memory slot, there will be no memory bandwidth competition problem.

To avoid memory bandwidth competition when MPI tasks run on the same NUMA node, this paper calculates the number of runnable processes on each NUMA node and allocates the memory required by the process to the local memory slot, thereby avoiding cross-NUMA node memory access. When calculating the number of runnable processes on each NUMA node, this paper uses Intel® open-source NUMA node monitoring service [29] to collect hardware-level metrics, particularly local and remote memory-bandwidth measurements. The specific calculation formula is shown in Equation (6), where $P_{i,j}$ is the number of runnable processes on the j-th NUMA node of the i-th node, $R^{mem}_{i,j}$, $R^{net}_{i,j}$, $R^{mbw}_{i,j}$ and $R^{core}_{i,j}$ are the remaining memory, network, memory bandwidth and core resources on the j-th NUMA node of the i-th node respectively, and R^{mem}_{k}, R^{net}_{k} and R^{mbw}_{k} are the predicted values of memory, network bandwidth and memory bandwidth consumed by each process of task k respectively.

$$P_{i,j} = \min \begin{cases} R^{mem}_{i,j} / R^{mem}_{k} \\ R^{net}_{i,j} / R^{net}_{k} \\ R^{mbw}_{i,j} / R^{mbw}_{k} \\ R^{core}_{i,j} \end{cases} \tag{6}$$

After calculating the number of runnable processes on each NUMA node, to reduce storage overhead, this paper uses the Bitmap data structure to store relevant information. The specific storage structure is shown in Figure 7. In this Bitmap, each entry represents the estimated runnable process capacity of one NUMA node. Every N bits (N is the number of available NUMA nodes on the node) form a group, representing a host node, and the whole represents the number of runnable processes on the host node.

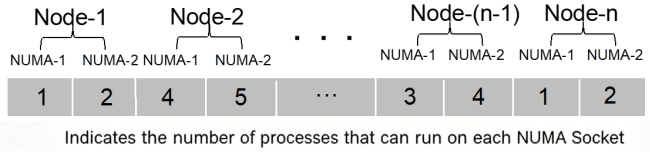

Figure 7: Storage structure of the number of runnable processes per NUMA node.

3.3.2 Node- and NUMA-Aware Process Placement

Based on the NUMA-level process capacities estimated in Section 3.3.1, the proposed placement method assigns MPI processes with node- and NUMA-level locality awareness. If one NUMA node has sufficient capacity for all processes of a job, the method places them on that NUMA node to avoid remote-memory access. If all processes cannot be placed on the same NUMA node, the method attempts to place them within the same physical node to avoid inter-node communication. For jobs that cannot run on the same physical node, the processes are distributed across as few physical nodes as possible to reduce communication overhead. To this end, the placement method consists of the following three steps:

First, the algorithm will try to deploy the task on the same NUMA node. This method sorts the constructed Bitmap structure according to the number of runnable processes. Then, according to the parallelism of the task, use the binary search algorithm to find the most suitable NUMA node to set affinity and deploy the task to minimize resource waste, as shown in Figure 8(a).

Second, for tasks that cannot run on the same NUMA node, this paper merges the number of runnable nodes originally counted at the NUMA node granularity into the host granularity, that is, clarifies the number of runnable processes on each host node, and sorts them in descending order. Similarly, use binary search to find the most suitable host node, and then complete the node and NUMA affinity setting and binding operation according to the number of runnable processes on each NUMA node, as shown in Figure 8(b).

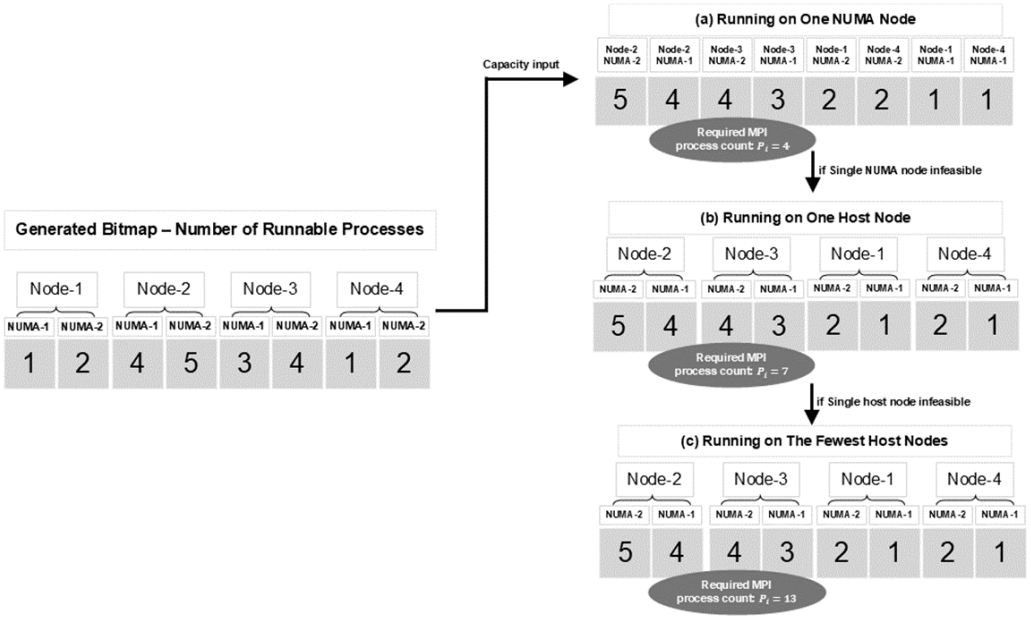

Figure 8: NUMA and node affinity setting process.

Finally, for tasks that cannot run on the same host node, this method deploys them on as few host nodes as possible to reduce cross-node communication. This process is similar to trying to deploy on the same host node. First, sort in descending order based on the number of runnable processes on each host node, and then sequentially set the affinity of some processes to these nodes until the number of processes to be deployed is 0, as shown in Figure 8(c).

4 Experimental Evaluation

This section evaluates the proposed framework in terms of prediction accuracy, parallelism selection, and node- and NUMA-aware process placement. The evaluation uses NAS Parallel Benchmarks under different cluster configurations and further includes a complementary case study with production-related RTM workloads.

4.1 Experimental Setup

The experiments in this section are conducted in a cluster containing 6 nodes. Each node includes 2 Intel(R) Xeon(R) Silver 4314 CPUs @ 2.40GHz, each CPU contains 16 physical cores, and the node memory is 256GB. In terms of system software, each node runs CentOS Linux release 8.5.2111 operating system. In terms of application software, Kubernetes v1.22.4 is installed in the above cluster as the cluster container scheduler, and the container runtime on each node is Docker v19.03.14. Considering that MPI-operator is currently one of the most widely used MPI task managers in cloud-native environment, this paper selects MPI-operator v0.5.0 as the task manager. For the cluster monitoring system, this paper deploys Prometheus v0.76.1.

4.2 Benchmark Workloads

We use the publicly available NASA NAS Parallel Benchmarks (NPB) benchmark suite, developed by NASA Ames Research Center, as the primary reproducible workload suite for evaluating the proposed methods [28]. As shown in Table 1, NPB contains widely used HPC benchmark applications with different computation and communication characteristics, including embarrassingly parallel computation, integer sorting, conjugate-gradient solving, multigrid computation, fast Fourier transforms, and implicit solvers. Its public availability, standardized workload classes, and widespread use in parallel-computing research facilitate experimental reproducibility and fair comparison with related studies under controlled cluster conditions. We use eight NPB applications at Class D and construct task sequences by randomly selecting benchmark applications. Although NPB provides representative HPC workloads, it cannot cover all production MPI applications. We therefore retain NPB as the primary reproducible evaluation suite and additionally present a single production-related RTM case study with two workload scales in Section 3.3.1. The RTM inputs were obtained from an industrial production workflow, while the reported experiments were reproduced on a dedicated test server. Separately, the proposed cloud-native MPI approach has been applied in a real petroleum-geophysical production environment, providing evidence of its practical deployability.

Table 1: NPB Benchmark Applications and Their Functions

Task Name	Description
IS	Integer Sorting Benchmark
EP	Embarrassingly Parallel Benchmark
CG	Conjugate Gradient Method Benchmark
MG	Multi-Grid Method Benchmark
FT	Fast Fourier Transform Benchmark
BT	Block Tridiagonal Benchmark
SP	Scalar Pentadiagonal Benchmark
LU	Lower-Upper Gauss-Seidel Benchmark

4.3 Evaluation Metrics

We evaluate the proposed optimization framework using the following metrics:

(1) Mean Absolute Percentage Error (MAPE). MAPE measures the prediction accuracy of the performance and resource models by quantifying the relative deviation between predicted and observed values. For n observations, MAPE is calculated as

$$\mathrm{MAPE} = \frac{1}{n}\sum_{i=1}^{n}\left|\frac{y_i - \hat{y}_i}{y_i}\right| \times 100\%, \tag{7}$$

where y_i and $\hat{y}_i$ denote the observed and predicted values. A lower MAPE indicates higher prediction accuracy.

(2) Task-sequence execution time. This metric is defined as the elapsed time from the submission of the first job to the completion of the last job in a task sequence. It captures both job waiting time and execution time and is therefore used to evaluate the overall scheduling efficiency of the proposed parallelism and placement strategies.

(3) Resource utilization. Resource utilization measures the proportion of available cluster resources used during task-sequence execution. We report CPU, memory, memory-bandwidth, and network-bandwidth utilization separately. For each resource type, utilization is calculated as the observed resource usage relative to the corresponding total cluster capacity over the task-sequence execution period. Higher utilization indicates that the available resources are used more continuously and efficiently, but it should be interpreted together with task-sequence execution time because high utilization alone does not necessarily imply better application performance.

4.4 Baseline Methods

We compare the proposed methods with the following baselines under identical cluster configurations and task sequences throughout all evaluation experiments. The baseline labels MP-STEOMP and Kube-HPC-FGS used below are consistent with those shown in Figures 10–18. For each MPI job, MP-STEOMP selects the process count that minimizes the execution time predicted by the same fitted performance model used by the proposed method. Following the performance-modeling approach in [19], this baseline selects, for each MPI job, the process count that minimizes the execution time predicted by the performance model. MP-STEOMP uses the same fitted performance model and searches the same candidate process-count range as the proposed parallelism-selection algorithm. However, its optimization objective considers only the predicted execution time of an individual job, whereas the proposed algorithm minimizes the estimated sum of job waiting time and execution time. MP-STEOMP does not incorporate current cluster availability, the predicted completion times of running jobs, or the waiting time caused by insufficient resources. The comparison therefore isolates the benefit of introducing cluster-state and waiting-time awareness into process-count selection.

For process placement, we compare the proposed node- and NUMA-aware method with three placement baselines:

(1) No affinity. No explicit affinity constraint is specified, and MPI worker Pods are placed by the default Kubernetes scheduler without additional placement constraints.

(2) Kube-HPC-FGS (soft). A preferred co-location constraint is used to encourage MPI worker Pods to be placed on the same physical node when possible, but the scheduler may violate the preference when resources are insufficient.

(3) Kube-HPC-FGS (hard). A required co-location constraint is used to restrict MPI worker Pods to an eligible physical node. A job remains Pending when no node satisfies the affinity constraint and its resource requirements.

The three placement baselines represent no explicit placement constraint, preferred node-level co-location, and required node-level co-location, respectively. Unlike the proposed method, they do not estimate the runnable process capacity of individual NUMA nodes or explicitly include per-process memory-bandwidth demand in the placement decision. The comparison therefore evaluates the additional benefit of resource-aware node- and NUMA-level placement over conventional Kubernetes affinity settings.

For fairness, all compared methods use the same hardware and software environment, NPB Class-D benchmark applications, input sizes, task sequences, cluster sizes, per-node CPU limit, job-submission order, and evaluation metrics. In the parallelism experiments, MP-STEOMP and the proposed method use the same performance model and candidate process-count range. In the placement experiments, all methods use the same process count and per-process resource requests for each MPI job. The same execution and result-reporting procedure is applied to all compared methods. Therefore, the experimental conditions are controlled so that the reported differences primarily reflect the process-count selection rule or placement policy under evaluation.

4.5 Prediction Accuracy

We evaluate the performance and resource models using eight NPB benchmark applications under different process counts and data scales. The performance and resource prediction models achieve MAPEs of approximately 7.3% and 8%, respectively. These results indicate that the models capture the dominant performance and resource trends required for subsequent parallelism selection and process placement. Within the evaluated workloads and cluster configurations, this level of accuracy provides sufficiently reliable estimates for comparing candidate process counts and supporting parallelism-selection decisions. The end-to-end results presented in Sections 4.6 and 4.7 further show that the resulting decisions reduce task-sequence execution time and improve resource utilization, demonstrating the practical value of the models for scheduling even though they do not explicitly reproduce every low-level performance effect. Because the models contain only a small number of fitting parameters, they are lightweight, provide fast predictions, and are suitable for practical deployment. Figure 9 illustrates the fitting results for the MG benchmark at data scale D.

4.6 Parallelism Optimization Results

The cloud-native resource sharing environment brings new challenges to the parallelism setting of MPI tasks. How to reasonably set the task parallelism to minimize the running time of a single task while improving the operating efficiency of the entire task sequence has become the main problem. To this

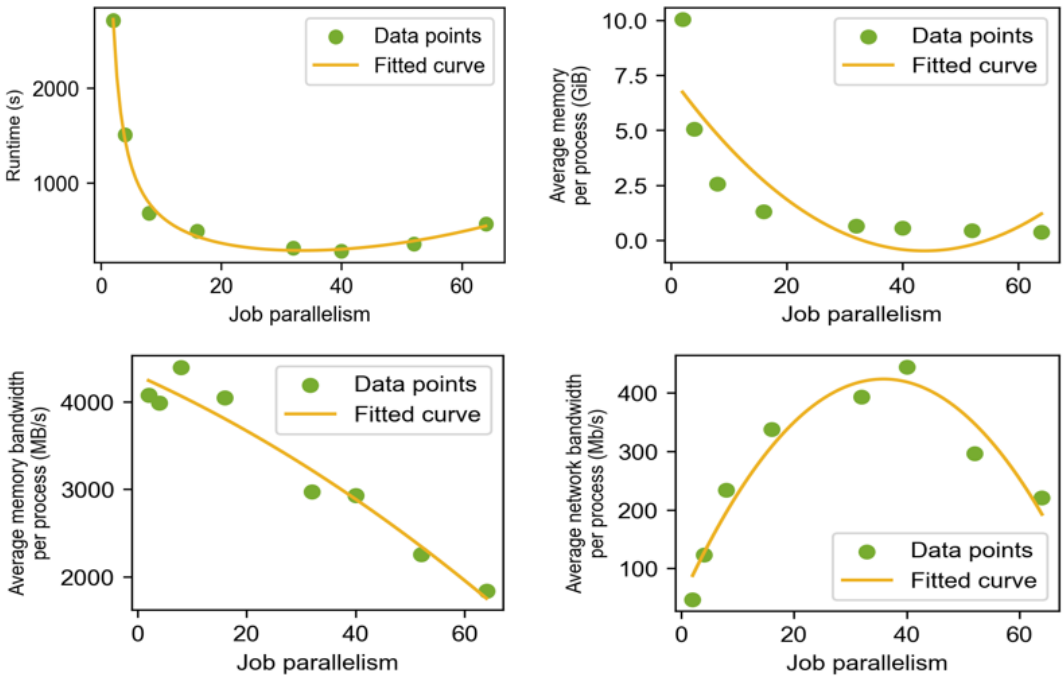

Figure 9: Fitting results of performance and resource prediction models for MG task.

end, the experiment in this section randomly selected 26 tasks from the NPB benchmark suite to form a task sequence to evaluate the robustness of the proposed optimization strategy, and ran them on clusters with 6, 4 and 2 nodes respectively. The corresponding task sequences were named $Trace_1$, $Trace_2$ and $Trace_3$ respectively.

In the specific experimental settings, to eliminate the interference of other services in the cluster on the experimental results, the number of CPUs available for running MPI jobs on each node is limited to 30. To highlight the advancement of the method in this paper, the traditional parallelism setting strategy based on the optimal performance of a single job is selected as the control to ensure a fair comparison under identical resource constraints and workloads.

In terms of task sequence running time, compared with the control method, the computing resource-aware parallelism strategy designed in this paper generally reduces the task execution time by at least 18%. As shown in Figure 10, $Trace_1$ shortens the execution time by 190 seconds, a decrease of 18%; $Trace_2$ shortens by 396 seconds, a decrease of 25%; $Trace_3$ shortens by 699 seconds, a decrease of 22%. This effect is due to the strategy's ability to identify the usage status of cluster resources and reasonably reduce the parallelism of tasks, thereby reducing the waiting time of tasks and greatly reducing the total running time of tasks, which improves the trade-off between single-job performance and sequence-level scheduling efficiency under resource-sharing conditions and reduces Pending-state accumulation during burst submissions. Especially in the case of resource shortage, such as the clusters corresponding to $Trace_2$ and $Trace_3$, due to the limited number of nodes leading to resource shortage, the parallelism adjustment algorithm in this paper can effectively alleviate this problem. By appropriately reducing the task parallelism, tasks can be started as soon as possible, avoiding idle waste of resources, and significantly improving the overall operating efficiency.

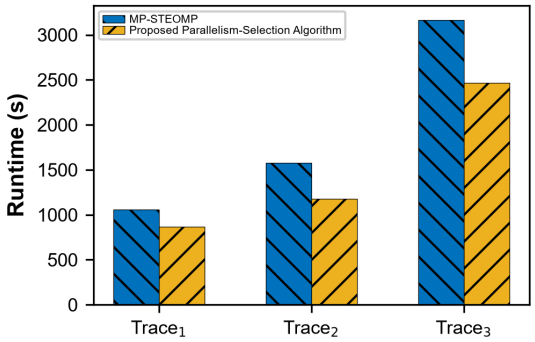

Figure 10: Comparison of total execution time of task sequences under different parallelism setting strategies.

In terms of resource utilization, the computing resource-aware parallelism setting strategy designed in this paper also has obvious advantages. Compared with the control method, the resource utilization rate is generally increased by more than 30%. Specifically, in terms of CPU resource utilization, as shown in Figure 11, the method in this paper is increased by 53%, 57% and 60% respectively compared with the control method; in terms of memory resource utilization, as shown in Figure 12, it is increased by 100%, 53% and 82% respectively; in terms of memory bandwidth resource utilization, as shown in Figure 13, it is increased by 31%, 57% and 53% respectively. The method in this paper has achieved a significant improvement in the utilization of key resources such as CPU, memory and memory bandwidth. This is mainly due to the method's ability to reasonably set the parallelism of tasks according to the real-time resource usage status of the cluster, avoiding the situation where tasks cannot be started due to insufficient resources, reducing the generation of resource gaps, and thus effectively improving the efficiency of resource use.

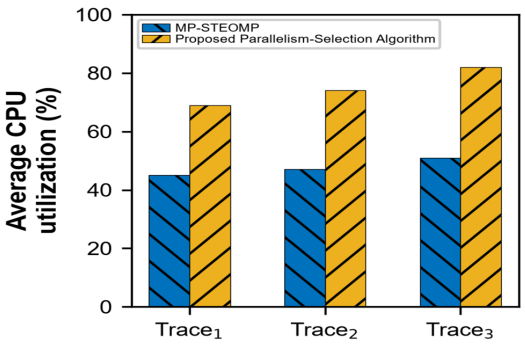

Figure 11: Comparison of cluster CPU utilization under different parallelism setting strategies.

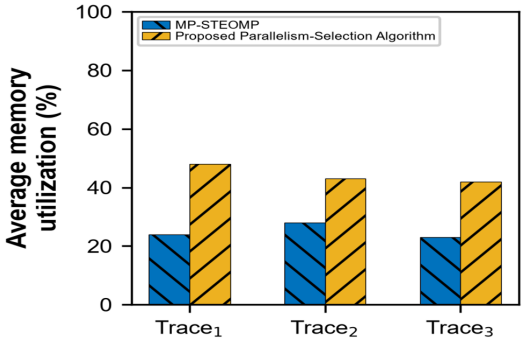

Figure 12: Comparison of cluster memory utilization under different parallelism setting strategies.

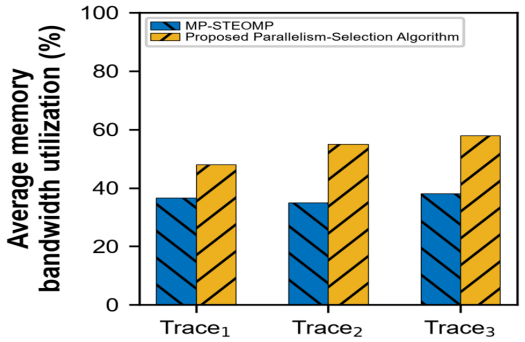

Figure 13: Comparison of cluster memory bandwidth utilization under different parallelism setting strategies.

4.7 Affinity Optimization Results

The affinity characteristics brought by cloud-native technology and the resource sharing environment, intertwined with the resource competition problem caused by the NUMA architecture, have brought severe challenges to the operation of multiple MPI tasks. To verify the effectiveness of the cloud-native MPI job affinity setting method proposed in this paper, the experiment in this section randomly selected 20 tasks from the NPB benchmark suite to form a task sequence, and ran them on clusters with 6, 4 and 2 nodes respectively. The corresponding task sequences are denoted as $Trace_1$, $Trace_2$, and $Trace_3$, respectively, and they are independently generated from those used in Section 4.6.

In the experimental setting, to eliminate the interference of other services in the cluster on the experimental results, the number of CPUs available for running MPI jobs on each node is limited to 30. To highlight the advancement of the method in this paper, three affinity setting methods: no affinity, hard affinity and soft affinity are selected as comparisons.

In terms of the total running time of the task sequence, the computing resource-aware MPI job affinity setting strategy designed in this paper performs well. Compared with other comparison methods, the task execution time is reduced by at least 10%. As shown in Figure 14, in the $Trace_1$ task sequence, the method in this paper reduces the total execution time by 272s, 370s and 231s respectively compared with the comparison methods, with decreases of 11%, 15% and 10% respectively; in $Trace_2$, it reduces by 378s, 322s and 423s respectively, with decreases of 20%, 18% and 22% respectively; in $Trace_3$, it reduces by 497s, 354s and 304s respectively, with decreases of 27%, 21% and 19% respectively. This effect is due to the affinity setting strategy in this paper's ability to accurately perceive the resource usage of NUMA nodes on each node and calculate the number of runnable processes on each NUMA node. On the one hand, it avoids the centralized deployment of processes of multiple tasks on the same NUMA node, thereby reducing the competition for memory bandwidth resources; on the other hand, it reduces cross-node MPI communication between tasks, effectively improving the operating efficiency of tasks. Especially in the execution sequences of $Trace_2$ and $Trace_3$ with more scarce resources, the affinity strategy in this paper has more prominent advantages, and can more effectively guarantee the resources required by tasks and avoid resource competition between multiple tasks in a resource-constrained environment.

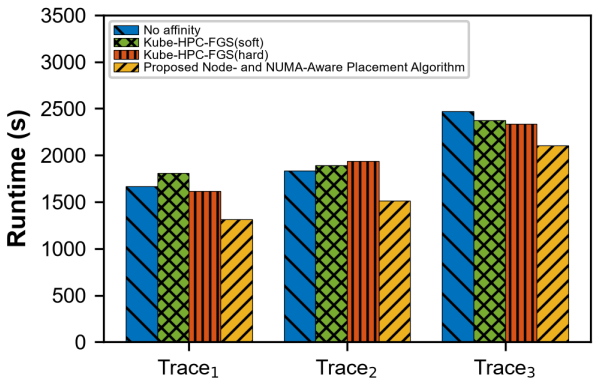

Figure 14: Comparison of task sequence running time under different affinity setting strategies.

In terms of resource utilization, the method in this paper improves the memory bandwidth utilization, which is the most sensitive to MPI tasks, by at least 10%. As shown in Figure 15, in terms of memory bandwidth usage, the affinity setting strategy in this paper is increased by 16%, 10% and 11% respectively in $Trace_1$ compared with other methods; in $Trace_2$, it is increased by 27%, 17% and 24% respectively; in $Trace_3$, it is increased by 42%, 25% and 19% respectively. The reason for the significant improvement in memory bandwidth utilization is that the key factor restricting the operating efficiency of MPI tasks is memory bandwidth. Traditional affinity allocation strategies are mainly based on memory and CPU usage, which easily lead to the demand for memory bandwidth of tasks deployed on the same node exceeding the supply capacity of the node, thereby causing memory bandwidth competition and affecting memory access timing.

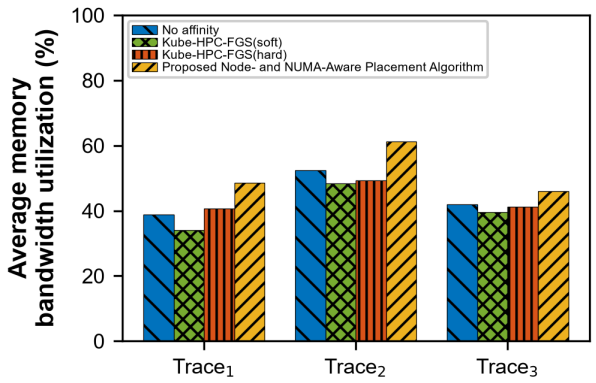

Figure 15: Comparison of cluster memory bandwidth utilization under different affinity setting strategies.

In terms of other resource utilization, the performance of the proposed method in terms of CPU and memory is generally better than the hard-affinity and soft-affinity methods, but slightly lower than the no-affinity method, as shown in Figure 16 and Figure 17. This is because the no-affinity method does not consider the association between tasks and resource usage, and disperses processes to various nodes of the cluster. Although it improves the utilization of CPU and memory to a certain extent, it causes huge network bandwidth consumption, which affects the operating efficiency of tasks.

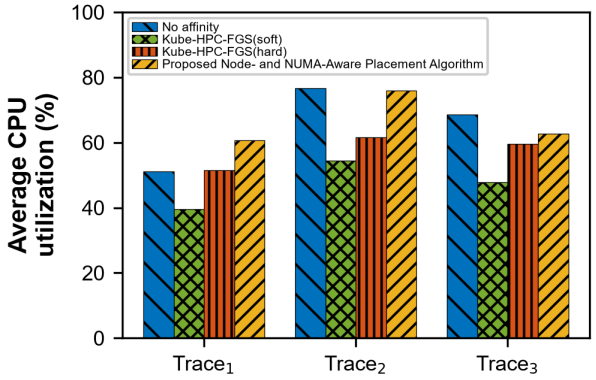

Figure 16: Comparison of in-cluster CPU utilization under different affinity setting strategies.

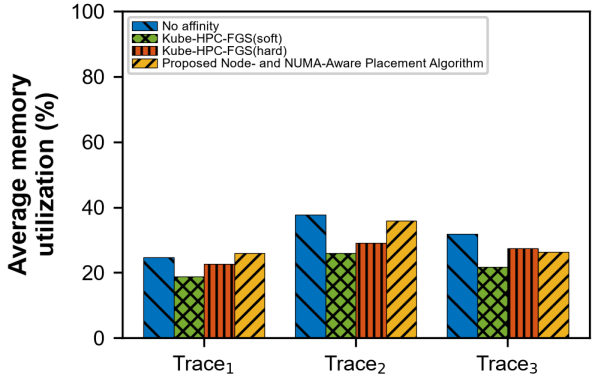

Figure 17: Comparison of cluster memory utilization under different affinity setting strategies.

In terms of network consumption, as shown in Figure 18, the hard affinity method has zero inter-node network traffic because it can only run on a single node; the no-affinity method disperses task processes on various nodes, causing huge network consumption; while the method in this paper can deploy tasks on the minimum number of nodes according to the resource status, effectively balancing resource utilization and network consumption.

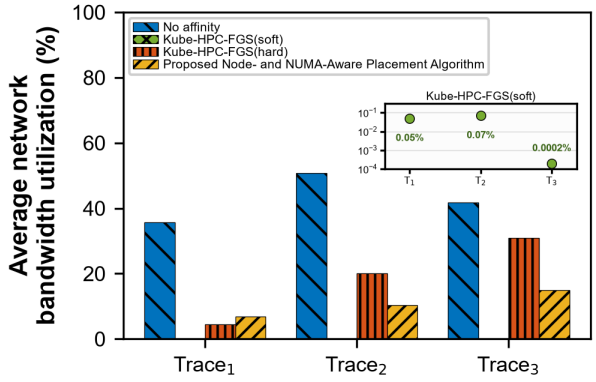

Figure 18: Comparison of cluster network bandwidth utilization under different affinity setting strategies.

4.8 Real-World RTM Case Study

To complement the reproducible NPB evaluation, we conducted a single production-related case study based on an industrial reverse-time migration (RTM) application. The RTM input data were obtained from a real petroleum-geophysical production workflow. For experimental controllability and confidentiality, the performance measurements reported in this section were reproduced on a dedicated test server rather than collected directly from the production system. In addition, the proposed cloud-native MPI approach has been applied in the actual production environment, demonstrating its practical applicability beyond the experimental testbed. The case study includes two workload scales, denoted as RTM-10 and RTM-100, containing 10 and 100 shots, respectively. The corresponding SEG-Y input sizes are approximately 165 MB and 1,650 MB. The approximate problem size of each shot is $600 \times 600 \times 400$. The test platform was equipped with two Intel Xeon Gold 6132 processors, each containing 14 physical cores and supporting 28 hardware threads, two NVIDIA Tesla P100 GPUs with 16 GB of device memory each, 128 GB of system memory, and a 25-Gbps network interface. The cloud-native environment used Kubernetes v1.24.3.

Because the RTM inputs originate from an industrial petroleum-geophysical application, the SEG-Y data, detailed application parameters, internal execution logs, and operational configurations cannot be publicly released because of data-security and confidentiality requirements. We therefore report anonymized workload descriptions, non-sensitive system configurations, and aggregate execution times. The publicly available NPB experiments remain the primary reproducible evidence for the end-to-end scheduling and placement improvements, while the RTM experiment serves as a complementary case study demonstrating practical deployment and process-count scalability in a production application.

Table 2 reports the execution times of the two RTM workload scales under different MPI process counts. For RTM-10, the execution time decreased from 670 s with one process to 350 s with two processes and 242 s with four processes, corresponding to speedups of approximately 1.91× and 2.77×,

respectively. For RTM-100, the execution time decreased from 7374 s with one process to 3729 s, 2128 s, and 1506 s with two, four, and six processes, corresponding to speedups of approximately 1.98×, 3.46×, and 4.90×, respectively.

Table 2: Execution Time of Real-World RTM Workloads under Different Process Counts

Workload	Process Count	Execution Time (s)
RTM-10	1	670
RTM-10	2	350
RTM-10	4	242
RTM-100	1	7374
RTM-100	2	3729
RTM-100	4	2128
RTM-100	6	1506

The execution time decreases consistently as the MPI process count increases for both workload scales. In particular, RTM-100 exhibits close-to-linear speedup over the evaluated process-count range, with parallel efficiencies of approximately 98.9%, 86.6%, and 81.6% at two, four, and six processes, respectively. RTM-10 also benefits substantially from additional processes, although its smaller workload scale leads to earlier diminishing returns at four processes. These results demonstrate that the cloud-native MPI implementation can efficiently utilize additional computing resources and provides good scalability for production RTM workloads.

Beyond the controlled testbed evaluation, the proposed cloud-native MPI approach has also been applied in a real petroleum-geophysical production environment, demonstrating its practical deployability. The testbed results further show that the RTM workflow can efficiently utilize additional computing resources under different MPI process-count configurations. Overall, the case study shows that cloud-native MPI parallelization provides an efficient, scalable, and practically deployable execution approach for real petroleum-geophysical applications.

This experiment represents a single production RTM case study with two workload scales rather than a comprehensive evaluation across multiple production applications or large-scale clusters. The publicly reproducible NPB experiments remain the primary evidence for the end-to-end improvements of the proposed scheduling and placement methods.

5 Conclusion and Future Work

This paper addresses the performance degradation of cloud-native MPI jobs caused by resource sharing and topology-unaware deployment. We proposed a resource-aware optimization framework with two components. The framework selects the process count that minimizes the estimated sum of job waiting time and execution time under the current cluster state and estimates NUMA-level process capacity for topology-aware placement. A Kubernetes prototype was implemented using Custom Resource Definition (CRD) and Operator mechanisms. Experiments using the NPB suite yielded performance and resource prediction MAPEs of 7.3% and 8%, respectively. Compared with the single-job performance-oriented configuration, the proposed parallelism-selection method reduced task-sequence execution time by at least 18% and improved the evaluated resource-utilization metrics by more than 30%. The proposed placement method reduced execution time by at least 10% over the affinity baselines. A complementary production-related RTM case study, using industrial input data and reproduced on a dedicated test server, demonstrates efficient process-count scaling. The application of the proposed cloud-native MPI approach in a real petroleum-geophysical production environment further confirms its practical deployability.

Several limitations remain. First, the prediction models depend on historical execution data and may be less accurate during initial deployment or when previously unseen workloads are introduced. Future work will investigate lightweight online learning and program-level performance modeling to improve cold-start and cross-workload prediction. Second, the current placement mechanism reserves resources according to peak demand, which may cause temporary resource underutilization. Phase-aware resource estimation and dynamic placement adjustment could improve utilization while preserving performance isolation. Finally, the framework can be extended to incorporate additional factors, including network latency, disk I/O, and cache behavior, and can be evaluated on larger clusters and more diverse MPI applications.

Funding

This work was supported by Open Fund (Grant Number: 36750000-24-FW0399-0013) of SINOPEC Key Laboratory of Geophysics and Natural Science Foundation of China(Grant Number: U24B6001).

Author Contributions

W.W.: Conceptualization, Methodology, Software, Formal analysis, Investigation, Validation, and Writing – original draft. Z.G.: Investigation, Visualization, Writing – original draft, and Writing – review and editing. G.J.: Investigation, Visualization, Writing – original draft, and Writing – review. Chentian Yong: Resources, Data curation, Investigation, Validation, Writing – review. B.L.: Project administration, Resources, Software, and Supervision. All authors have read and agreed to the published version of the manuscript.

Conflict of Interest

All the authors declare that they have no conflict of interest.

Data Available

Due to space limitations, additional experimental results including detailed resource utilization curves and ablation study results are available upon request from the corresponding author.

References

[1] Baysal, E., Kosloff, D.D., Sherwood, J.W.C.: Reverse Time Migration. Geophysics **48**(11), 1514–1524 (1983). https://doi.org/10.1190/1.1441434

[2] Zhou, H.-W., Hu, H., Zou, Z., Wo, Y., Youn, O.: Reverse Time Migration: A Prospect of Seismic Imaging Methodology. Earth-Science Reviews **179**, 207–227 (2018). https://doi.org/10.1016/j.earscirev.2018.02.008

[3] Araya-Polo, M., Cabezas, J., Hanzich, M., Pericas, M., Rubio, F., Gelado, I., Shafiq, M., Morancho, E., Navarro, N., Ayguade, E., *et al.*: Assessing Accelerator-Based HPC Reverse Time Migration. IEEE Transactions on Parallel and Distributed Systems **22**(1), 147–162 (2011). https://doi.org/10.1109/TPDS.2010.144

[4] Gabriel, E., Fagg, G.E., Bosilca, G., Angskun, T., Dongarra, J.J., Squyres, J.M., Sahay, V., Kambadur, P., Barrett, B., Lumsdaine, A., *et al.*: Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation. In Recent Advances in Parallel Virtual Machine and Message Passing Interface: the 11th European PVM/MPI Users' Group Meeting (Euro PVM/MPI 2004), pp. 97–104 (2004). https://doi.org/10.1007/978-3-540-30218-6_19

[5] https://www.docker.com/ Accessed 2026-07-22

[6] https://kubernetes.io/ Accessed 2026-07-22

[7] https://spark.apache.org/ Accessed 2026-07-22

[8] Huang, X., Gu, R., Huang, Y.: Towards Efficient Serverless MapReduce Computing on Cloud-Native Platforms. Big Data Mining and Analytics **8**(3), 575–591 (2025). https://doi.org/10.26599/BDMA.2024.9020084

[9] Zhang, X., Wang, X., Shang, J., *et al.*: Efficient Serverless Stream Processing Based on High-Level Programming and Parallelism AutoTuning. In the 2024 IEEE International Conference on High Performance Computing and Communications (HPCC 2024), pp. 474–481 (2024). https://doi.org/10.1109/HPCC64274.2024.00070

[10] Kubeflow: Kubeflow MPI Operator. https://github.com/kubeflow/mpi-operator Accessed 2026-07-22

[11] Kubernetes: Kubernetes Documentation: Assigning Pods to Nodes. https://kubernetes.io/docs/concepts/scheduling-eviction/assign-pod-node/ Accessed 2026-07-22

[12] Kubernetes: Kubernetes Documentation: Control Topology Management Policies on a Node. https://kubernetes.io/docs/tasks/administer-cluster/topology-manager/ Accessed 2026-07-22

[13] Mujkanovic, N., Durillo, J.J., Hammer, N., Müller, T.: Survey of Adaptive Containerization Architectures for HPC. In Proceedings of the SC '23 Workshops of the International Conference on High Performance Computing, Network, Storage, and Analysis, pp. 165–176 (2023). https://doi.org/10.1145/3624062.3624588

[14] Beltre, A.M., Saha, P., Govindaraju, M., Younge, A., Grant, R.E.: Enabling HPC Workloads on Cloud Infrastructure Using Kubernetes Container Orchestration Mechanisms. In 2019 IEEE/ACM International Workshop on Containers and New Orchestration Paradigms for Isolated Environments in HPC (CANOPIE-HPC), pp. 11–20 (2019). https://doi.org/10.1109/CANOPIE-HPC49598.2019.00007

[15] Hursey, J.: Design Considerations for Building and Running Containerized MPI Applications. In the 2020 2nd International Workshop on Containers and New Orchestration Paradigms for Isolated Environments in HPC (CANOPIE-HPC 2020), pp. 35–44 (2020). https://doi.org/10.1109/CANOPIEHPC51917.2020.00010

[16] Zhou, N.: Containerization and Orchestration on HPC Systems. In Sustained Simulation Performance 2019 and 2020: the Joint Workshop on Sustained Simulation Performance (PJWSSP 2019/2020), pp. 133–147 (2021). https://doi.org/10.1007/978-3-030-68049-7_10

[17] Zhang, J., Lu, X., Panda, D.K.: High Performance MPI Library for Container-Based HPC Cloud on InfiniBand Clusters. In 2016 45th International Conference on Parallel Processing (ICPP), pp. 268–277 (2016). https://doi.org/10.1109/ICPP.2016.38

[18] Liu, P., Guitart, J.: Performance Comparison of Multi-Container Deployment Schemes for HPC Workloads: An Empirical Study. The Journal of Supercomputing **77**(6), 6273–6312 (2021). https://doi.org/10.1007/s11227-020-03518-1

[19] Denis, A., Jaeger, J., Jeannot, E., Pérache, M., Taboada, H.: Study on Progress Threads Placement and Dedicated Cores for Overlapping MPI Nonblocking Collectives on Manycore Processor. The International Journal of High Performance Computing Applications **33**(6), 1240–1254 (2019). https://doi.org/10.1177/1094342019860184

[20] Gallardo, E., Vienne, J., Fialho, L., Teller, P., Browne, J.: MPI Advisor: A Minimal Overhead Tool for MPI Library Performance Tuning. In Proceedings of the 22nd European MPI Users' Group Meeting, pp. 1–10 (2015). https://doi.org/10.1145/2802658.2802667

[21] Ashwini, J.P., Sanjay, H.A., Nayana, M.C., Shastry, K.A., K, M.M.M.: Framework for Performance Enhancement of MPI-Based Application on Cloud. Scalable Computing: Practice and Experience **24**(1), 55–67 (2023). https://doi.org/10.12694/scpe.v24i1.2086

[22] Feng, Y., Yuan, J., Liu, J., Lu, Y., Wu, H.: Cross-Domain Collaborative Federated Intelligence for Wireless Computing Power Networks. Journal of Intelligent Computing and Networking **2**(1), 22–34 (2026). https://doi.org/10.64509/jicn.21.75

[23] Chen, W., Deng, D., Xiang, C., Liu, Z., Xiao, B.: When

One-Shot Federated Learning Meets Diffusion Models at the Edge: Technological Advances and Applications. Journal of Intelligent Computing and Networking **2**(1), 35–54 (2026). https://doi.org/10.64509/jicn.21.64

[24] Lameter, C.: NUMA (Non-Uniform Memory Access): An Overview. Queue **11**(7), 40–51 (2013). https://doi.org/10.1145/2508834.2513149

[25] Springer, R., Lowenthal, D.K., Rountree, B., Freeh, V.W.: Minimizing Execution Time in MPI Programs on an Energy-Constrained, Power-Scalable Cluster. In the 2006 Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2006), pp. 230–238 (2006). https://doi.org/10.1145/1122971.1123006

[26] Kumar, M., Kaur, G.: Containerized MPI Application on InfiniBand based HPC: An Empirical Study. In 2022 3rd International Conference for Emerging Technology (INCET), pp. 1–6 (2022). https://doi.org/10.1109/INCET54531.2022.9824366

[27] Souza Filho, P., Bulcão, A., Panetta, J., Lough, M., Monnet, B.: Efficient Execution of MPI Containers. In the 2023 EAGE Seventh High Performance Computing Workshop (HPCW 2023), vol. 2023, pp. 1–5 (2023). https://doi.org/10.3997/2214-4609.2023630030

[28] NASA Advanced Supercomputing Division: NAS Parallel Benchmarks. https://www.nas.nasa.gov/software/npb.html Accessed 2026-07-22

[29] Intel: Intel Performance Counter Monitor (Intel PCM). https://github.com/intel/pcm Accessed 2026-07-22